**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра САПР**

отчет

**по лабораторной работе №2**

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

Тема: Самобалансирующиеся двоичные деревья поиска

|  |  |  |
| --- | --- | --- |
| Студент гр. 3353 |  | Вотинов А.С. |
| Преподаватель |  | Пестерев Д.О. |

Санкт-Петербург

2024

**Содержание**

Цель работы……………………………………………………………………….3

AVL-дерево……………………………………………………………………...3-5

Красно-черное дерево…………………………………………………………..6-9

Практическая часть……………………………………………………………9-15

Вывод………………………………………………………………………….15-16

Листинг………………………………………………………………………..16-56

Ссылка на git https://github.com/Votinov123/AISD\_lab\_2

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

В первую очередь, хотелось бы рассказать о двоичных деревьях.

**AVL-дерево -** это самобалансирующееся бинарное дерево поиска, в котором для каждого узла поддерживается фактор баланса. Этот фактор равен разнице высот левого и правого поддеревьев и может принимать значения -1, 0 или 1. Если разница превышает эти значения, дерево автоматически балансируется с помощью вращений.

Дерево поддерживает свою сбалансированность после каждой операции вставки или удаления. Такой параметр позволяет обеспечить логарифмическое время выполнения поиска элемента.

**Вставка в AVL-дерево:**

Вставка нового узла в AVL-дерево выполняется так же, как в обычном бинарном дереве поиска. Сравниваем ключ нового узла с ключом текущего узла и перемещаемся в левое или правое поддерево, пока не найдем место для вставки. После вставки нового узла обновляем высоту каждого узла, начиная с места вставки и двигаясь вверх к корню. Далее необходимо проверить и обновить коэффициент баланса для каждого узла на пути от вставленного узла к корню.

**Балансировка дерева:**

Если после обновления коэффициента баланса у какого-либо узла он выходит за пределы допустимых значений, необходимо выполнить вращение для восстановления сбалансированности дерева. Существует четыре случая:

* Правый поворот: выполняется при добавлении узла в левое поддерево левого потомка.
* Левый поворот: выполняется при добавлении узла в правое поддерево правого потомка.
* Большой правый поворот: выполняется при добавлении узла в правое поддерево левого потомка.
* Большой левый поворот: выполняется при добавлении узла в левое поддерево правого потомка.

**Удаление из AVL-дерева:**

Сначала находим узел с заданным ключом, а затем удаляем его. Если удаляемый узел имеет двух потомков, заменяем его значением минимального узла из правого поддерева и затем удаляем этот минимальный узел. После удаления необходимо обновить высоту и коэффициент баланса для каждого узла на пути от удаленного узла к корню. Аналогично вставке, если у какого-либо узла коэффициент баланса выходит за пределы допустимых значений, выполняем соответствующее вращение для восстановления сбалансированности.

**Пример вставки:**

Есть исходное дерево:

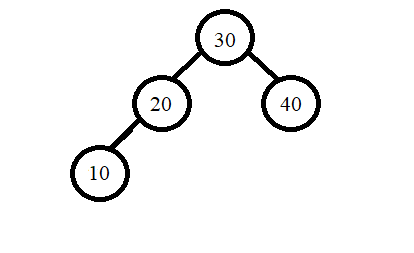


Рис.1

Хотим вставить узел с ключом 5. Тогда он переместится в левое поддерево как левый потомок узла 10, получим следующее дерево:

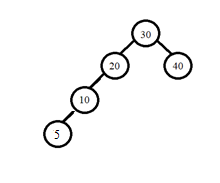
****

Рис.2

Видим, что теперь у узла 30 фактор баланса равен 2, что вынуждает нас сделать правый поворот для балансировки. Тогда узел 20 станет корнем, узел 30 станет его правым потомком и 40 станет правым потомком узла 30, а 10 останется левым потомком узла 20, 5 – левый потомок 20. Дерево сбалансировано:

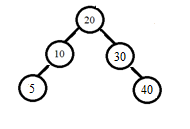
****

Рис.3

**Пример удаления:**

Теперь удалим узел 10 из дерева. Так как он является простым листом, его можно спокойно удалить:

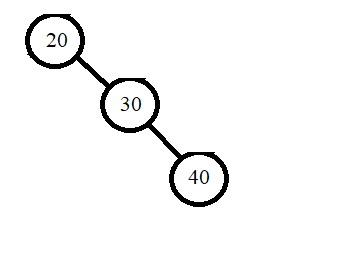


Рис.4

Видим, что баланс дерева нарушен, проведем левый поворот:

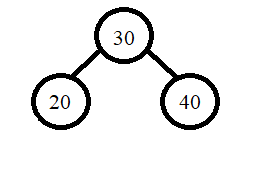


Рис.5

**Получим верхнюю оценку высоты:**

Есть AVL-дерево высоты h. Пусть mh – минимальное число вершин в таком дереве. В качестве ключевого момента заметим, что работа данной структуры напрямую зависит от чисел Фибоначчи.

Можно заметить: mh+2=mh+1+mh+1. Докажем зависимость этого числа от чисел Фибоначчи (Мат. индукция).

БИ: m1=F3-1 – верно, тогда предположим, что Fh+2 – 1 – верно.

Тогда mh+1 = mh+mh-1+1=Fh+2+Fh+1-1-1+1=Fh+3-1, тогда Fh+2 – 1 – верно.

Применив формулу Бине, можем получить следующее соотношение:

mh = Fh+3 – 1 – (1/sqrt(5))(((1+sqrt(5))/2)^(h+3)- ((1-sqrt(5))/2)^(h+3))

При таком соотношении, mh ≈ (1+sqrt(5))/2)^(h+3), а количество элементов в таком дереве: nh >= (1/sqrt(5))((1+sqrt(5))/2)^(h+3).

Применив логарифмирование, получим: h+3<=Clog(n).

Вернемся к тому, что количество элементов больше или равно Fh+3, тогда это также больше или равно сокращенному варианту по формуле Бине:

n>= (1/sqrt(5))(1+sqrt(5))/2)^(h+3)

log(n)>=(h+3)log((1+sqrt(5))/2)

После того, как раскроем скобки и представим некоторые выражения в форме констант: 1.44log(n)-1.5>=h => h=O(logn)

**Красно-чёрное дерево** - это тип самобалансирующегося бинарного дерева поиска, которое обеспечивает логарифмическую высоту и позволяет эффективно выполнять основные операции, такие как вставка, удаление и поиск узлов. Каждый узел может быть либо красным, либо черным, корень дерева всегда черный, все листья также черные и не содержат данных. Красный узел не может иметь красных потомков, тогда оба потомка красного узла должны быть черными. Для любого узла все пути от этого узла до его листьев содержат одинаковое количество черных узлов – это **чёрная высота**. Это свойство помогает поддерживать сбалансированность дерева. Красно-черные деревья обеспечивают гарантированную логарифмическую высоту.

**Вставка в красно-чёрное дерево:**

Новый узел изначально окрашивается в красный цвет. Если новый узел становится корнем, он перекрашивается в черный.

**Балансировка при вставке:**

После вставки нового узла необходимо проверить, не нарушаются ли свойства красно-черного дерева. Если родитель нового узла также красный, это нарушает свойство, когда оба потомка красного узла должны быть черными. В зависимости от расположения узлов (левый или правый потомок) выполняются различные действия:

* Если родитель и дядя (брат родителя) также красные, оба перекрашиваются в черный, а дедушка — в красный, и процесс продолжается вверх по дереву.
* Если дядя черный, выполняется поворот (левый или правый), чтобы восстановить баланс.

**Удаление из красно-чёрного дерева**

Удаление узла происходит как в обычном бинарном дереве поиска. Если удаляемый узел имеет двух потомков, его значение заменяется на минимальное значение из правого поддерева или максимальное из левого поддерева. Далее восстанавливаются свойства красно-черного дерева.

**Балансировка при удалении:**

После удаления необходимо проверить, не нарушаются ли свойства дерева. Если удаленный узел был черным, это может нарушить черную высоту.

В зависимости от ситуации выполняются различные действия:

* Если брат удаленного узла красный, он перекрашивается в черный, а родитель — в красный, после чего выполняется поворот на родителе.
* Если брат черный и имеет хотя бы одного красного ребенка, выполняется поворот (левый или правый, это зависит от того, где красный потомок) и перекраска.

