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: Деревья поиска

Студент гр. 4352 _____ Рагимов П. Р.

Преподаватель _____ Пестерев Д.О.

Санкт-Петербург
2025

Цель работы:

Теоретическая часть

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

Практическая часть

1. Реализовать бинарное дерево поиска (BST) и следующие операции:

- поиск;
- вставка;
- удаление;
- поиск максимума;
- поиск минимума;
- прямой/центрированный/обратный обход;
- обход в ширину.

2. Реализовать на основе BST АВЛ дерево, дополнив исходную структуру/класс и операция вставки/удаления.
3. Реализовать на основе BST красно-черное дерево, дополнив исходную структуру/класс и операция вставки/удаления.
4. Получить экспериментальную зависимость высоты BST от количества ключей, при условии, что значение ключа - случайная величина, распределенная равномерно.
   Вывести полученные результаты на графики Какая асимпотика функции высоты h(n) наблюдается у двоичного дерева поиска?
5. Получить экспериментальную зависимость высоты АВЛ и красно-черного дерева от количества ключей, при условии, что значение ключа - случайная величина, распределенная равномерно.
   Вывести полученные результаты на графики. Сравнить с графиками теоретической верхней и нижней оценкой высоты.
6. Получить экспериментальную зависимость высоты АВЛ и красно-черного дерева от количества ключей, при условии, что значения ключей монотонно возрастают.
   Вывести полученные результаты на графики. Сравнить с графиками теоретической верхней и нижней оценкой высоты.

Основные теоретические положения.

АВЛ-дерево

Определение: АВЛ-дерево представляет собой бинарное дерево поиска, сбалансированное по высоте. Для любого узла дерева разница высот его левого и правого поддеревьев не превышает единицы. Балансировка обеспечивается через отслеживание баланс-фактора, вычисляемого как разность высот правого и левого поддеревьев. Допустимые значения баланс-фактора: -1, 0, 1.

Алгоритм вставки:
Процесс начинается со стандартной процедуры вставки в бинарное дерево поиска - элемент сравнивается с текущим узлом и в зависимости от результата перемещается в левое или правое поддерево. После вставки выполняется балансировка дерева. Если обнаруживается, что высота левого поддерева превышает высоту правого более чем на 1, выполняется правый поворот. При этом узел, относительно которого выполняется поворот, заменяется своим левым потомком, а сам становится правым потомком этого узла. Соответственно, если перевешивает правое поддерево, выполняется левый поворот. В некоторых случаях требуются двойные повороты: большой левый (правый поворот правого потомка с последующим левым поворотом исходного узла) или большой правый (левый поворот левого потомка с последующим правым поворотом исходного узла).

Алгоритм удаления:
Удаление выполняется по правилам стандартного бинарного дерева поиска с последующей балансировкой. Если удаляемый узел не имеет потомков, он просто удаляется. При наличии одного потомка узел заменяется этим потомком. Если у узла два потомка, он заменяется либо наибольшим элементом левого поддерева, либо наименьшим элементом правого поддерева. После удаления выполняется проверка баланс-факторов всех узлов на пути от удаленного элемента до корня, при необходимости выполняются балансировочные повороты.

Оценка высоты:
Для получения верхней оценки высоты рассмотрим дерево с минимальным количеством узлов при заданной высоте h, где высота левого поддерева равна h-1, а правого - h-2. Количество узлов в таком дереве описывается рекуррентным соотношением: $N(h) = 1 + N(h-1) + N(h-2)$. Данное соотношение соответствует последовательности Фибоначчи, что позволяет получить приближенную формулу: $N(h) \approx \varphi^{(h+2)}/\sqrt{5} - 1$, где $\varphi = 1.618$ (золотое сечение). Выражая высоту через количество узлов и логарифмируя, получаем $h \leq 1.44 \cdot \log_2 n$, что свидетельствует о логарифмической зависимости высоты от количества элементов.

