МИНОБРНАУКИ РОССИИ САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ «ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)

ОТЧЕТ

по лабораторной работе № 2

по дисциплине «Алгоритмы и Структуры данных»

Студент гр. 3351 ________________________Баринов А.А.

Преподаватель ________________________Пестерев Д.О.

Санкт-Петербург

2024

Цель лабораторной работы: реализация самобалансирующихся деревьев поиска и экспериментальная проверка оценок высоты данных деревьев.

Теоретическая часть.

1. Определение АВЛ дерева. Кратко описать алгоритм вставки/удаления с последующей балансировкой для АВЛ-дерева. Получить верхнюю оценку высоты АВЛ-дерева.
2. Определение красно-черного дерева. Кратко описать алгоритм вставки/удаления с последующей балансировкой для красно-черного дерева. Получить верхнюю оценку высоты красно-черного дерева.

1. АВЛ-дерево.

АВЛ-дерево — это двоичное дерево поиска, в котором разность высот левого и правого поддеревьев каждого узла (баланс-фактор) может принимать значения только -1, 0 или 1. Это ограничение сохраняет высоту дерева в пределах O(log n), что обеспечивает эффективное выполнение операций вставки, удаления и поиска.

Алгоритм вставки в АВЛ-дерево:

Для вставки нового элемента в АВЛ-дерево сначала выполняется стандартная операция вставки, как в обычном двоичном дереве поиска: если значение меньше текущего узла, переходят в левое поддерево, если больше — в правое. Новый узел вставляется на подходящее место с высотой, равной 1. После вставки обновляются высоты всех узлов на пути от места вставки до корня. Высота узла вычисляется по формуле: высота узла = 1 + max(высота левого поддерева, высота правого поддерева).

Для каждого узла на этом пути вычисляется баланс-фактор как разность высот левого и правого поддеревьев: баланс-фактор = высота левого поддерева − высота правого поддерева. Если баланс-фактор выходит за пределы диапазона [-1, 1], выполняется балансировка. В зависимости от положения нового узла относительно его предков выполняется одна из следующих ротаций:

- LL-ротация (правое вращение) — если дисбаланс возник в левом поддереве левого ребёнка.
- RR-ротация (левое вращение) — если дисбаланс возник в правом поддереве правого ребёнка.
- LR-ротация (левое вращение, затем правое) — если дисбаланс возник в правом поддереве левого ребёнка.

- RL-ротация (правое вращение, затем левое) — если дисбаланс возник в левом поддереве правого ребёнка.

Алгоритм удаления из АВЛ-дерева:

При удалении элемента из АВЛ-дерева сначала находят узел для удаления. Если узел является листом, его просто удаляют. Если у узла один ребёнок, узел заменяется своим потомком. Если у узла два ребёнка, он заменяется либо его предшественником, либо преемником, после чего дубликат удаляется.

После удаления обновляются высоты всех узлов от места удаления до корня. Высоты вычисляются по формуле: высота узла = 1 + max(высота левого поддерева, высота правого поддерева).

Затем проверяется баланс-фактор каждого узла: баланс-фактор = высота левого поддерева − высота правого поддерева. Если баланс-фактор выходит за пределы диапазона [-1, 1], выполняется соответствующая ротация (LL, RR, LR, RL) для восстановления баланса.

Вывод верхней оценки высоты АВЛ-дерева:

Рекуррентное соотношение для минимального числа узлов
Минимальное количество узлов N(h) для дерева высоты h определяется следующим образом:

$N(h) = 1 + N(h-1) + N(h-2)$, где: $N(0) = 1, N(1) = 2.$

Связь с числами Фибоначчи
Рекуррентное соотношение $N(h)$аналогично определению чисел Фибоначчи:

$F_k = F_{k-1} + F_{k-2}$, где $F_0 = 0, F_1 = 1.$ Отсюда:

$N(h) \geq F_{h-2}$, где F_k— число Фибоначчи.

Для чисел Фибоначчи выполняется асимптотическая формула:
$F_k \sim \frac{\varphi^k}{\sqrt{5}}$,

где $\varphi^k = \frac{1+\sqrt{5}}{2}$ — золотое сечение.

Вывод высоты через число узлов
Подставляя F_{h-2} в выражение для минимального числа узлов, получим:

$N(h) \geq \frac{\varphi^{h+2}}{\sqrt{5}}.$

Решим это неравенство для h:

$n \geq \frac{\varphi^{h+2}}{\sqrt{5}}$, где n — общее количество узлов в дереве. Тогда, применяя свойство логарифма, получим:

$h + 2 \leq \log_{\varphi} \sqrt{5}\, n$, в конечном итоге получаем:

$h \leq \log_{\varphi}(\sqrt{5}\, n) - 2$, что соответствует $h = O(\log n)$.

Таким образом, высота АВЛ-дерева растёт логарифмически относительно количества узлов, обеспечивая эффективность операций вставки, удаления и поиска.

2. Красно-черное дерево.

Красно-черное дерево — это двоичное дерево поиска, которое удовлетворяет следующим свойствам:

1. Каждый узел окрашен либо в красный, либо в черный цвет.
2. Корень дерева всегда черный.
3. Листья (NULL-указатели) считаются черными.
4. Если узел красный, то оба его потомка (если они существуют) должны быть черными (свойство «нет двух подряд красных узлов»).
5. На каждом пути от корня до любого листа должно быть одинаковое количество черных узлов.