**Пример вставки:**

Есть исходное дерево:

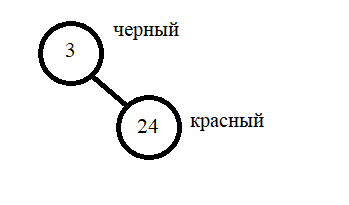
****

Рис.6

Хотим вставить узел с ключом 35. Тогда он станет правым потомком узла 24, получим следующее дерево:

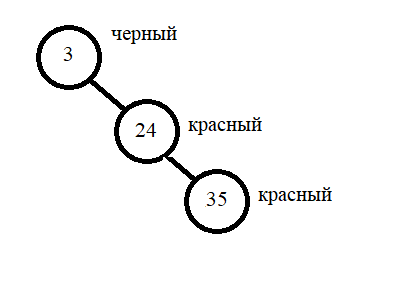
****

Рис.7

Видим, что условия красно-черного дерева нарушены, поэтому совершим левый поворот относительно родителя 35 (24) и перекрасим его в черный, чтобы также сохранить условия:

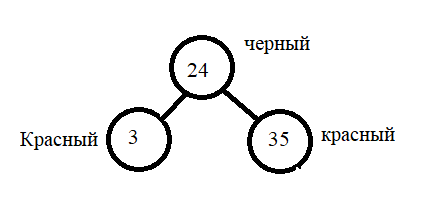
****

Рис.8

Если мы захотим вставить, например, узел с номером 19, то он спокойно станет правым потомком 3, но при этом опять нарушатся условия – нужно перекрасить родителя и дядю в черный цвет:

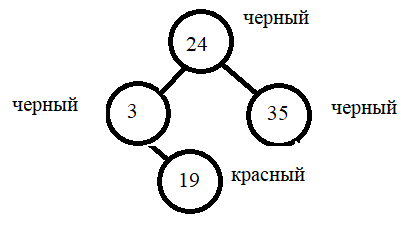
****

Рис.9

**Пример удаления:**

Есть исходное дерево:

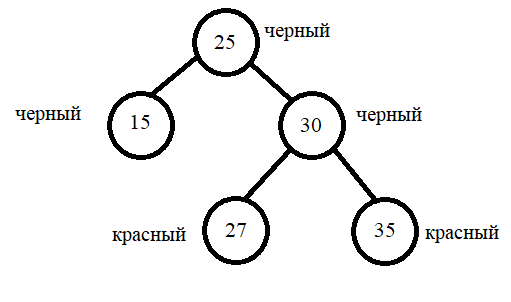


Рис.10

Хотим удалить узел 15, он спокойно будет исключен, но при этом нарушатся условия красно-черного дерева. Для решения данной проблемы, выполним левый поворот относительно правого потомка корня и после этого перекрасим узел 35 в черный цвет, чтобы сохранить условия, а узел 27 станет правым потомком 25:

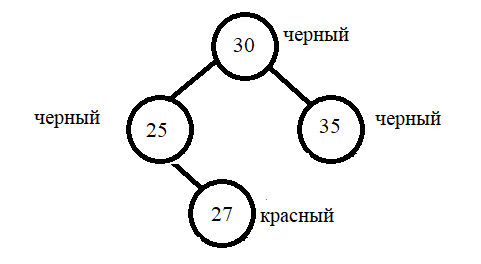


Рис.11

**Получим верхнюю оценку высоты:**

В случае красно-черного дерева будем отталкиваться от черной высоты(hb). Для начала докажем, что количество внутренних узлов не менее 2^(bh-1)-1 (Мат. индукция):

БИ: если у узла высота – 1, то это лист. Подставив это значение в формулу, получим 0.

Так как любой внутренний узел имеет двух потомков, то, учитывая базу, черные высоты потомков будут bh для красного и bh-1 для черного. Также из базы индукции понимаем, что в каждом из поддеревьев не менее 2^(bh-2)-1 узлов. Тогда в поддереве не менее 2(2^(bh-2)-1) + 1 узлов. При сокращении получим: 2^(bh-1)-1, тогда утверждение верно.

Теперь рассмотрим дерево высотой h. Так как у красного узла черные потомки, то количество красных вершин не более h/2 => черных не менее h/2-1.

Исходя из этого: n>=2^(h/2)-1.

log(n+1)>=h/2

2log(n+1)>=h => h=O(logn)

**Практическая часть**

В качестве практического задания были реализованы программы для представления бинарного дерева поиска, AVL-дерева и красно-черного дерева. Программа случайным образом генерирует ключи, составляя из них деревья, пользователь способен искать ключи в деревьях, удалять и добавлять новые.

**Бинарное дерево поиска**

Была написана программа для получения зависимости высоты дерева поиска от количества ключей, учитывая, что их значение - случайная величина, распределенная равномерно.

По завершении работы, мы можем наблюдать следующий график. По экспериментальным точкам была построена регрессия. Как можно заметить, регрессия представляет из себя логарифмическую функцию:

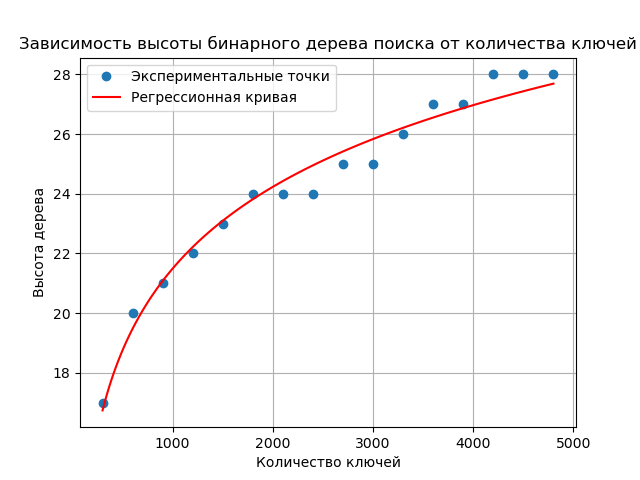


Рис.12

y ≈ 3.9488 \* log(x) + -5.7808

**AVL и красно-черное дерево**

Была написана программа для получения зависимости высоты дерева поиска от количества ключей, учитывая, что их значения монотонно возрастают. Также, как и в случае с бинарным деревом, была построена кривая регрессии с экспериментальными точками и получены полиномы:

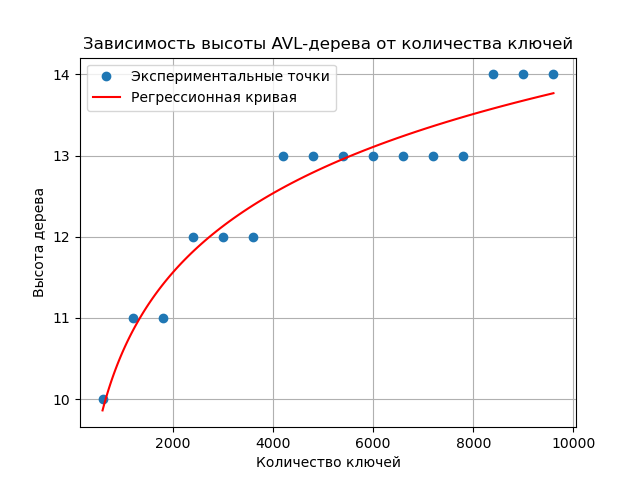


Рис.13

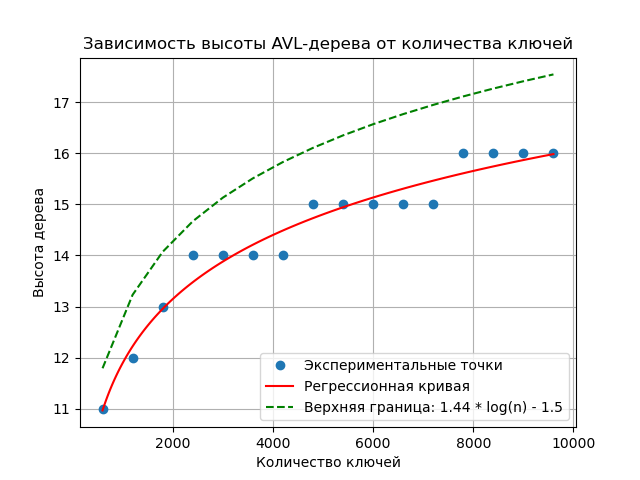


Рис.14

y ≈ 1.4096 \* log(x) + 0.8434

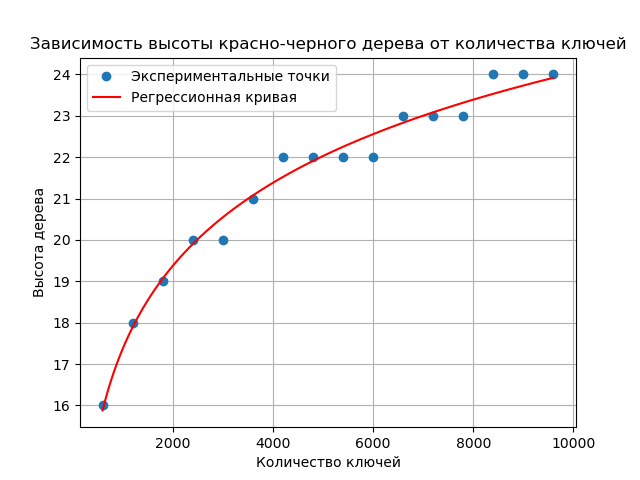


Рис.15

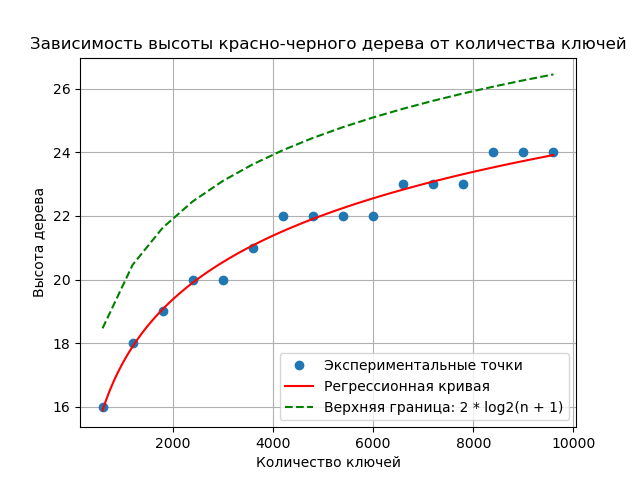


Рис.16

y ≈ 2.8998 \* log(x) + -2.6709

Судя по расположению экспериментальных точек, можно сказать, что алгоритмы AVL и красно-черного деревьев укладываются в верхние границы.