Красно-черное дерево

Определение: Красно-черное дерево - это бинарное дерево поиска, в котором каждый узел имеет цветовую метку (красный или черный). Корень дерева всегда черный, а красные узлы могут иметь только черных потомков. Вместо стандартных листьев используются псевдолисты черного цвета. Дерево сбалансировано по черной высоте - количеству черных узлов на пути от корня к любому псевдолисту, причем все такие пути содержат одинаковое количество черных узлов.

Алгоритм вставки:
Изначально вставка выполняется как в обычном бинарном дереве поиска, причем новому узлу присваивается красный цвет для сохранения черной высоты. Если вставленный узел является корнем, он перекрашивается в черный. В остальных случаях возможны три основных сценария балансировки. Если у красного узла красный дядя, выполняется перекрашивание родителя и дяди в черный цвет, а деда - в красный, после чего проверка балансировки продолжается от деда. Если дядя черный и вставленный узел является правым потомком, а его родитель - левым потомком деда, выполняется левый поворот относительно родителя. Если же вставленный узел - левый потомок и его родитель также левый потомок деда, выполняется правый поворот относительно деда, завершающий балансировку.

Алгоритм удаления:
Удаление следует стандартной процедуре для бинарных деревьев поиска с последующей коррекцией свойств красно-черного дерева. Основная сложность возникает при удалении черного узла, что может нарушить свойство черной высоты. В этом случае вводится понятие "двойного черного" узла, который компенсирует удаленную черную вершину. Для устранения двойного черного цвета рассматриваются различные случаи в зависимости от цвета брата и его потомков, включая перекрашивания и повороты различного типа.

Оценка высоты:
Черная высота bh удовлетворяет неравенству $n \geq 2^{bh} - 1$, откуда следует $bh \leq \log_2(n+1)$. Учитывая, что между двумя черными узлами может находиться не более одного красного, получаем $bh \geq h/2$. Комбинируя эти неравенства, приходим к оценке высоты: $h \leq 2 \cdot \log_2(n+1)$, что гарантирует логарифмическую зависимость высоты дерева от количества элементов.

ПРАКТИЧЕСКАЯ ЧАСТЬ

```
================================================================================
ЛАБОРАТОРНАЯ РАБОТА №2: АНАЛИЗ БИНАРНЫХ ДЕРЕВЬЕВ ПОИСКА
================================================================================
Случайные ключи: [4, 5, 12, 14, 15, 18, 29, 32, 36, 55, 70, 76, 82, 87, 95]
==========================================================================
ДЕМОНСТРАЦИЯ ПОИСКА
==========================================================================

Поиск ключа 18:
BST: НАЙДЕН
AVL: НАЙДЕН
RBT: НАЙДЕН

Поиск ключа 999:
BST: НЕ НАЙДЕН
AVL: НЕ НАЙДЕН
RBT: НЕ НАЙДЕН

==========================================================================
ДЕМОНСТРАЦИЯ УДАЛЕНИЯ
==========================================================================

Удаление существующего ключа 32:
Ключ удален из всех деревьев

Удаление несуществующего ключа 888:
Такого ключа нет в деревьях
```

Рис.1 – Поиск и удаление

```
=== BST ДЕРЕВО ===
Высота: 6
Обход в ширину: 82 15 95 4 36 87 14 29 70 12 18 55 76 5
Прямой обход: 82 15 4 14 12 5 36 29 18 70 55 76 95 87
Симметричный обход: 4 5 12 14 15 18 29 36 55 70 76 82 87 95
Обратный обход: 5 12 14 4 18 29 55 76 70 36 15 87 95 82
```

Рис.2 – Обходы BST

```
=== AVL ДЕРЕВО ===
Высота: 5
Обход в ширину: 36 15 76 12 29 70 87 4 14 18 55 82 95 5
Прямой обход: 36 15 12 4 5 14 29 18 76 70 55 87 82 95
Симметричный обход: 4 5 12 14 15 18 29 36 55 70 76 82 87 95
Обратный обход: 5 4 14 12 18 29 15 55 70 82 95 87 76 36
Баланс: Max |balance|: 1, Min balance: -1
```