Эти свойства обеспечивают балансировку дерева, гарантируя его логарифмическую высоту.

Алгоритм вставки в красно-черное дерево:

При вставке нового узла в красно-черное дерево он всегда добавляется как красный, чтобы не нарушить свойство о равном количестве черных узлов на всех путях. После вставки выполняется проверка и восстановление свойств дерева:

1. Если новый узел — корень дерева, он перекрашивается в черный.
2. Если родитель нового узла черный, никаких действий не требуется, так как свойства дерева не нарушены.
3. Если родитель нового узла красный, возникает конфликт, так как два подряд красных узла запрещены. В этом случае используются следующие действия:

   Перекраска. Если дядя нового узла (брат родителя) тоже красный, родитель и дядя перекрашиваются в черный, а дедушка становится красным. После этого проверка продолжается выше по дереву.

Ротация. Если дядя нового узла черный, выполняются вращения (левое, правое или комбинация) для устранения конфликта. После вращений родитель становится черным, дедушка — красным.

Ротации и перекраски повторяются до тех пор, пока свойства дерева не будут восстановлены.

Алгоритм удаления из красно-черного дерева:

При удалении узла из красно-черного дерева существуют два основных случая:

1. Если удаляемый узел имеет не более одного потомка, он заменяется своим единственным потомком.
2. Если у узла есть два потомка, его заменяют минимальным узлом из правого поддерева (или максимальным из левого поддерева). После замены удаляется дубликат, оставшийся в дереве.

После удаления могут быть нарушены свойства дерева, и выполняются следующие действия:

- Если удаленный узел был красным, никаких изменений не требуется.
- Если удаленный узел был черным, нарушается баланс черных узлов на путях. Для восстановления используются следующие действия:

  Устранение дефицита черного цвета. Если у узла-замены «лишний» черный цвет, он переносится к его родителю и братьям.

  Ротация и перекраска. Если баланс черных узлов нарушен у братьев узла-замены, выполняются ротации (левые, правые или комбинация) и перекраска узлов.

Процесс повторяется до тех пор, пока свойства красно-черного дерева не будут восстановлены.

Вывод верхней оценки высоты красно-черного дерева:

Пусть n— общее количество узлов в дереве, а h — его высота.
Определим чёрную высоту дерева $bh(x)$ как количество чёрных узлов на пути от узла x до любого листа. Для корня дерева чёрная высота обозначается $bh(root)$.

Из свойства равенства чёрной высоты на всех путях и того факта, что каждый красный узел имеет чёрного потомка, следует, что высота h не может быть больше чем удвоенная чёрная высота:

$$h \leq 2 \cdot bh(root).$$

Чёрная высота $bh(root)$ не превышает $\log_2 n + 1$, так как минимальное количество узлов в дереве с чёрной высотой k — это полное чёрное дерево, имеющее$2^k - 1$ узлов. Таким образом, получаем:

$$h \leq 2 \cdot \log_2 n + 1.$$

Таким образом, высота красно-черного дерева растет пропорционально $O(\log n)$.

Практическая часть.

1. Реализовать бинарное дерево поиска, красно-черное дерево и АВЛ дерево (структура, балансировка, операции вставки/удаления/поиска).
2. Получить зависимость высоты дерева поиска от количества ключей, при условии, что значение ключа - случайная величина, распределенная равномерно. Какая асимпотика функции h(n) наблюдается у двоичного дерева поиска?
3. Получить зависимость АВЛ и красно-черного дерева поиска от количества ключей, при условии, что значения ключей монотонно возрастают.
3. Вывести полученные результаты на графики
4. Сравнить с теоретической оценкой высоты.
5. Реализовать обходы в глубину и обход в ширину двоичного дерева с выводом результата.

Бинарное дерево поиска

В заданном условии значение ключа - случайная величина, распределенная равномерно. Равномерное распределение ключей означает, что каждый ключ имеет одинаковую вероятность быть вставленным в любую позицию дерева. Это приводит к тому, что дерево с высокой вероятностью будет сбалансированным, так как вставки будут равномерно распределены по всем уровням дерева

В среднем случае, когда ключи распределены равномерно, сложность операций будет следующей:

Поиск: $O(\log n)$
Вставка: $O(\log n)$

Удаление: $O(\log n)$

В худшем случае двоичное дерево поиска может стать вырожденным, то есть превратиться в линейную структуру (например, когда все узлы добавляются в порядке возрастания или убывания). В этом случае:

Поиск: $O(\log n)$
Вставка: $O(\log n)$
Удаление: $O(\log n)$

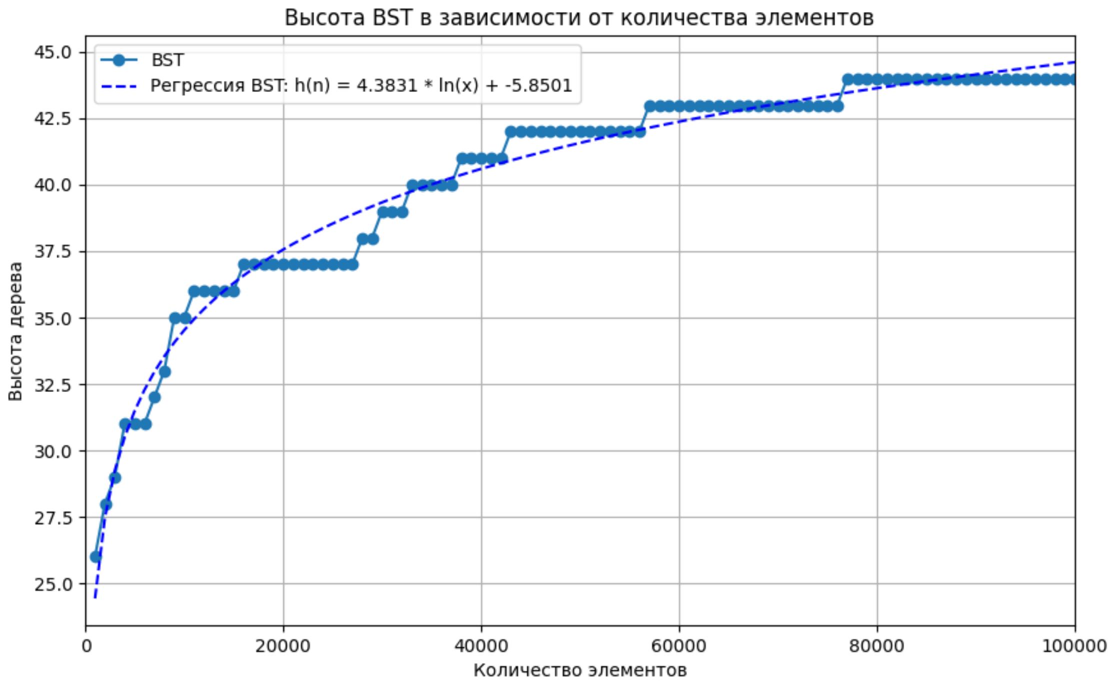

Рис.1 – Зависимость высоты дерева от количества ключей для дерева поиска.

Высота дерева поиска в данном эксперименте подтверждает теоретическую асимптотику $O(\log n)$. На графике видно, что фактические данные хорошо согласуются с логарифмической моделью.

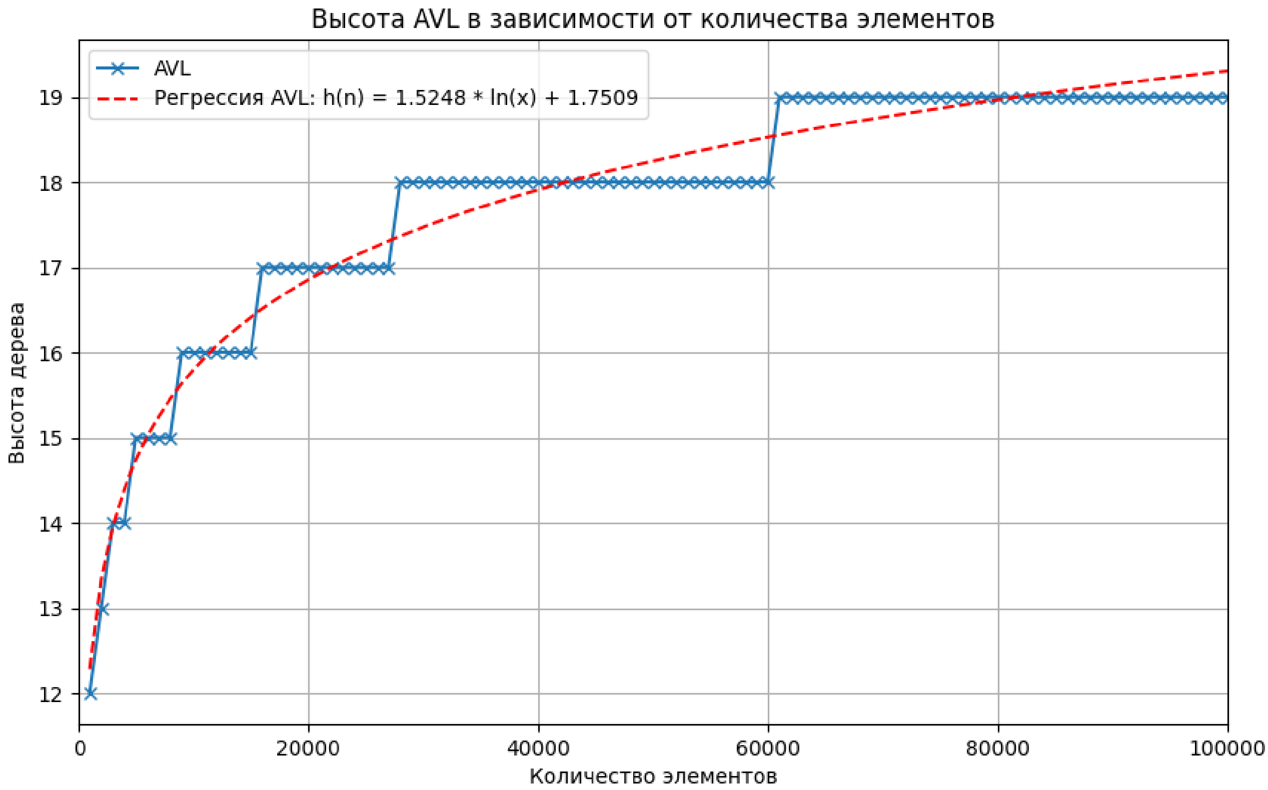

Рис.2 – Зависимость высоты АВЛ дерева от количества.