**Обход бинарного дерева** - это процесс посещения всех узлов дерева с целью выполнения определенных операций, например, такой операцией может быть поиск элемента. Существует два основных подхода к обходу бинарного дерева: обход в глубину и обход в ширину.

**Обход в глубину (**Depth-First Traversal)

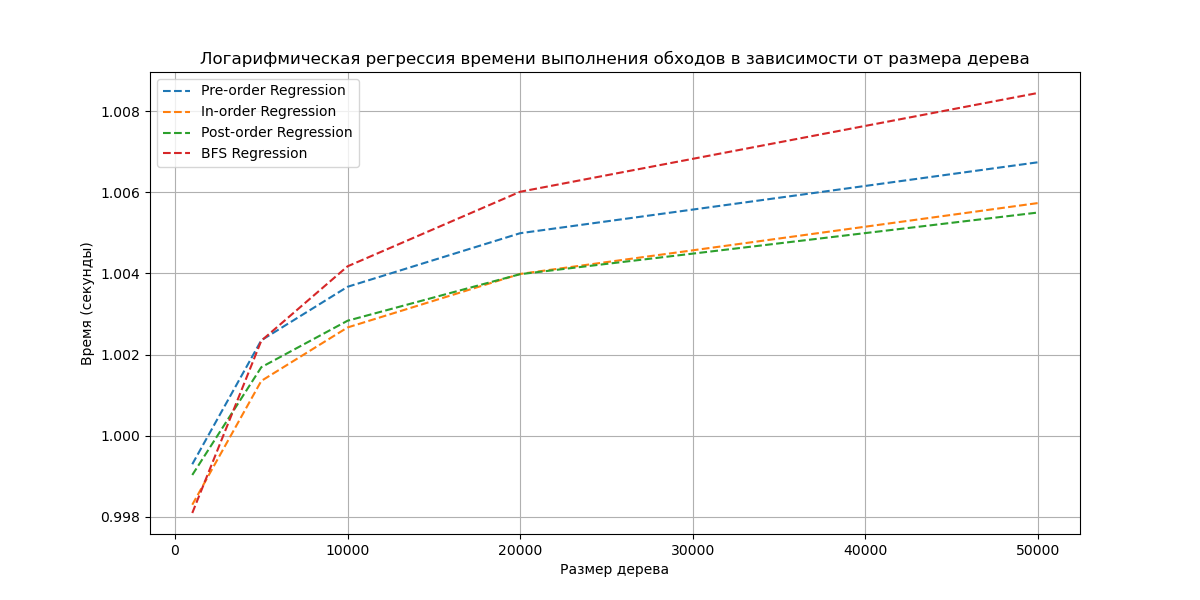
Обход в глубину предполагает, что мы будем углубляться в дерево, начиная с корня, и будем посещать узлы по одной ветви, пока не достигнем листа. Существует три основных способа обхода в глубину:

* Прямой обход (Pre-order): для начала нужно обработать корень, после этого рекурсивно обходим левое поддерево, а после этого обходим правое поддерево.
* Симметричный обход (In-order): отличие данного метода от предыдущего заключается в том, что обработка начинается с рекурсивного обхода левого поддерева. Затем обрабатываем корень дерева. После этого рекурсивно обходим правое поддерево.
* Обратный обход (Post-order): в данном случае сначала рекурсивно обходим левое поддерево, затем правое, а после этого и корень дерева.

**Обход в ширину** **(**Breadth-First Traversal)

Обход в ширину выполняется по уровням. Мы начинаем с корня и последовательно посещаем все узлы на уровне корня, затем переходим к следующему уровню и так далее.

Далее были реализованы обход в глубину и в ширину. Работа была рассмотрена на примере AVL-дерева. В первом случае, были сделаны обходы увеличивающегося бинарного дерева без поиска какого-либо узла. Далее был выполнен поиск случайного элемента в постоянно увеличивающемся дереве, оформлен вывод узлов в порядке обхода.

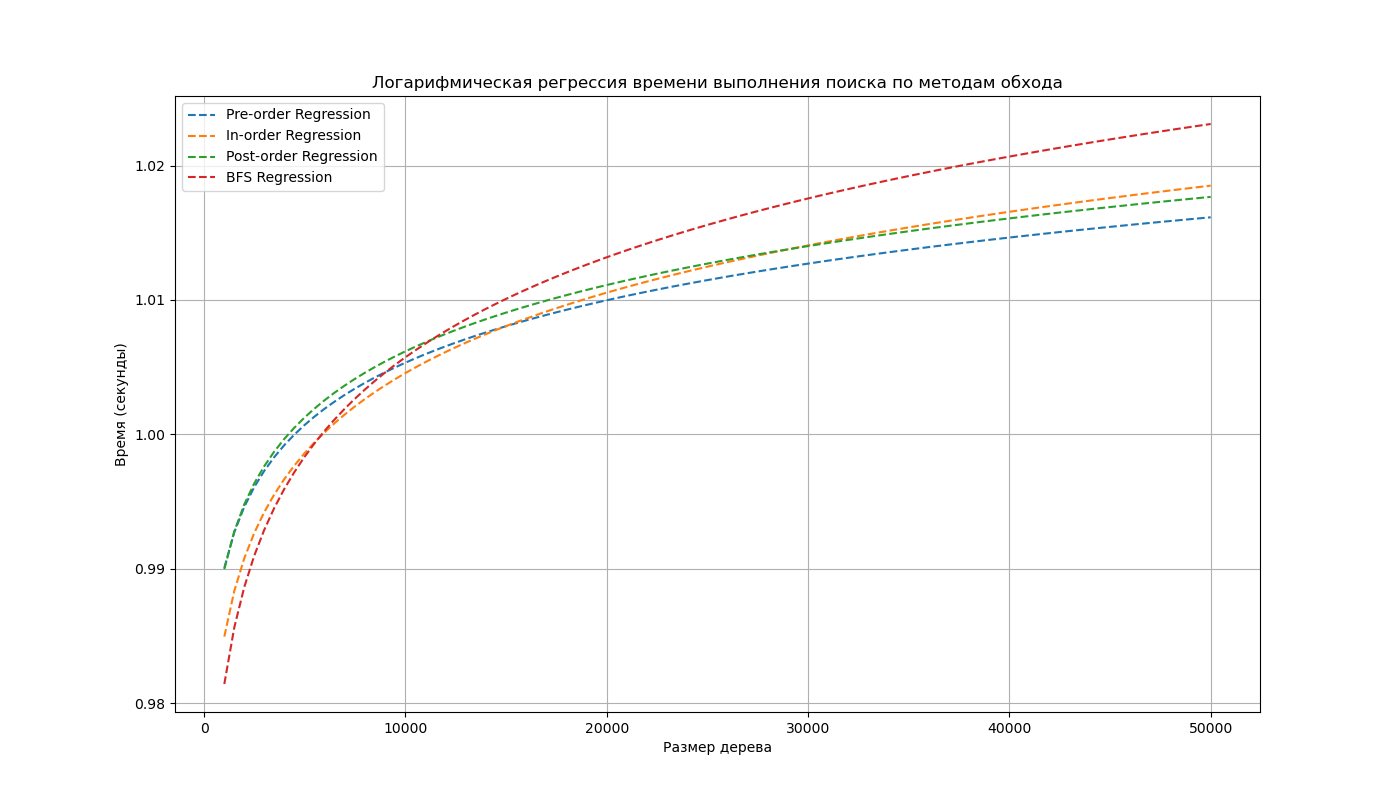
Рис.17

Pre-order Regression: y ≈ 0.001898 \* ln(x) + -0.013818

In-order Regression: y ≈ 0.001898 \* ln(x) + -0.014819

Post-order Regression: y ≈ 0.001651 \* ln(x) + -0.012377

BFS Regression: y ≈ 0.002640 \* ln(x) + -0.020146

Рис.18

Pre-order Regression: y ≈ 0.006648 \* ln(x) + -0.055902

In-order Regression: y ≈ 0.008562 \* ln(x) + -0.074306

Post-order Regression: y ≈ 0.007054 \* ln(x) + -0.058802

BFS Regression: y ≈ 0.010628 \* ln(x) + -0.092159

Как можно заметить, обходы в глубину показывают более совершенный результат в поиске некоторого элемента, но при этом их превосходство ненамного лучше обхода в глубину.

**Пример обхода дерева:**

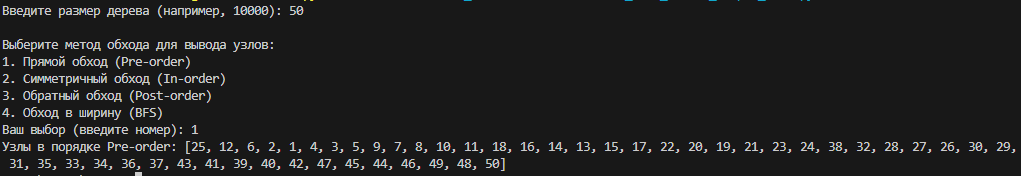


Рис.19

**Вывод**: в ходе выполненной лабораторной работы нами были написаны алгоритмы бинарных самобалансирующихся деревьев поиска: двоичного, AVL, красно-черного деревьев, выведены их теоретические верхние границы высоты, выполнены тесты для получения экспериментальной зависимости, рассмотрены различные способы их обхода. Как показал результат, полученные алгоритмы укладываются в теоретическую оценку, что показывает истинность полученных результатов.