Рис.3 – Обходы AVL

```
=== RBT ДЕРЕВО ===
Высота: 5
Обход в ширину: 36 15 82 12 29 70 95 4 14 18 55 76 87 5
Прямой обход: 36 15 12 4 5 14 29 18 82 70 55 76 95 87
Симметричный обход: 4 5 12 14 15 18 29 36 55 70 76 82 87 95
Обратный обход: 5 4 14 12 18 29 15 55 76 70 87 95 82 36
```

Рис.4 – Обходы RBT

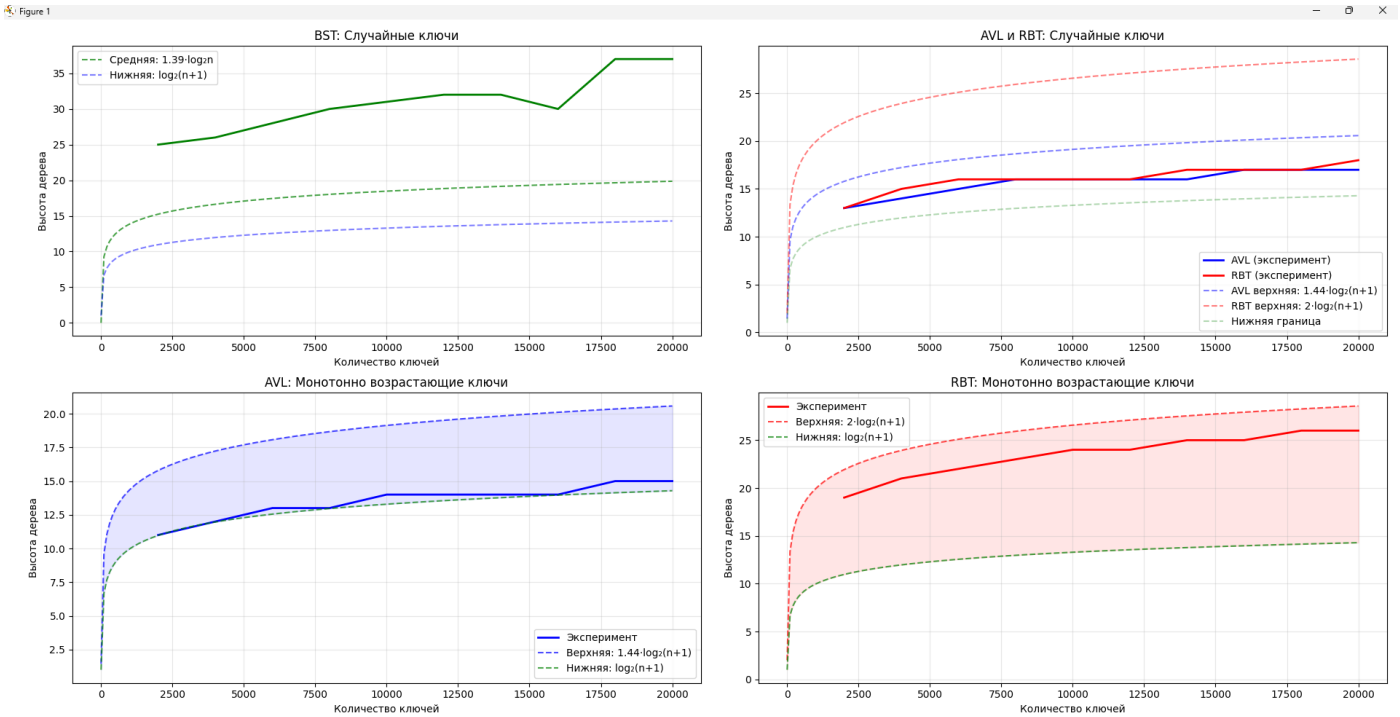

Рис.5 – Графики

ВЫВОД

Код демонстрирует экспериментальное сравнение высот трёх типов деревьев поиска: обычного BST, AVL и красно-черного (RB) деревьев. Результаты показывают, что BST имеет линейную верхнюю границу высоты (O(n)) и сильно зависит от порядка вставки, достигая логарифмической высоты (O($log\ n$)) только при случайных ключах. AVL дерево гарантирует строгую балансировку с высотой не более 1.44 $log_2 n$, что обеспечивает самый быстрый поиск, но требует частых поворотов при вставке/удалении. RB дерево имеет более слабую балансировку (высота $\leq 2\ log_2 n$), но требует меньше перестроений, что делает его эффективнее для частых дополнений. Оба сбалансированных дерева сохраняют логарифмическую высоту даже при отсортированных ключах, в отличие от BST, который вырождается в список. Таким образом, выбор структуры зависит от преобладающих операций: AVL для частого поиска, RB для частых вставок/удалений, а BST — только для простых случаев с случайными данными.