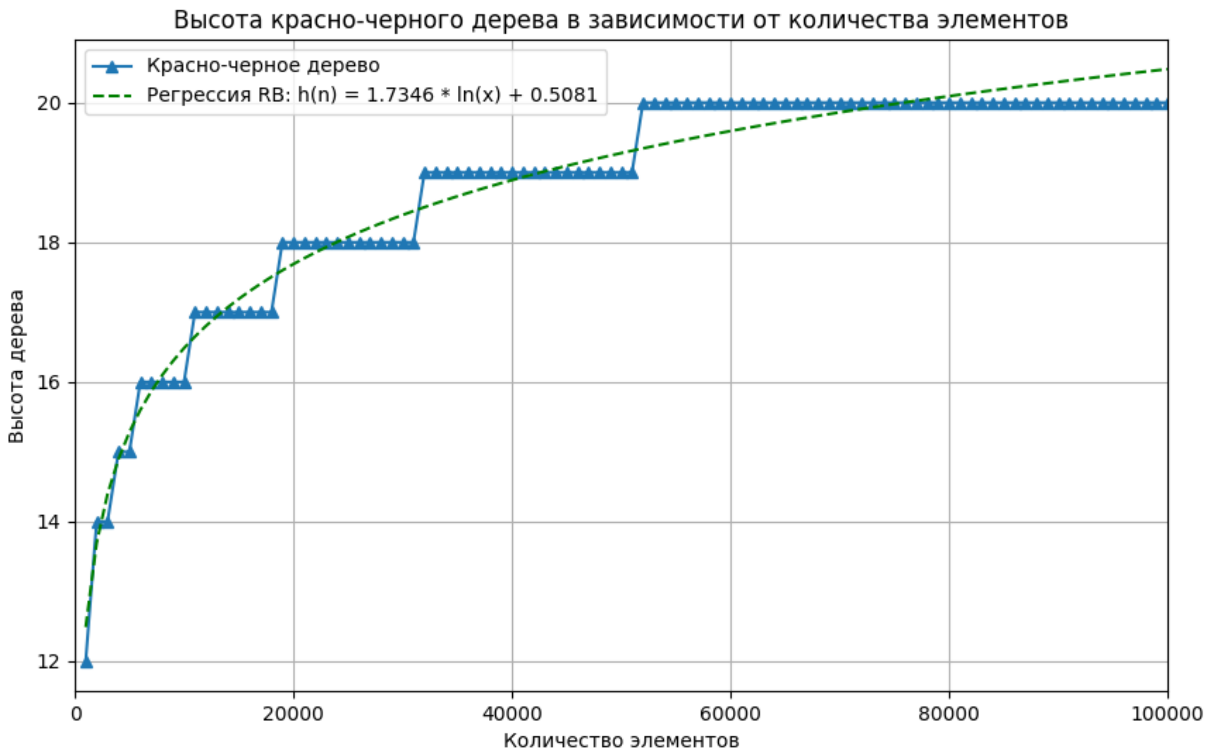

Рис.3 – Зависимость высоты для красно-черного дерева.