Листинг кода:

import random

class TreeNode:

    def \_\_init\_\_(self, key):

        self.key = key

        self.left = None

        self.right = None

        self.height = 1

class AVLTree:

    @staticmethod

    def get\_height(node):

        return node.height if node else 0

    @staticmethod

    def get\_balance(node):

        return AVLTree.get\_height(node.left) - AVLTree.get\_height(node.right) if node else 0

    @staticmethod

    def right\_rotate(y):

        x = y.left

        T2 = x.right

        x.right = y

        y.left = T2

        y.height = 1 + max(AVLTree.get\_height(y.left), AVLTree.get\_height(y.right))

        x.height = 1 + max(AVLTree.get\_height(x.left), AVLTree.get\_height(x.right))

        return x

    @staticmethod

    def left\_rotate(x):

        y = x.right

        T2 = y.left

        y.left = x

        x.right = T2

        x.height = 1 + max(AVLTree.get\_height(x.left), AVLTree.get\_height(x.right))

        y.height = 1 + max(AVLTree.get\_height(y.left), AVLTree.get\_height(y.right))

        return y

    @staticmethod

    def insert(node, key):

        if not node:

            return TreeNode(key)

        if key < node.key:

            node.left = AVLTree.insert(node.left, key)

        else:

            node.right = AVLTree.insert(node.right, key)

        node.height = 1 + max(AVLTree.get\_height(node.left), AVLTree.get\_height(node.right))

        balance = AVLTree.get\_balance(node)

        if balance > 1 and key < node.left.key:

            return AVLTree.right\_rotate(node)

        if balance < -1 and key > node.right.key:

            return AVLTree.left\_rotate(node)

        if balance > 1 and key > node.left.key:

            node.left = AVLTree.left\_rotate(node.left)

            return AVLTree.right\_rotate(node)

        if balance < -1 and key < node.right.key:

            node.right = AVLTree.right\_rotate(node.right)

            return AVLTree.left\_rotate(node)

        return node

    @staticmethod

    def delete(root, key):

        if not root:

            return root

        if key < root.key:

            root.left = AVLTree.delete(root.left, key)

        elif key > root.key:

            root.right = AVLTree.delete(root.right, key)

        else:

            if not root.left:

                return root.right

            elif not root.right:

                return root.left

            temp = AVLTree.min\_value\_node(root.right)

            root.key = temp.key

            root.right = AVLTree.delete(root.right, temp.key)

        root.height = 1 + max(AVLTree.get\_height(root.left), AVLTree.get\_height(root.right))

        balance = AVLTree.get\_balance(root)

        if balance > 1 and AVLTree.get\_balance(root.left) >= 0:

            return AVLTree.right\_rotate(root)

        if balance < -1 and AVLTree.get\_balance(root.right) <= 0:

            return AVLTree.left\_rotate(root)

        if balance > 1 and AVLTree.get\_balance(root.left) < 0:

            root.left = AVLTree.left\_rotate(root.left)

            return AVLTree.right\_rotate(root)

        if balance < -1 and AVLTree.get\_balance(root.right) > 0:

            root.right = AVLTree.right\_rotate(root.right)

            return AVLTree.left\_rotate(root)

        return root

    @staticmethod

    def min\_value\_node(node):

        current = node

        while current.left is not None:

            current = current.left

        return current

    @staticmethod

    def find(node, key):

        if not node or node.key == key:

            return node

        if key < node.key:

            return AVLTree.find(node.left, key)

        return AVLTree.find(node.right, key)

    @staticmethod

    def print\_tree(node, level=0, prefix="Root: "):

        if node is not None:

            print(" " \* (level \* 4) + prefix + str(node.key))

            if node.right is not None:

                AVLTree.print\_tree(node.right, level + 1, prefix=" |- Правый узел: ")

            if node.left is not None:

                AVLTree.print\_tree(node.left, level + 1, prefix=" |- Левый узел: ")

class AVLTreeApp:

    def \_\_init\_\_(self):

        self.tree = AVLTree()

        self.root = None

    def run(self):

         for value in random.sample(range(1, 301), 100):

             self.root = AVLTree.insert(self.root, value)

         while True:

             action = input("Выберите действие:\n1. Вставить узел\n2. Удалить узел\n3. Найти узел\n4. Вывести дерево\n5. Выход\n")

             if action == "1":

                 key = int(input("Введите значение для вставки: "))

                 self.root = AVLTree.insert(self.root, key)

                 print(f"Вставлено: {key}")

             elif action == "2":

                 key = int(input("Введите значение для удаления: "))

                 self.root = AVLTree.delete(self.root, key)

                 print(f"Удалено: {key}")

             elif action == "3":

                 key = int(input("Введите значение для поиска: "))

                 found\_node = AVLTree.find(self.root, key)

                 message = "Узел найден!" if found\_node else "Узел не найден."

                 print(message)

             elif action == "4":

                 print("Вывод дерева:")

                 self.tree.print\_tree(self.root)

             elif action == "5":

                 break

             else:

                 print("Пожалуйста, введите корректное действие.")

if \_\_name\_\_ == "\_\_main\_\_":

    app = AVLTreeApp()

    app.run()

import random

class TreeNode:

    def \_\_init\_\_(self, key):

        self.key = key

        self.left = None

        self.right = None

class BinarySearchTree:

    def insert(self, node, key):

        if node is None:

            return TreeNode(key)

        if key < node.key:

            node.left = self.insert(node.left, key)

        else:

            node.right = self.insert(node.right, key)

        return node

    def find(self, node, key):

        if node is None or node.key == key:

            return node

        if key < node.key:

            return self.find(node.left, key)

        return self.find(node.right, key)

    def delete(self, root, key):

        if root is None:

            return root

        if key < root.key:

            root.left = self.delete(root.left, key)

        elif key > root.key:

            root.right = self.delete(root.right, key)

        else:

            if root.left is None:

                return root.right

            elif root.right is None:

                return root.left

            temp = self.min\_value\_node(root.right)

            root.key = temp.key

            root.right = self.delete(root.right, temp.key)

        return root

    def min\_value\_node(self, node):

        current = node

        while current.left is not None:

            current = current.left

        return current

    def print\_tree(self, node, level=0, prefix="Root: "):

        if node is not None:

            print(" " \* (level \* 4) + prefix + str(node.key))

            if node.right is not None:

                self.print\_tree(node.right, level + 1, prefix=" |- Правый узел: ")

            if node.left is not None:

                self.print\_tree(node.left, level + 1, prefix=" |- Левый узел: ")

class BinarySearchTreeApp:

    def \_\_init\_\_(self):

        self.tree = BinarySearchTree()

        self.root = None

    def run(self):

         for value in random.sample(range(1, 301), 100):

             self.root = self.tree.insert(self.root, value)

         while True:

             action = input("Выберите действие:\n1. Вставить узел\n2. Удалить узел\n3. Найти узел\n4. Вывести дерево\n5. Выход\n")

             if action == "1":

                 key = int(input("Введите значение для вставки: "))

                 self.root = self.tree.insert(self.root, key)

                 print(f"Вставлено: {key}")

             elif action == "2":

                 key = int(input("Введите значение для удаления: "))

                 self.root = self.tree.delete(self.root, key)

                 print(f"Удалено: {key}")

             elif action == "3":

                 key = int(input("Введите значение для поиска: "))

                 found\_node = self.tree.find(self.root, key)

                 message = "Узел найден!" if found\_node else "Узел не найден."

                 print(message)

             elif action == "4":

                 print("Вывод дерева:")

                 self.tree.print\_tree(self.root)

             elif action == "5":

                 break

             else:

                 print("Пожалуйста, введите корректное действие.")

if \_\_name\_\_ == "\_\_main\_\_":

    app = BinarySearchTreeApp()

    app.run()

    app.run()

import random

class TreeNode:

    def \_\_init\_\_(self, key):

        self.key = key

        self.color = 'red'

        self.left = None

        self.right = None

        self.parent = None

class RedBlackTree:

    def \_\_init\_\_(self):

        self.NIL\_LEAF = TreeNode(None)

        self.NIL\_LEAF.color = 'black'

        self.root = self.NIL\_LEAF

    def insert(self, key):

        new\_node = TreeNode(key)

        new\_node.left = self.NIL\_LEAF

        new\_node.right = self.NIL\_LEAF

        parent = None

        current = self.root

        while current != self.NIL\_LEAF:

            parent = current

            if new\_node.key < current.key:

                current = current.left

            else:

                current = current.right

        new\_node.parent = parent

        if parent is None:

            self.root = new\_node

        elif new\_node.key < parent.key:

            parent.left = new\_node

        else:

            parent.right = new\_node

        new\_node.color = 'red'

        self.fix\_insert(new\_node)

    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':

                    node.parent.color = 'black'

                    uncle.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':

                    node.parent.color = 'black'

                    uncle.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 left\_rotate(self, x):

        y = x.right

        x.right = y.left

        if y.left != self.NIL\_LEAF:

            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, y):

        x = y.left

        y.left = x.right

        if x.right != self.NIL\_LEAF:

            x.right.parent = y

        x.parent = y.parent

        if y.parent is None:

            self.root = x

        elif y == y.parent.right:

            y.parent.right = x

        else:

            y.parent.left = x

        x.right = y

        y.parent = x

    def find(self, key):

        return self.\_find(self.root, key)

    def \_find(self, node, key):

        if node == self.NIL\_LEAF or key == node.key:

            return node

        if key < node.key:

            return self.\_find(node.left, key)

        return self.\_find(node.right, key)

    def delete(self, key):

        z = self.find(key)

        if z == self.NIL\_LEAF:

            print(f"Узел с ключом {key} не найден.")