Ссылка на гитхаб - https://github.com/doublekan0/AISD_LAB2

КОД ПРОГРАММЫ.

```
IMPORT MATPLOTLIB.PYPLOT AS PLT
IMPORT RANDOM
IMPORT NUMPY AS NP
FROM COLLECTIONS IMPORT DEQUE
IMPORT SYS
IMPORT MATH

SYS.SETRECURSIONLIMIT(1000000)


CLASS TREENODE:
    DEF __INIT__(SELF, KEY):
        SELF.KEY = KEY
        SELF.LEFT = NONE
        SELF.RIGHT = NONE
        SELF.PARENT = NONE
        SELF.BALANCE_FACTOR = 0
        SELF.COLOR = 'RED'
```

```
CLASS TREEBASE:
    DEF SEARCH(SELF, ROOT, KEY):
        IF ROOT IS NONE OR GETATTR(ROOT, "KEY", NONE) == KEY:
            RETURN ROOT
        IF KEY < ROOT.KEY:
            RETURN SELF.SEARCH(ROOT.LEFT, KEY)
        RETURN SELF.SEARCH(ROOT.RIGHT, KEY)

    DEF GET_MIN_VALUE_NODE(SELF, NODE):
        CURRENT = NODE
        WHILE CURRENT AND CURRENT.LEFT:
            CURRENT = CURRENT.LEFT
        RETURN CURRENT

    DEF GET_MAX_VALUE_NODE(SELF, NODE):
        CURRENT = NODE
        WHILE CURRENT AND CURRENT.RIGHT:
            CURRENT = CURRENT.RIGHT
        RETURN CURRENT

    DEF PREORDER(SELF, ROOT):
        IF ROOT:
            PRINT(ROOT.KEY, END=' ')
            SELF.PREORDER(ROOT.LEFT)
            SELF.PREORDER(ROOT.RIGHT)

    DEF INORDER(SELF, ROOT):
        IF ROOT:
            SELF.INORDER(ROOT.LEFT)
            PRINT(ROOT.KEY, END=' ')
            SELF.INORDER(ROOT.RIGHT)

    DEF POSTORDER(SELF, ROOT):
        IF ROOT:
            SELF.POSTORDER(ROOT.LEFT)
            SELF.POSTORDER(ROOT.RIGHT)
            PRINT(ROOT.KEY, END=' ')

    DEF LEVEL_ORDER(SELF, ROOT):
        IF NOT ROOT:
            RETURN []
        RESULT, QUEUE = [], DEQUE([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

    DEF CALCULATE_HEIGHT(SELF, NODE):
        IF NOT NODE:
```

```
            RETURN 0
        RETURN 1 + MAX(SELF.CALCULATE_HEIGHT(NODE.LEFT), SELF.CALCULATE_HEIGHT(NODE.RIGHT))


CLASS BST(TREEBASE):
    DEF INSERT(SELF, ROOT, KEY):
        IF NOT ROOT:
            RETURN TREENODE(KEY)
        PARENT = NONE
        CUR = ROOT
        WHILE CUR:
            PARENT = CUR
            IF KEY == CUR.KEY:
                RETURN ROOT
            ELIF KEY < CUR.KEY:
                CUR = CUR.LEFT
            ELSE:
                CUR = CUR.RIGHT
        NEW_NODE = TREENODE(KEY)
        NEW_NODE.PARENT = PARENT
        IF KEY < PARENT.KEY:
            PARENT.LEFT = NEW_NODE
        ELSE:
            PARENT.RIGHT = NEW_NODE
        RETURN ROOT

    DEF DELETE(SELF, ROOT, KEY):
        IF NOT ROOT:
            RETURN ROOT
        IF KEY < ROOT.KEY:
            ROOT.LEFT = SELF.DELETE(ROOT.LEFT, KEY)