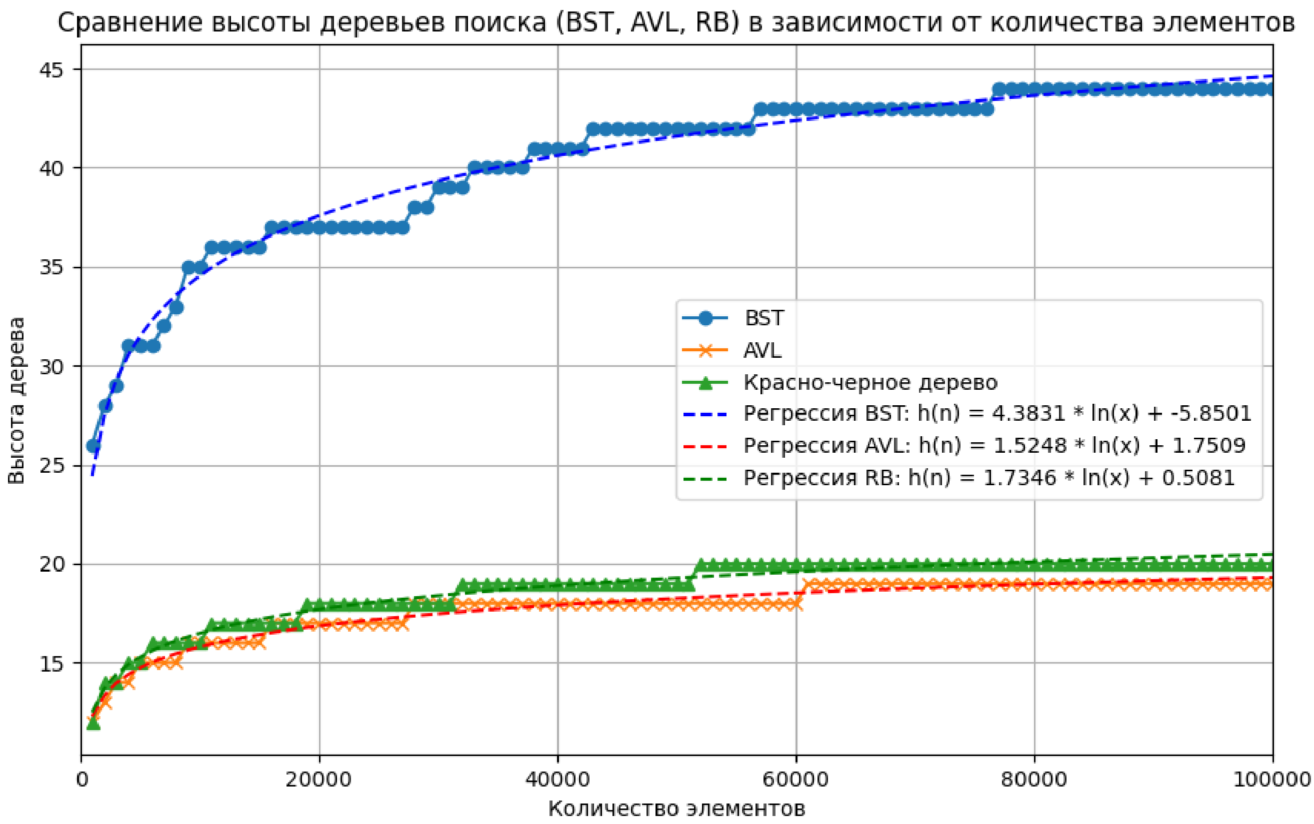

Рис.4 – Зависимость высоты дерева от количества ключей для всех заданных моделей дерева.

BST логарифмическая регрессия: h(n) = 4.3831 * ln(x) + -5.8501
AVL логарифмическая регрессия: h(n) = 1.5248 * ln(x) + 1.7509
RB-Tree логарифмическая регрессия: h(n) = 1.7346 * ln(x) + 0.5081

Высота АВЛ-дерева на графике остается почти постоянной после определенного количества ключей. Это демонстрирует, что дерево стабилизируется на уровне $O(\log n)$, так как АВЛ-дерево сбалансировано. Согласуется с теоретической оценкой.

высота красно-черного дерева также растет медленно, что подтверждает теоретическую оценку $O(\log n)$.

Пример работы обходов в ширину и глубину:

```
C:\Users\Comp\PycharmProjects\python24\venv\Scripts\python.exe C:\Users\Comp\PycharmProjects\pythonProject3\probnik15.py
In-order BST: [175, 248, 439, 579, 719, 740, 780, 813, 896, 919]
In-order AVL: [175, 248, 439, 579, 719, 740, 780, 813, 896, 919]
In-order RB: [175, 248, 439, 579, 719, 740, 780, 813, 896, 919]
-----------
Pre-order BST: [439, 248, 175, 579, 740, 719, 896, 813, 780, 919]
Pre-order AVL: [579, 248, 175, 439, 813, 740, 719, 780, 896, 919]
Pre-order RB: [579, 248, 175, 439, 813, 740, 719, 780, 896, 919]
-----------
Post-order BST: [175, 248, 719, 780, 813, 919, 896, 740, 579, 439]
Post-order AVL: [175, 439, 248, 719, 780, 740, 919, 896, 813, 579]
Post-order RB: [175, 439, 248, 719, 780, 740, 919, 896, 813, 579]
-----------
BFS BST: [439, 248, 579, 175, 740, 719, 896, 813, 919, 780]
BFS AVL: [579, 248, 813, 175, 439, 740, 896, 719, 780, 919]
BFS RB: [579, 248, 813, 175, 439, 740, 896, 719, 780, 919]

Process finished with exit code 0
```

Рис 5 – Пример работы обходов.

Ссылки:

Репозиторий GitHub: https://github.com/barinovartur/algost2.git

Основной код с реализацией деревьев, а также их вывода:

```
import random
import matplotlib.pyplot as plt
import numpy as np
from scipy.optimize import curve_fit

# Узел бинарного дерева поиска
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

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, current, key):
        if key < current.key:
            if current.left is None:
                current.left = Node(key)
            else:
                self._insert(current.left, key)
        else:
            if current.right is None:
                current.right = Node(key)
            else:
                self._insert(current.right, key)

    def height(self):
        return self._height(self.root)

    def _height(self, current):
        if current is None:
            return 0
        return 1 + max(self._height(current.left), self._height(current.right))
```

```
# Узел AVL-дерева
class AVLNode:
  def __init__(self, key):
    self.key = key
    self.left = None
    self.right = None
    self.height = 1

class AVLTree:
  def __init__(self):
    self.root = None

  def insert(self, key):
    self.root = self._insert(self.root, key)

  def _insert(self, current, key):
    if not current:
      return AVLNode(key)
    if key < current.key:
      current.left = self._insert(current.left, key)
    elif key > current.key:
      current.right = self._insert(current.right, key)
    else:
      return current

    current.height = 1 + max(self._get_height(current.left),
self._get_height(current.right))
    balance = self._get_balance(current)

    if balance > 1 and key < current.left.key:
      return self._rotate_right(current)
    if balance < -1 and key > current.right.key:
      return self._rotate_left(current)
    if balance > 1 and key > current.left.key:
      current.left = self._rotate_left(current.left)
      return self._rotate_right(current)
    if balance < -1 and key < current.right.key:
      current.right = self._rotate_right(current.right)
      return self._rotate_left(current)

    return current

  def _rotate_left(self, z):
    y = z.right
    T2 = y.left
```

```
        y.left = z
        z.right = T2
        z.height = 1 + max(self._get_height(z.left), self._get_height(z.right))
        y.height = 1 + max(self._get_height(y.left), self._get_height(y.right))
        return y

    def _rotate_right(self, z):
        y = z.left
        T3 = y.right
        y.right = z
        z.left = T3
        z.height = 1 + max(self._get_height(z.left), self._get_height(z.right))
        y.height = 1 + max(self._get_height(y.left), self._get_height(y.right))
        return y

    def _get_height(self, current):
        if not current:
            return 0
        return current.height

    def _get_balance(self, current):
        if not current:
            return 0
        return self._get_height(current.left) - self._get_height(current.right)

    def height(self):
        return self._get_height(self.root)

# Узел красно-черного дерева
class RBNode:
    def __init__(self, key, color="RED"):
        self.key = key
        self.left = None
        self.right = None
        self.parent = None
        self.color = color

class RBTree:
    def __init__(self):
        self.NIL = RBNode(key=None, color="BLACK")
        self.root = self.NIL

    def insert(self, key):
        new_node = RBNode(key)
        new_node.left = self.NIL
```

```
        new_node.right = self.NIL
        self._insert(new_node)

    def _insert(self, z):
        y = None
        x = self.root
        while x != self.NIL:
            y = x
            if z.key < x.key:
                x = x.left
            else:
                x = x.right

        z.parent = y
        if y is None:
            self.root = z
        elif z.key < y.key:
            y.left = z
        else:
            y.right = z

        z.color = "RED"
        self._fix_insert(z)

    def _fix_insert(self, z):
        while z != self.root and z.parent.color == "RED":
            if z.parent == z.parent.parent.left:
                y = z.parent.parent.right
                if y.color == "RED":
                    z.parent.color = "BLACK"
                    y.color = "BLACK"
                    z.parent.parent.color = "RED"
                    z = z.parent.parent
                else:
                    if z == z.parent.right:
                        z = z.parent
                        self._rotate_left(z)
                    z.parent.color = "BLACK"
                    z.parent.parent.color = "RED"
                    self._rotate_right(z.parent.parent)
            else:
                y = z.parent.parent.left
                if y.color == "RED":
                    z.parent.color = "BLACK"
                    y.color = "BLACK"
```

```
            z.parent.parent.color = "RED"
            z = z.parent.parent
        else:
            if z == z.parent.left:
                z = z.parent
                self._rotate_right(z)
            z.parent.color = "BLACK"
            z.parent.parent.color = "RED"
            self._rotate_left(z.parent.parent)
    self.root.color = "BLACK"

def _rotate_left(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

def _rotate_right(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

def height(self):
    def _height(node):
        if node == self.NIL:
            return 0
```

```
        left_height = _height(node.left)
        right_height = _height(node.right)
        return max(left_height, right_height) + 1

    return _height(self.root)

# Построение графиков
def build_and_plot():
  bst = BST()
  avl = AVLTree()
  rb = RBTree()

  keys = [random.randint(1, 100000) for _ in range(100000)]  # увеличиваем до
100000
  bst_heights = []
  avl_heights = []
  rb_heights = []
  x_vals = range(1000, 100001, 1000)

  for i, key in enumerate(keys):
    bst.insert(key)
    avl.insert(key)
    rb.insert(key)

    if (i + 1) % 1000 == 0:
      bst_heights.append(bst.height())
      avl_heights.append(avl.height())
      rb_heights.append(rb.height())

  # Функция для логарифмической регрессии
  def log_func(x, a, b):
    return a * np.log(x) + b

  # Регрессия для BST
  bst_params, _ = curve_fit(log_func, x_vals, bst_heights)
  bst_equation = f"h(n) = {bst_params[0]:.4f} * ln(x) + {bst_params[1]:.4f}"
  print(f"BST логарифмическая регрессия: {bst_equation}")

  # Регрессия для AVL
  avl_params, _ = curve_fit(log_func, x_vals, avl_heights)
  avl_equation = f"h(n) = {avl_params[0]:.4f} * ln(x) + {avl_params[1]:.4f}"
  print(f"AVL логарифмическая регрессия: {avl_equation}")

  # Регрессия для RB
  rb_params, _ = curve_fit(log_func, x_vals, rb_heights)
```

```
    rb_equation = f"h(n) = {rb_params[0]:.4f} * ln(x) + {rb_params[1]:.4f}"
    print(f"RB-Tree логарифмическая регрессия: {rb_equation}")

    # Построение графиков
    plt.figure(figsize=(10, 6))
    plt.plot(x_vals, bst_heights, label="BST", marker='o')
    plt.plot(x_vals, log_func(np.array(x_vals), *bst_params), linestyle='--', color='b',
label=f"Регрессия BST: {bst_equation}")
    plt.xlabel('Количество элементов')
    plt.ylabel('Высота дерева')
    plt.title('Высота BST в зависимости от количества элементов')
    plt.legend()
    plt.grid(True)
    plt.xlim(0, 100000)
    plt.show()

    plt.figure(figsize=(10, 6))
    plt.plot(x_vals, avl_heights, label="AVL", marker='x')
    plt.plot(x_vals, log_func(np.array(x_vals), *avl_params), linestyle='--', color='r',
label=f"Регрессия AVL: {avl_equation}")
    plt.xlabel('Количество элементов')
    plt.ylabel('Высота дерева')
    plt.title('Высота AVL в зависимости от количества элементов')
    plt.legend()
    plt.grid(True)
    plt.xlim(0, 100000)
    plt.show()

    plt.figure(figsize=(10, 6))
    plt.plot(x_vals, rb_heights, label="Красно-черное дерево", marker='^')
    plt.plot(x_vals, log_func(np.array(x_vals), *rb_params), linestyle='--', color='g',
label=f"Регрессия RB: {rb_equation}")
    plt.xlabel('Количество элементов')
    plt.ylabel('Высота дерева')
    plt.title('Высота красно-черного дерева в зависимости от количества
элементов')
    plt.legend()
    plt.grid(True)
    plt.xlim(0, 100000)
    plt.show()

    # Совместный график
    plt.figure(figsize=(10, 6))
    plt.plot(x_vals, bst_heights, label="BST", marker='o')
    plt.plot(x_vals, avl_heights, label="AVL", marker='x')
```

```
    plt.plot(x_vals, rb_heights, label="Красно-черное дерево", marker='^')
    plt.plot(x_vals, log_func(np.array(x_vals), *bst_params), linestyle='--', color='b',
label=f"Регрессия BST: {bst_equation}")
    plt.plot(x_vals, log_func(np.array(x_vals), *avl_params), linestyle='--', color='r',
label=f"Регрессия AVL: {avl_equation}")
    plt.plot(x_vals, log_func(np.array(x_vals), *rb_params), linestyle='--', color='g',
label=f"Регрессия RB: {rb_equation}")
    plt.xlabel('Количество элементов')
    plt.ylabel('Высота дерева')
    plt.title('Сравнение высоты деревьев поиска (BST, AVL, RB) в зависимости от
количества элементов')
    plt.legend()
    plt.grid(True)
    plt.xlim(0, 100000)
    plt.show()

build_and_plot()
```