            return

        original\_color\_y= z.color

        if z.left == self.NIL\_LEAF:

            x= z.right

            self.transplant(z, z.right)

        elif z.right == self.NIL\_LEAF:

            x= z.left

            self.transplant(z, z.left)

        else:

            y= self.min\_value\_node(z.right)

            original\_color\_y= y.color

            x= y.right

            if y.parent == z:

                x.parent= y

            else:

                self.transplant(y, y.right)

                y.right= z.right

                y.right.parent= y

            self.transplant(z, y)

            y.left= z.left

            y.left.parent= y

            y.color= z.color

        if original\_color\_y == 'black':

            self.fix\_delete(x)

    def transplant(self, u, v):

        if u.parent is None:

            self.root = v

        elif u == u.parent.left:

            u.parent.left= v

        else:

            u.parent.right= v

        v.parent= u.parent

    def fix\_delete(self, x):

       while x !=self.root and x.color =='black':

           if x ==x.parent.left:

               w=x.parent.right

               if w.color =='red':

                   w.color ='black'

                   x.parent.color ='red'

                   self.left\_rotate(x.parent)

                   w=x.parent.right

               if w.left.color=='black' and w.right.color=='black':

                   w.color ='red'

                   x=x.parent

               else:

                   if w.right.color=='black':

                       w.left.color='black'

                       w.color='red'

                       self.right\_rotate(w)

                       w=x.parent.right

                   w.color=x.parent.color

                   x.parent.color ='black'

                   w.right.color ='black'

                   self.left\_rotate(x.parent)

                   x=self.root

           else:

               w=x.parent.left

               if w.color =='red':

                   w.color ='black'

                   x.parent.color ='red'

                   self.right\_rotate(x.parent)

                   w=x.parent.left

               if w.right.color=='black' and w.left.color=='black':

                   w.color ='red'

                   x=x.parent

               else:

                   if w.left.color=='black':

                       w.right.color='black'

                       w.color='red'

                       self.left\_rotate(w)

                       w=x.parent.left

                   w.color=x.parent.color

                   x.parent.color ='black'

                   w.left.color ='black'

                   self.right\_rotate(x.parent)

                   x=self.root

       x.color='black'

    def min\_value\_node(self, node):

       current=node

       while current.left!=self.NIL\_LEAF:

           current=current.left

       return current

    def print\_tree(self, node, level=0, prefix="Root: "):

       if node !=self.NIL\_LEAF:

           print(" " \* (level \* 4) + prefix + str(node.key) + f" ({node.color})")

           if node.right !=self.NIL\_LEAF:

               self.print\_tree(node.right, level +1 , prefix=" |- Правый узел: ")

           if node.left !=self.NIL\_LEAF:

               self.print\_tree(node.left, level +1 , prefix=" |- Левый узел: ")

class RedBlackTreeApp:

    def \_\_init\_\_(self):

       self.tree=RedBlackTree()

       for value in random.sample(range(1, 301), 300):

           self.tree.insert(value)

    def run(self):

       while True:

           action=input("Выберите действие:\n1. Вставить узел\n2. Удалить узел\n3. Найти узел\n4. Вывести дерево\n5. Выход\n")

           if action=="1":

               key=int(input("Введите значение для вставки: "))

               print(f"Вставлено: {key}")

               self.tree.insert(key)

           elif action=="2":

               key=int(input("Введите значение для удаления: "))

               print(f"Удалено: {key}")

               self.tree.delete(key)

           elif action=="3":

               key=int(input("Введите значение для поиска: "))

               found\_node=self.tree.find(key)

               message="Узел найден!"if found\_node!=self.tree.NIL\_LEAF else"Узел не найден."

               print(message)

           elif action=="4":

               print("Вывод дерева:")

               self.tree.print\_tree(self.tree.root)

           elif action=="5":

               break

           else:

               print("Пожалуйста, введите корректное действие.")

if \_\_name\_\_=="\_\_main\_\_":

   app=RedBlackTreeApp()

   app.run()

import matplotlib.pyplot as plt

import numpy as np

import random

class TreeNode:

    def \_\_init\_\_(self, key):

        self.key = key

        self.left = None

        self.right = None

        self.height = 1

class AVLTree:

    @staticmethod

    def get\_height(node):

        return node.height if node else 0

    @staticmethod

    def get\_balance(node):

        return AVLTree.get\_height(node.left) - AVLTree.get\_height(node.right) if node else 0

    @staticmethod

    def right\_rotate(y):

        x = y.left

        T2 = x.right

        x.right = y

        y.left = T2

        y.height = 1 + max(AVLTree.get\_height(y.left), AVLTree.get\_height(y.right))

        x.height = 1 + max(AVLTree.get\_height(x.left), AVLTree.get\_height(x.right))

        return x

    @staticmethod

    def left\_rotate(x):

        y = x.right

        T2 = y.left

        y.left = x

        x.right = T2

        x.height = 1 + max(AVLTree.get\_height(x.left), AVLTree.get\_height(x.right))

        y.height = 1 + max(AVLTree.get\_height(y.left), AVLTree.get\_height(y.right))

        return y

    @staticmethod

    def insert(node, key):

        if not node:

            return TreeNode(key)

        if key < node.key:

            node.left = AVLTree.insert(node.left, key)

        else:

            node.right = AVLTree.insert(node.right, key)

        node.height = 1 + max(AVLTree.get\_height(node.left), AVLTree.get\_height(node.right))

        balance = AVLTree.get\_balance(node)

        if balance > 1 and key < node.left.key:

            return AVLTree.right\_rotate(node)

        if balance < -1 and key > node.right.key:

            return AVLTree.left\_rotate(node)

        if balance > 1 and key > node.left.key:

            node.left = AVLTree.left\_rotate(node.left)

            return AVLTree.right\_rotate(node)

        if balance < -1 and key < node.right.key:

            node.right = AVLTree.right\_rotate(node.right)

            return AVLTree.left\_rotate(node)

        return node

def main():

    avl\_tree = AVLTree()

    root = None

    heights = []

    num\_keys = []

    keys = random.sample(range(1, 10001), 10000)

    for i in range(len(keys)):

        root = AVLTree.insert(root, keys[i])

        if (i + 1) % 600 == 0:

            heights.append(AVLTree.get\_height(root))

            num\_keys.append(i + 1)

    plt.plot(num\_keys, heights, 'o', label='Экспериментальные точки')

    log\_num\_keys = np.log(num\_keys)

    coefficients = np.polyfit(log\_num\_keys, heights, 1)

    polynomial = np.poly1d(coefficients)

    x\_fit = np.linspace(min(log\_num\_keys), max(log\_num\_keys), 100)

    y\_fit = polynomial(x\_fit)

    plt.plot(np.exp(x\_fit), y\_fit, label='Регрессионная кривая', color='red')

    upper\_bound\_heights = [1.44 \* np.log2(n) - 1.5 for n in num\_keys]

    plt.plot(num\_keys, upper\_bound\_heights, label='Верхняя граница: 1.44 \* log(n) - 1.5', color='green', linestyle='--')

    plt.title("Зависимость высоты AVL-дерева от количества ключей")

    plt.xlabel("Количество ключей")

    plt.ylabel("Высота дерева")

    plt.legend()

    plt.grid()

    plt.show()

if \_\_name\_\_ == "\_\_main\_\_":

    main()

import matplotlib.pyplot as plt

import numpy as np

class TreeNode:

    def \_\_init\_\_(self, key):

        self.key = key

        self.color = 'red'

        self.left = None

        self.right = None

        self.parent = None

class RedBlackTree:

    def \_\_init\_\_(self):

        self.NIL\_LEAF = TreeNode(None)

        self.NIL\_LEAF.color = 'black'

        self.root = self.NIL\_LEAF

    def insert(self, key):

        new\_node = TreeNode(key)

        new\_node.left = self.NIL\_LEAF

        new\_node.right = self.NIL\_LEAF

        parent = None

        current = self.root

        while current != self.NIL\_LEAF:

            parent = current

            if new\_node.key < current.key:

                current = current.left

            else:

                current = current.right

        new\_node.parent = parent

        if parent is None:

            self.root = new\_node

        elif new\_node.key < parent.key:

            parent.left = new\_node

        else:

            parent.right = new\_node

        new\_node.color = 'red'

        self.fix\_insert(new\_node)

    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':

                    node.parent.color = 'black'

                    uncle.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':

                    node.parent.color = 'black'

                    uncle.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 left\_rotate(self, x):

        y = x.right

        x.right = y.left

        if y.left != self.NIL\_LEAF:

            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, y):

        x = y.left

        y.left = x.right

        if x.right != self.NIL\_LEAF:

            x.right.parent = y

        x.parent = y.parent

        if y.parent is None:

            self.root = x

        elif y == y.parent.right:

            y.parent.right = x

        else:

            y.parent.left = x

        x.right = y

        y.parent = x

    def height(self, node):

        """Возвращает высоту дерева."""