            IF ROOT.LEFT:
                ROOT.LEFT.PARENT = ROOT
        ELIF KEY > ROOT.KEY:
            ROOT.RIGHT = SELF.DELETE(ROOT.RIGHT, KEY)
            IF ROOT.RIGHT:
                ROOT.RIGHT.PARENT = ROOT
        ELSE:
            IF NOT ROOT.LEFT:
                TEMP = ROOT.RIGHT
                IF TEMP:
                    TEMP.PARENT = ROOT.PARENT
                RETURN TEMP
            ELIF NOT ROOT.RIGHT:
                TEMP = ROOT.LEFT
                IF TEMP:
                    TEMP.PARENT = ROOT.PARENT
                RETURN TEMP
            TEMP = SELF.GET_MIN_VALUE_NODE(ROOT.RIGHT)
            ROOT.KEY = TEMP.KEY
            ROOT.RIGHT = SELF.DELETE(ROOT.RIGHT, TEMP.KEY)
            IF ROOT.RIGHT:
```

```
            ROOT.RIGHT.PARENT = ROOT
        RETURN ROOT

    DEF GET_HEIGHT(SELF, ROOT):
        RETURN SELF.CALCULATE_HEIGHT(ROOT)


CLASS AVL(BST):
    DEF _CALCULATE_NODE_BALANCE(SELF, NODE):
        IF NOT NODE:
            RETURN 0
        LEFT_HEIGHT = SELF.CALCULATE_HEIGHT(NODE.LEFT)
        RIGHT_HEIGHT = SELF.CALCULATE_HEIGHT(NODE.RIGHT)
        RETURN LEFT_HEIGHT - RIGHT_HEIGHT

    DEF _UPDATE_BALANCE_FACTORS(SELF, NODE):
        IF NOT NODE:
            RETURN
        NODE.BALANCE_FACTOR = SELF._CALCULATE_NODE_BALANCE(NODE)
        SELF._UPDATE_BALANCE_FACTORS(NODE.LEFT)
        SELF._UPDATE_BALANCE_FACTORS(NODE.RIGHT)

    DEF ROTATE_RIGHT(SELF, Y):
        X = Y.LEFT
        T2 = X.RIGHT
        X.RIGHT = Y
        Y.LEFT = T2
        IF T2:
            T2.PARENT = Y
        X.PARENT = Y.PARENT
        Y.PARENT = X
        SELF._UPDATE_BALANCE_FACTORS(Y)
        SELF._UPDATE_BALANCE_FACTORS(X)
        RETURN X

    DEF ROTATE_LEFT(SELF, X):
        Y = X.RIGHT
        T2 = Y.LEFT
        Y.LEFT = X
        X.RIGHT = T2
        IF T2:
            T2.PARENT = X
        Y.PARENT = X.PARENT
        X.PARENT = Y
        SELF._UPDATE_BALANCE_FACTORS(X)
        SELF._UPDATE_BALANCE_FACTORS(Y)
        RETURN Y

    DEF INSERT(SELF, ROOT, KEY):
        IF NOT ROOT:
            RETURN TREENODE(KEY)
        IF KEY == ROOT.KEY:
```

```
        RETURN ROOT
    IF KEY < ROOT.KEY:
        ROOT.LEFT = SELF.INSERT(ROOT.LEFT, KEY)
        IF ROOT.LEFT:
            ROOT.LEFT.PARENT = ROOT
    ELSE:
        ROOT.RIGHT = SELF.INSERT(ROOT.RIGHT, KEY)
        IF ROOT.RIGHT:
            ROOT.RIGHT.PARENT = ROOT
    ROOT.BALANCE_FACTOR = SELF._CALCULATE_NODE_BALANCE(ROOT)
    BALANCE = ROOT.BALANCE_FACTOR
    IF BALANCE > 1 AND KEY < ROOT.LEFT.KEY:
        RETURN SELF.ROTATE_RIGHT(ROOT)