Код с реализацией обходов:

```
import random
import matplotlib.pyplot as plt
import numpy as np
from scipy.optimize import curve_fit
from collections import deque

# Узел бинарного дерева поиска
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

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, current, key):
        if key < current.key:
```

```
        if current.left is None:
            current.left = Node(key)
        else:
            self._insert(current.left, key)
    else:
        if current.right is None:
            current.right = Node(key)
        else:
            self._insert(current.right, key)

def height(self):
    return self._height(self.root)

def _height(self, current):
    if current is None:
        return 0
    return 1 + max(self._height(current.left), self._height(current.right))

# Симметричный обход (In-order)
def in_order(self):
    result = []
    self._in_order(self.root, result)
    return result

def _in_order(self, node, result):
    if node:
        self._in_order(node.left, result)
        result.append(node.key)
        self._in_order(node.right, result)

# Прямой обход (Pre-order)
def pre_order(self):
    result = []
    self._pre_order(self.root, result)
    return result

def _pre_order(self, node, result):
    if node:
        result.append(node.key)
        self._pre_order(node.left, result)
        self._pre_order(node.right, result)

# Обратный обход (Post-order)
def post_order(self):
    result = []
```

```
        self._post_order(self.root, result)
        return result

    def _post_order(self, node, result):
        if node:
            self._post_order(node.left, result)
            self._post_order(node.right, result)
            result.append(node.key)

    # Обход в ширину (BFS)
    def bfs(self):
        result = []
        if not self.root:
            return result
        queue = deque([self.root])
        while queue:
            node = queue.popleft()
            result.append(node.key)
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)
        return result

# Узел AVL-дерева
class AVLNode:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
        self.height = 1

class AVLTree:
    def __init__(self):
        self.root = None

    def insert(self, key):
        self.root = self._insert(self.root, key)

    def _insert(self, current, key):
        if not current:
            return AVLNode(key)
        if key < current.key:
            current.left = self._insert(current.left, key)
        elif key > current.key:
```

```
            current.right = self._insert(current.right, key)
        else:
            return current

        current.height = 1 + max(self._get_height(current.left), self._get_height(current.right))
        balance = self._get_balance(current)

        if balance > 1 and key < current.left.key:
            return self._rotate_right(current)
        if balance < -1 and key > current.right.key:
            return self._rotate_left(current)
        if balance > 1 and key > current.left.key:
            current.left = self._rotate_left(current.left)
            return self._rotate_right(current)
        if balance < -1 and key < current.right.key:
            current.right = self._rotate_right(current.right)
            return self._rotate_left(current)

        return current

    def _rotate_left(self, z):
        y = z.right
        T2 = y.left
        y.left = z
        z.right = T2
        z.height = 1 + max(self._get_height(z.left), self._get_height(z.right))
        y.height = 1 + max(self._get_height(y.left), self._get_height(y.right))
        return y

    def _rotate_right(self, z):
        y = z.left
        T3 = y.right
        y.right = z
        z.left = T3
        z.height = 1 + max(self._get_height(z.left), self._get_height(z.right))
        y.height = 1 + max(self._get_height(y.left), self._get_height(y.right))
        return y

    def _get_height(self, current):
        if not current:
            return 0
        return current.height

    def _get_balance(self, current):
        if not current:
```

```
        return 0
    return self._get_height(current.left) - self._get_height(current.right)

def height(self):
    return self._get_height(self.root)

# Симметричный обход (In-order)
def in_order(self):
    result = []
    self._in_order(self.root, result)
    return result

def _in_order(self, node, result):
    if node:
        self._in_order(node.left, result)
        result.append(node.key)
        self._in_order(node.right, result)

# Прямой обход (Pre-order)
def pre_order(self):
    result = []
    self._pre_order(self.root, result)
    return result

def _pre_order(self, node, result):
    if node:
        result.append(node.key)
        self._pre_order(node.left, result)
        self._pre_order(node.right, result)

# Обратный обход (Post-order)
def post_order(self):
    result = []
    self._post_order(self.root, result)
    return result

def _post_order(self, node, result):
    if node:
        self._post_order(node.left, result)
        self._post_order(node.right, result)
        result.append(node.key)

# Обход в ширину (BFS)
def bfs(self):
    result = []
```

```
        if not self.root:
            return result
        queue = deque([self.root])
        while queue:
            node = queue.popleft()
            result.append(node.key)