        if node == self.NIL\_LEAF:

            return 0

        return 1 + max(self.height(node.left), self.height(node.right))

class RedBlackTreeApp:

    def \_\_init\_\_(self):

        self.tree=RedBlackTree()

    def run(self):

         heights = []

         num\_keys = []

         for i in range(1, 10001):

             key = i

             self.tree.insert(key)

             if i % 600 == 0:

                 heights.append(self.tree.height(self.tree.root))

                 num\_keys.append(i)

         plt.plot(num\_keys, heights, 'o', label='Экспериментальные точки')

         log\_num\_keys = np.log(num\_keys)

         coefficients = np.polyfit(log\_num\_keys, heights, 1)

         polynomial\_func = np.poly1d(coefficients)

         x\_fit = np.linspace(min(log\_num\_keys), max(log\_num\_keys), 100)

         y\_fit = polynomial\_func(x\_fit)

         plt.plot(np.exp(x\_fit), y\_fit, label='Регрессионная кривая', color='red')

         upper\_bound\_heights = [2 \* np.log2(n + 1) for n in num\_keys]

         plt.plot(num\_keys, upper\_bound\_heights, label='Верхняя граница: 2 \* log2(n + 1)', color='green', linestyle='--')

         plt.title("Зависимость высоты красно-черного дерева от количества ключей")

         plt.xlabel("Количество ключей")

         plt.ylabel("Высота дерева")

         plt.legend()

         plt.grid()

         plt.show()

         print("Уравнение регрессионной кривой:")

         print(f"y ≈ {coefficients[0]:.4f} \* log(x) + {coefficients[1]:.4f}")

if \_\_name\_\_ == "\_\_main\_\_":

    app=RedBlackTreeApp()

    app.run()

import random

import matplotlib.pyplot as plt

import numpy as np

class TreeNode:

    def \_\_init\_\_(self, key):

        self.key = key

        self.left = None

        self.right = None

class BinarySearchTree:

    def insert(self, node, key):

        """Вставка нового узла с указанным ключом."""

        if node is None:

            return TreeNode(key)

        if key < node.key:

            node.left = self.insert(node.left, key)

        else:

            node.right = self.insert(node.right, key)

        return node

    def find(self, node, key):

        """Поиск узла по ключу."""

        if node is None or node.key == key:

            return node

        if key < node.key:

            return self.find(node.left, key)

        return self.find(node.right)

    def delete(self, root, key):

        """Удаление узла по ключу."""

        if root is None:

            return root

        if key < root.key:

            root.left = self.delete(root.left, key)

        elif key > root.key:

            root.right = self.delete(root.right, key)

        else:

            if root.left is None:

                return root.right

            elif root.right is None:

                return root.left

            temp = self.min\_value\_node(root.right)

            root.key = temp.key

            root.right = self.delete(root.right, temp.key)

        return root

    def min\_value\_node(self, node):

        """Получение узла с минимальным ключом."""

        current = node

        while current.left is not None:

            current = current.left

        return current

    def height(self, node):

        """Вычисление высоты дерева."""

        if node is None:

            return 0

        else:

            left\_height = self.height(node.left)

            right\_height = self.height(node.right)

            return max(left\_height, right\_height) + 1

class BinarySearchTreeApp:

    def \_\_init\_\_(self):

        self.tree = BinarySearchTree()

        self.root = None

    def run(self):

         heights = []

         num\_keys = []

         for i in range(1, 5001):

             key = random.randint(1, 10000)

             self.root = self.tree.insert(self.root, key)

             if i % 300 == 0:

                 heights.append(self.tree.height(self.root))

                 num\_keys.append(i)

         plt.plot(num\_keys, heights, 'o', label='Экспериментальные точки')

         log\_num\_keys = np.log(num\_keys)

         coefficients = np.polyfit(log\_num\_keys, heights, 1)

         polynomial = np.poly1d(coefficients)

         x\_fit = np.linspace(min(log\_num\_keys), max(log\_num\_keys), 100)

         y\_fit = polynomial(x\_fit)

         plt.plot(np.exp(x\_fit), y\_fit, label='Регрессионная кривая', color='red')

         plt.title("Зависимость высоты бинарного дерева поиска от количества ключей")

         plt.xlabel("Количество ключей")

         plt.ylabel("Высота дерева")

         plt.legend()

         plt.grid()

         plt.show()

         print("Уравнение регрессионной кривой:")

         print(f"y ≈ {coefficients[0]:.4f} \* log(x) + {coefficients[1]:.4f}")

if \_\_name\_\_ == "\_\_main\_\_":

    app = BinarySearchTreeApp()

    app.run()

import random

import time

from collections import deque

import matplotlib.pyplot as plt

import numpy as np

class TreeNode:

    def \_\_init\_\_(self, key):

        self.key = key

        self.left = None

        self.right = None

        self.height = 1

class AVLTree:

    @staticmethod

    def get\_height(node):

        if not node:

            return 0

        return node.height

    @staticmethod

    def get\_balance(node):

        if not node:

            return 0

        return AVLTree.get\_height(node.left) - AVLTree.get\_height(node.right)

    @staticmethod

    def right\_rotate(y):

        x = y.left

        T2 = x.right

        x.right = y

        y.left = T2

        y.height = 1 + max(AVLTree.get\_height(y.left), AVLTree.get\_height(y.right))

        x.height = 1 + max(AVLTree.get\_height(x.left), AVLTree.get\_height(x.right))

        return x

    @staticmethod

    def left\_rotate(x):

        y = x.right

        T2 = y.left

        y.left = x

        x.right = T2

        x.height = 1 + max(AVLTree.get\_height(x.left), AVLTree.get\_height(x.right))

        y.height = 1 + max(AVLTree.get\_height(y.left), AVLTree.get\_height(y.right))

        return y

    @staticmethod

    def insert(node, key):

        if not node:

            return TreeNode(key)

        if key < node.key:

            node.left = AVLTree.insert(node.left, key)

        else:

            node.right = AVLTree.insert(node.right, key)

        node.height = 1 + max(AVLTree.get\_height(node.left), AVLTree.get\_height(node.right))

        balance = AVLTree.get\_balance(node)

        if balance > 1 and key < node.left.key:

            return AVLTree.right\_rotate(node)

        if balance < -1 and key > node.right.key:

            return AVLTree.left\_rotate(node)

        if balance > 1 and key > node.left.key:

            node.left = AVLTree.left\_rotate(node.left)

            return AVLTree.right\_rotate(node)

        if balance < -1 and key < node.right.key:

            node.right = AVLTree.right\_rotate(node.right)

            return AVLTree.left\_rotate(node)

        return node

    @staticmethod

    def dfs\_pre\_order(node):

         result = []

         def traverse(n):

             if n:

                 result.append(n.key)

                 traverse(n.left)

                 traverse(n.right)

         traverse(node)

         return result

    @staticmethod

    def dfs\_in\_order(node):

         result = []

         def traverse(n):

             if n:

                 traverse(n.left)

                 result.append(n.key)

                 traverse(n.right)

         traverse(node)

         return result

    @staticmethod

    def dfs\_post\_order(node):

         result = []

         def traverse(n):

             if n:

                 traverse(n.left)

                 traverse(n.right)

                 result.append(n.key)

         traverse(node)

         return result

    @staticmethod

    def bfs\_traversal(root):

         result = []

         if not root:

             return result

         queue = deque([root])

         while queue:

             current = queue.popleft()

             result.append(current.key)

             if current.left:

                 queue.append(current.left)

             if current.right:

                 queue.append(current.right)

         return result

class AVLTreeApp:

     def \_\_init\_\_(self):

          self.root = None

     def measure\_traversal\_time(self, size):

          values\_to\_insert = random.sample(range(1, size + 1), size)

          self.root = None

          for value in values\_to\_insert:

              self.root = AVLTree.insert(self.root, value)

          times = {}

          start\_time\_pre\_order = time.time()

          AVLTree.dfs\_pre\_order(self.root)

          times['Pre-order'] = time.time() - start\_time\_pre\_order

          start\_time\_in\_order = time.time()

          AVLTree.dfs\_in\_order(self.root)

          times['In-order'] = time.time() - start\_time\_in\_order

          start\_time\_post\_order = time.time()

          AVLTree.dfs\_post\_order(self.root)

          times['Post-order'] = time.time() - start\_time\_post\_order

          start\_time\_bfs = time.time()

          AVLTree.bfs\_traversal(self.root)

          times['BFS'] = time.time() - start\_time\_bfs

          return times

     def run(self):

          sizes = [1000, 5000, 10000, 20000, 50000]

          results = {size: self.measure\_traversal\_time(size) for size in sizes}

          pre\_order\_times = [results[size]['Pre-order'] for size in sizes]

          in\_order\_times = [results[size]['In-order'] for size in sizes]

          post\_order\_times = [results[size]['Post-order'] for size in sizes]

          bfs\_times = [results[size]['BFS'] for size in sizes]

          plt.figure(figsize=(12, 6))

          for times, label in zip([pre\_order\_times, in\_order\_times, post\_order\_times, bfs\_times],

                                   ['Pre-order', 'In-order', 'Post-order', 'BFS']):

              log\_sizes = np.log(sizes)

              coeffs = np.polyfit(log\_sizes, times, deg=1)

              regression\_line = np.exp(coeffs[0] \* log\_sizes + coeffs[1])

              plt.plot(sizes, regression\_line, linestyle='--', label=f'{label} Regression')

              print(f"{label} Regression: y ≈ {coeffs[0]:.6f} \* ln(x) + {coeffs[1]:.6f}")

          plt.title('Логарифмическая регрессия времени выполнения обходов в зависимости от размера дерева')

          plt.xlabel('Размер дерева')

          plt.ylabel('Время (секунды)')

          plt.legend()

          plt.grid(True)

          plt.savefig('tree\_traversal\_regression.png')

          plt.show()

if \_\_name\_\_ == "\_\_main\_\_":

     app = AVLTreeApp()

     app.run()

import random

import time

from collections import deque

import matplotlib.pyplot as plt

import numpy as np

class TreeNode:

    def \_\_init\_\_(self, key):

        self.key = key

        self.left = None

        self.right = None

        self.height = 1

class AVLTree:

    @staticmethod

    def get\_height(node):

        return node.height if node else 0

    @staticmethod

    def get\_balance(node):

        return AVLTree.get\_height(node.left) - AVLTree.get\_height(node.right) if node else 0

    @staticmethod

    def right\_rotate(y):

        x = y.left

        T2 = x.right

        x.right = y

        y.left = T2

        y.height = 1 + max(AVLTree.get\_height(y.left), AVLTree.get\_height(y.right))

        x.height = 1 + max(AVLTree.get\_height(x.left), AVLTree.get\_height(x.right))

        return x

    @staticmethod

    def left\_rotate(x):

        y = x.right

        T2 = y.left

        y.left = x

        x.right = T2

        x.height = 1 + max(AVLTree.get\_height(x.left), AVLTree.get\_height(x.right))

        y.height = 1 + max(AVLTree.get\_height(y.left), AVLTree.get\_height(y.right))

        return y

    @staticmethod

    def insert(node, key):

        if not node:

            return TreeNode(key)

        if key < node.key:

            node.left = AVLTree.insert(node.left, key)

        else:

            node.right = AVLTree.insert(node.right, key)

        node.height = 1 + max(AVLTree.get\_height(node.left), AVLTree.get\_height(node.right))

        balance = AVLTree.get\_balance(node)

        if balance > 1 and key < node.left.key:

            return AVLTree.right\_rotate(node)

        if balance < -1 and key > node.right.key:

            return AVLTree.left\_rotate(node)

        if balance > 1 and key > node.left.key:

            node.left = AVLTree.left\_rotate(node.left)

            return AVLTree.right\_rotate(node)

        if balance < -1 and key < node.right.key:

            node.right = AVLTree.right\_rotate(node.right)

            return AVLTree.left\_rotate(node)

        return node

    @staticmethod

    def dfs\_pre\_order(node):

         result = []

         def traverse(n):

             if n:

                 result.append(n.key)

                 traverse(n.left)

                 traverse(n.right)

         traverse(node)

         return result

    @staticmethod

    def dfs\_in\_order(node):

         result = []

         def traverse(n):

             if n:

                 traverse(n.left)

                 result.append(n.key)

                 traverse(n.right)

         traverse(node)

         return result

    @staticmethod

    def dfs\_post\_order(node):

         result = []

         def traverse(n):

             if n:

                 traverse(n.left)

                 traverse(n.right)

                 result.append(n.key)

         traverse(node)

         return result

    @staticmethod

    def bfs\_traversal(root):

         result = []

         if not root:

             return result

         queue = deque([root])

         while queue:

             current = queue.popleft()

             result.append(current.key)

             if current.left:

                 queue.append(current.left)

             if current.right:

                 queue.append(current.right)

         return result

class AVLTreeApp:

     def \_\_init\_\_(self):

          self.root = None

     def measure\_traversal\_time(self, size):

          values\_to\_insert = random.sample(range(1, size + 1), size)

          self.root = None

          for value in values\_to\_insert:

              self.root = AVLTree.insert(self.root, value)

          times = {}

          start\_time\_pre\_order = time.time()

          pre\_order\_result = AVLTree.dfs\_pre\_order(self.root)

          times['Pre-order'] = time.time() - start\_time\_pre\_order

          start\_time\_in\_order = time.time()

          in\_order\_result = AVLTree.dfs\_in\_order(self.root)

          times['In-order'] = time.time() - start\_time\_in\_order

          start\_time\_post\_order = time.time()

          post\_order\_result = AVLTree.dfs\_post\_order(self.root)

          times['Post-order'] = time.time() - start\_time\_post\_order

          start\_time\_bfs = time.time()

          bfs\_result = AVLTree.bfs\_traversal(self.root)

          times['BFS'] = time.time() - start\_time\_bfs

          return times, pre\_order\_result, in\_order\_result, post\_order\_result, bfs\_result

     def run(self):

          size = int(input("Введите размер дерева (например, 10000): "))

          search\_times, pre\_order\_result, in\_order\_result, post\_order\_result, bfs\_result = self.measure\_traversal\_time(size)

          print("\nВыберите метод обхода для вывода узлов:")

          print("1. Прямой обход (Pre-order)")

          print("2. Симметричный обход (In-order)")

          print("3. Обратный обход (Post-order)")

          print("4. Обход в ширину (BFS)")

          choice = input("Ваш выбор (введите номер): ")

          if choice == "1":

              print("Узлы в порядке Pre-order:", pre\_order\_result)

          elif choice == "2":

              print("Узлы в порядке In-order:", in\_order\_result)

          elif choice == "3":

              print("Узлы в порядке Post-order:", post\_order\_result)

          elif choice == "4":

              print("Узлы в порядке BFS:", bfs\_result)

          else:

              print("Неверный выбор.")

if \_\_name\_\_ == "\_\_main\_\_":

     app = AVLTreeApp()

     app.run()

import random

import time

from collections import deque

import matplotlib.pyplot as plt

import numpy as np

class TreeNode:

    def \_\_init\_\_(self, key):

        self.key = key

        self.left = None

        self.right = None

        self.height = 1

class AVLTree:

    @staticmethod

    def get\_height(node):

        if not node:

            return 0

        return node.height

    @staticmethod

    def get\_balance(node):

        if not node:

            return 0

        return AVLTree.get\_height(node.left) - AVLTree.get\_height(node.right)

    @staticmethod

    def right\_rotate(y):

        x = y.left

        T2 = x.right

        x.right = y

        y.left = T2

        y.height = 1 + max(AVLTree.get\_height(y.left), AVLTree.get\_height(y.right))

        x.height = 1 + max(AVLTree.get\_height(x.left), AVLTree.get\_height(x.right))

        return x

    @staticmethod

    def left\_rotate(x):

        y = x.right

        T2 = y.left

        y.left = x

        x.right = T2

        x.height = 1 + max(AVLTree.get\_height(x.left), AVLTree.get\_height(x.right))

        y.height = 1 + max(AVLTree.get\_height(y.left), AVLTree.get\_height(y.right))

        return y

    @staticmethod

    def insert(node, key):

        if not node:

            return TreeNode(key)

        if key < node.key:

            node.left = AVLTree.insert(node.left, key)

        else:

            node.right = AVLTree.insert(node.right, key)

        node.height = 1 + max(AVLTree.get\_height(node.left), AVLTree.get\_height(node.right))

        balance = AVLTree.get\_balance(node)

        if balance > 1 and key < node.left.key:

            return AVLTree.right\_rotate(node)

        if balance < -1 and key > node.right.key:

            return AVLTree.left\_rotate(node)

        if balance > 1 and key > node.left.key:

            node.left = AVLTree.left\_rotate(node.left)

            return AVLTree.right\_rotate(node)

        if balance < -1 and key < node.right.key:

            node.right = AVLTree.right\_rotate(node.right)

            return AVLTree.left\_rotate(node)

        return node

    @staticmethod

    def dfs\_pre\_order(node):

         result = []

         def traverse(n):

             if n:

                 result.append(n.key)

                 traverse(n.left)

                 traverse(n.right)

         traverse(node)

         return result

    @staticmethod

    def dfs\_in\_order(node):

         result = []

         def traverse(n):

             if n:

                 traverse(n.left)

                 result.append(n.key)

                 traverse(n.right)

         traverse(node)

         return result

    @staticmethod

    def dfs\_post\_order(node):

         result = []

         def traverse(n):

             if n:

                 traverse(n.left)

                 traverse(n.right)

                 result.append(n.key)

         traverse(node)

         return result

    @staticmethod

    def bfs\_traversal(root):

         result = []

         if not root:

             return result

         queue = deque([root])

         while queue:

             current = queue.popleft()

             result.append(current.key)

             if current.left:

                 queue.append(current.left)

             if current.right:

                 queue.append(current.right)

         return result

class AVLTreeApp:

     def \_\_init\_\_(self):

          self.root = None

     def measure\_search\_time(self, size):

          values\_to\_insert = random.sample(range(1, size + 1), size)

          self.root = None

          for value in values\_to\_insert:

              self.root = AVLTree.insert(self.root, value)

          search\_key = random.choice(values\_to\_insert)

          times = {}

          start\_time\_pre\_order = time.time()

          found\_node\_pre\_order = AVLTree.dfs\_pre\_order(self.root)

          times['Pre-order'] = time.time() - start\_time\_pre\_order

          start\_time\_in\_order = time.time()

          found\_node\_in\_order = AVLTree.dfs\_in\_order(self.root)

          times['In-order'] = time.time() - start\_time\_in\_order

          start\_time\_post\_order = time.time()

          found\_node\_post\_order = AVLTree.dfs\_post\_order(self.root)

          times['Post-order'] = time.time() - start\_time\_post\_order

          start\_time\_bfs = time.time()

          found\_node\_bfs = AVLTree.bfs\_traversal(self.root)

          times['BFS'] = time.time() - start\_time\_bfs

          return times

     def run(self):

          sizes = [1000, 5000, 10000, 20000, 50000]

          results\_preorder, results\_inorder, results\_postorder, results\_bfs= [], [], [], []

          for size in sizes:

              search\_times = self.measure\_search\_time(size)

              results\_preorder.append(search\_times['Pre-order'])

              results\_inorder.append(search\_times['In-order'])

              results\_postorder.append(search\_times['Post-order'])

              results\_bfs.append(search\_times['BFS'])

          plt.figure(figsize=(14, 8))

          for times, label in zip(

              [results\_preorder, results\_inorder, results\_postorder, results\_bfs],

              ['Pre-order', 'In-order', 'Post-order', 'BFS']):

              log\_sizes = np.log(sizes)

              coeffs = np.polyfit(log\_sizes[1:], times[1:], deg=1)

              regression\_line\_x\_values = np.linspace(min(sizes), max(sizes), num=100)

              regression\_line\_y\_values = np.exp(coeffs[0] \* np.log(regression\_line\_x\_values) + coeffs[1])

              plt.plot(regression\_line\_x\_values, regression\_line\_y\_values, linestyle='--', label=f'{label} Regression')

              print(f"{label} Regression: y ≈ {coeffs[0]:.6f} \* ln(x) + {coeffs[1]:.6f}")

          plt.title('Логарифмическая регрессия времени выполнения поиска по методам обхода')

          plt.xlabel('Размер дерева')

          plt.ylabel('Время (секунды)')

          plt.legend()

          plt.grid(True)

          plt.savefig('search\_time\_regression.png')

          plt.show()

if \_\_name\_\_ == "\_\_main\_\_":

     app = AVLTreeApp()

     app.run()

import random

import time

from collections import deque

import matplotlib.pyplot as plt

import numpy as np

class TreeNode:

    def \_\_init\_\_(self, key):

        self.key = key

        self.left = None

        self.right = None

class BinaryTree:

    def \_\_init\_\_(self):

        self.root = None

    def insert(self, key):

        if not self.root:

            self.root = TreeNode(key)

        else:

            self.\_insert\_recursively(self.root, key)

    def \_insert\_recursively(self, node, key):

        if key < node.key:

            if node.left is None:

                node.left = TreeNode(key)

            else:

                self.\_insert\_recursively(node.left, key)

        else:

            if node.right is None:

                node.right = TreeNode(key)

            else:

                self.\_insert\_recursively(node.right, key)

    def dfs\_pre\_order(self, node):

         result = []

         def traverse(n):

             if n:

                 result.append(n.key)

                 traverse(n.left)

                 traverse(n.right)

         traverse(node)

         return result

    def dfs\_in\_order(self, node):

         result = []

         def traverse(n):

             if n:

                 traverse(n.left)

                 result.append(n.key)

                 traverse(n.right)

         traverse(node)

         return result

    def dfs\_post\_order(self, node):

         result = []

         def traverse(n):

             if n:

                 traverse(n.left)

                 traverse(n.right)

                 result.append(n.key)

         traverse(node)

         return result

    def bfs\_traversal(self, root):

         result = []

         if not root:

             return result

         queue = deque([root])

         while queue:

             current = queue.popleft()

             result.append(current.key)

             if current.left:

                 queue.append(current.left)

             if current.right:

                 queue.append(current.right)

         return result

class BinaryTreeApp:

     def \_\_init\_\_(self):

          self.tree = BinaryTree()

     def measure\_traversal\_time(self, size):

          values\_to\_insert = random.sample(range(1, size + 1), size)

          for value in values\_to\_insert:

              self.tree.insert(value)

          times = {}

          start\_time\_pre\_order = time.time()

          pre\_order\_result = self.tree.dfs\_pre\_order(self.tree.root)

          times['Pre-order'] = time.time() - start\_time\_pre\_order

          start\_time\_in\_order = time.time()

          in\_order\_result = self.tree.dfs\_in\_order(self.tree.root)

          times['In-order'] = time.time() - start\_time\_in\_order

          start\_time\_post\_order = time.time()

          post\_order\_result = self.tree.dfs\_post\_order(self.tree.root)

          times['Post-order'] = time.time() - start\_time\_post\_order

          start\_time\_bfs = time.time()

          bfs\_result = self.tree.bfs\_traversal(self.tree.root)

          times['BFS'] = time.time() - start\_time\_bfs

          return times, pre\_order\_result, in\_order\_result, post\_order\_result, bfs\_result

     def run(self):

          size = int(input("Введите размер дерева (например, 10000): "))

          search\_times, pre\_order\_result, in\_order\_result, post\_order\_result, bfs\_result= self.measure\_traversal\_time(size)

          print("\nВыберите метод обхода для вывода узлов:")

          print("1. Прямой обход (Pre-order)")

          print("2. Симметричный обход (In-order)")

          print("3. Обратный обход (Post-order)")

          print("4. Обход в ширину (BFS)")

          choice= input("Ваш выбор (введите номер): ")

          if choice == "1":

              print("Узлы в порядке Pre-order:", pre\_order\_result)

          elif choice == "2":

              print("Узлы в порядке In-order:", in\_order\_result)

          elif choice == "3":

              print("Узлы в порядке Post-order:", post\_order\_result)

          elif choice == "4":

              print("Узлы в порядке BFS:", bfs\_result)

          else:

              print("Неверный выбор.")

if \_\_name\_\_ == "\_\_main\_\_":

     app= BinaryTreeApp()

     app.run()

import random

import time

from collections import deque

import matplotlib.pyplot as plt

import numpy as np

class TreeNode:

    def \_\_init\_\_(self, key):

        self.key = key

        self.color = 'red'

        self.left = None

        self.right = None

        self.parent = None

class RedBlackTree:

    def \_\_init\_\_(self):

        self.NIL\_LEAF = TreeNode(None)

        self.NIL\_LEAF.color = 'black'

        self.root = self.NIL\_LEAF

    def insert(self, key):

        new\_node = TreeNode(key)

        new\_node.left = self.NIL\_LEAF

        new\_node.right = self.NIL\_LEAF

        parent = None

        current = self.root

        while current != self.NIL\_LEAF:

            parent = current

            if new\_node.key < current.key:

                current = current.left

            else:

                current = current.right

        new\_node.parent = parent

        if parent is None:

            self.root = new\_node

        elif new\_node.key < parent.key:

            parent.left = new\_node

        else:

            parent.right = new\_node

        new\_node.color = 'red'

        self.fix\_insert(new\_node)

    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':

                    node.parent.color = 'black'

                    uncle.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':

                    node.parent.color = 'black'

                    uncle.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 left\_rotate(self, x):

        y = x.right

        x.right = y.left

        if y.left != self.NIL\_LEAF:

            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, y):

        x = y.left

        y.left = x.right

        if x.right != self.NIL\_LEAF:

            x.right.parent = y

        x.parent = y.parent

        if y.parent is None:

            self.root = x

        elif y == y.parent.right:

            y.parent.right = x

        else:

            y.parent.left = x

        x.right = y

        y.parent = x

    def dfs\_pre\_order(self, node):

         result = []

         def traverse(n):

             if n != self.NIL\_LEAF:

                 result.append(n.key)

                 traverse(n.left)

                 traverse(n.right)

         traverse(node)

         return result

    def dfs\_in\_order(self, node):

         result = []

         def traverse(n):

             if n != self.NIL\_LEAF:

                 traverse(n.left)

                 result.append(n.key)

                 traverse(n.right)

         traverse(node)

         return result

    def dfs\_post\_order(self, node):

         result = []

         def traverse(n):

             if n != self.NIL\_LEAF:

                 traverse(n.left)

                 traverse(n.right)

                 result.append(n.key)

         traverse(node)

         return result

    def bfs\_traversal(self, root):

         result = []

         if not root or root == self.NIL\_LEAF:

             return result

         queue = deque([root])

         while queue:

             current = queue.popleft()

             result.append(current.key)

             if current.left != self.NIL\_LEAF:

                 queue.append(current.left)

             if current.right != self.NIL\_LEAF:

                 queue.append(current.right)

         return result

class RedBlackTreeApp:

     def \_\_init\_\_(self):

          self.root = None

     def measure\_traversal\_time(self, size):

          values\_to\_insert = random.sample(range(1, size + 1), size)

          tree = RedBlackTree()

          for value in values\_to\_insert:

              tree.insert(value)

          times = {}

          start\_time\_pre\_order = time.time()

          pre\_order\_result = tree.dfs\_pre\_order(tree.root)

          times['Pre-order'] = time.time() - start\_time\_pre\_order

          start\_time\_in\_order = time.time()

          in\_order\_result = tree.dfs\_in\_order(tree.root)

          times['In-order'] = time.time() - start\_time\_in\_order

          start\_time\_post\_order = time.time()

          post\_order\_result = tree.dfs\_post\_order(tree.root)

          times['Post-order'] = time.time() - start\_time\_post\_order

          start\_time\_bfs = time.time()

          bfs\_result = tree.bfs\_traversal(tree.root)

          times['BFS'] = time.time() - start\_time\_bfs

          return times, pre\_order\_result, in\_order\_result, post\_order\_result, bfs\_result

     def run(self):

          size = int(input("Введите размер дерева (например, 10000): "))

          search\_times, pre\_order\_result, in\_order\_result, post\_order\_result, bfs\_result= self.measure\_traversal\_time(size)

          print("\nВыберите метод обхода для вывода узлов:")

          print("1. Прямой обход (Pre-order)")

          print("2. Симметричный обход (In-order)")

          print("3. Обратный обход (Post-order)")

          print("4. Обход в ширину (BFS)")

          choice= input("Ваш выбор (введите номер): ")

          if choice == "1":

              print("Узлы в порядке Pre-order:", pre\_order\_result)

          elif choice == "2":

              print("Узлы в порядке In-order:", in\_order\_result)

          elif choice == "3":

              print("Узлы в порядке Post-order:", post\_order\_result)

          elif choice == "4":

              print("Узлы в порядке BFS:", bfs\_result)

          else:

              print("Неверный выбор.")

if \_\_name\_\_ == "\_\_main\_\_":

     app= RedBlackTreeApp()

     app.run()