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)
        return result

# Узел красно-черного дерева
class RBNode:
    def __init__(self, key, color="RED"):
        self.key = key
        self.left = None
        self.right = None
        self.parent = None
        self.color = color

class RBTree:
    def __init__(self):
        self.NIL = RBNode(key=None, color="BLACK")
        self.root = self.NIL

    def insert(self, key):
        new_node = RBNode(key)
        new_node.left = self.NIL
        new_node.right = self.NIL
        self._insert(new_node)

    def _insert(self, z):
        y = None
        x = self.root
        while x != self.NIL:
            y = x
            if z.key < x.key:
                x = x.left
            else:
                x = x.right

        z.parent = y
        if y is None:
            self.root = z
```

```
        elif z.key < y.key:
            y.left = z
        else:
            y.right = z

        z.color = "RED"
        self._fix_insert(z)

    def _fix_insert(self, z):
        while z != self.root and z.parent.color == "RED":
            if z.parent == z.parent.parent.left:
                y = z.parent.parent.right
                if y.color == "RED":
                    z.parent.color = "BLACK"
                    y.color = "BLACK"
                    z.parent.parent.color = "RED"
                    z = z.parent.parent
                else:
                    if z == z.parent.right:
                        z = z.parent
                        self._rotate_left(z)
                    z.parent.color = "BLACK"
                    z.parent.parent.color = "RED"
                    self._rotate_right(z.parent.parent)
            else:
                y = z.parent.parent.left
                if y.color == "RED":
                    z.parent.color = "BLACK"
                    y.color = "BLACK"
                    z.parent.parent.color = "RED"
                    z = z.parent.parent
                else:
                    if z == z.parent.left:
                        z = z.parent
                        self._rotate_right(z)
                    z.parent.color = "BLACK"
                    z.parent.parent.color = "RED"
                    self._rotate_left(z.parent.parent)
        self.root.color = "BLACK"

    def _rotate_left(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

    def _rotate_right(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

    def height(self):
        def _height(node):
            if node == self.NIL:
                return 0
            left_height = _height(node.left)
            right_height = _height(node.right)
            return max(left_height, right_height) + 1

        return _height(self.root)

    # Симметричный обход (In-order)
    def in_order(self):
        result = []
        self._in_order(self.root, result)
        return result

    def _in_order(self, node, result):
        if node != self.NIL:
            self._in_order(node.left, result)
            result.append(node.key)
```

```
            self._in_order(node.right, result)

    # Прямой обход (Pre-order)
    def pre_order(self):
        result = []
        self._pre_order(self.root, result)
        return result

    def _pre_order(self, node, result):
        if node != self.NIL:
            result.append(node.key)
            self._pre_order(node.left, result)
            self._pre_order(node.right, result)

    # Обратный обход (Post-order)
    def post_order(self):
        result = []
        self._post_order(self.root, result)
        return result

    def _post_order(self, node, result):
        if node != self.NIL:
            self._post_order(node.left, result)
            self._post_order(node.right, result)
            result.append(node.key)

    # Обход в ширину (BFS)
    def bfs(self):
        result = []
        if self.root == self.NIL:
            return result
        queue = deque([self.root])
        while queue:
            node = queue.popleft()
            result.append(node.key)
            if node.left != self.NIL:
                queue.append(node.left)
            if node.right != self.NIL:
                queue.append(node.right)
        return result

# Построение графиков
def build_and_plot():
    bst = BST()
    avl = AVLTree()
```

```
rb = RBTree()

for _ in range(10):  # 10 случайных чисел
    key = random.randint(1, 1000)
    bst.insert(key)
    avl.insert(key)
    rb.insert(key)

bst_in_order = bst.in_order()
avl_in_order = avl.in_order()
rb_in_order = rb.in_order()

bst_pre_order = bst.pre_order()
avl_pre_order = avl.pre_order()
rb_pre_order = rb.pre_order()

bst_post_order = bst.post_order()
avl_post_order = avl.post_order()
rb_post_order = rb.post_order()

bst_bfs = bst.bfs()
avl_bfs = avl.bfs()
rb_bfs = rb.bfs()

print(f"In-order BST: {bst_in_order}")
print(f"In-order AVL: {avl_in_order}")
print(f"In-order RB: {rb_in_order}")

print(f"-----------")

print(f"Pre-order BST: {bst_pre_order}")
print(f"Pre-order AVL: {avl_pre_order}")
print(f"Pre-order RB: {rb_pre_order}")

print(f"-----------")

print(f"Post-order BST: {bst_post_order}")
print(f"Post-order AVL: {avl_post_order}")
print(f"Post-order RB: {rb_post_order}")

print(f"-----------")

print(f"BFS BST: {bst_bfs}")
print(f"BFS AVL: {avl_bfs}")
print(f"BFS RB: {rb_bfs}")
```

```
# Вызов функции для построения и вывода результатов
build_and_plot()
```