    IF BALANCE < -1 AND KEY > ROOT.RIGHT.KEY:
        RETURN SELF.ROTATE_LEFT(ROOT)
    IF BALANCE > 1 AND KEY > ROOT.LEFT.KEY:
        ROOT.LEFT = SELF.ROTATE_LEFT(ROOT.LEFT)
        IF ROOT.LEFT:
            ROOT.LEFT.PARENT = ROOT
        RETURN SELF.ROTATE_RIGHT(ROOT)
    IF BALANCE < -1 AND KEY < ROOT.RIGHT.KEY:
        ROOT.RIGHT = SELF.ROTATE_RIGHT(ROOT.RIGHT)
        IF ROOT.RIGHT:
            ROOT.RIGHT.PARENT = ROOT
        RETURN SELF.ROTATE_LEFT(ROOT)
    RETURN ROOT

DEF DELETE(SELF, ROOT, KEY):
    IF NOT ROOT:
        RETURN ROOT
    IF KEY < ROOT.KEY:
        ROOT.LEFT = SELF.DELETE(ROOT.LEFT, KEY)
        IF ROOT.LEFT:
            ROOT.LEFT.PARENT = ROOT
    ELIF KEY > ROOT.KEY:
        ROOT.RIGHT = SELF.DELETE(ROOT.RIGHT, KEY)
        IF ROOT.RIGHT:
            ROOT.RIGHT.PARENT = ROOT
    ELSE:
        IF NOT ROOT.LEFT:
            TEMP = ROOT.RIGHT
            IF TEMP:
                TEMP.PARENT = ROOT.PARENT
            RETURN TEMP
        ELIF NOT ROOT.RIGHT:
            TEMP = ROOT.LEFT
            IF TEMP:
                TEMP.PARENT = ROOT.PARENT
            RETURN TEMP
        TEMP = SELF.GET_MIN_VALUE_NODE(ROOT.RIGHT)
        ROOT.KEY = TEMP.KEY
        ROOT.RIGHT = SELF.DELETE(ROOT.RIGHT, TEMP.KEY)
```

```
            IF ROOT.RIGHT:
                ROOT.RIGHT.PARENT = ROOT
        IF NOT ROOT:
            RETURN ROOT
        ROOT.BALANCE_FACTOR = SELF._CALCULATE_NODE_BALANCE(ROOT)
        BALANCE = ROOT.BALANCE_FACTOR
        IF BALANCE > 1 AND SELF._CALCULATE_NODE_BALANCE(ROOT.LEFT) >= 0:
            RETURN SELF.ROTATE_RIGHT(ROOT)
        IF BALANCE > 1 AND SELF._CALCULATE_NODE_BALANCE(ROOT.LEFT) < 0:
            ROOT.LEFT = SELF.ROTATE_LEFT(ROOT.LEFT)
            IF ROOT.LEFT:
                ROOT.LEFT.PARENT = ROOT
            RETURN SELF.ROTATE_RIGHT(ROOT)
        IF BALANCE < -1 AND SELF._CALCULATE_NODE_BALANCE(ROOT.RIGHT) <= 0:
            RETURN SELF.ROTATE_LEFT(ROOT)
        IF BALANCE < -1 AND SELF._CALCULATE_NODE_BALANCE(ROOT.RIGHT) > 0:
            ROOT.RIGHT = SELF.ROTATE_RIGHT(ROOT.RIGHT)
            IF ROOT.RIGHT:
                ROOT.RIGHT.PARENT = ROOT
            RETURN SELF.ROTATE_LEFT(ROOT)
        RETURN ROOT

    DEF GET_BALANCE_INFO(SELF, ROOT):
        IF NOT ROOT:
            RETURN "EMPTY TREE"
        MAX_BALANCE = 0
        MIN_BALANCE = 0

        DEF TRAVERSE(NODE):
            NONLOCAL MAX_BALANCE, MIN_BALANCE
            IF NODE:
                MAX_BALANCE = MAX(MAX_BALANCE, ABS(NODE.BALANCE_FACTOR))
                MIN_BALANCE = MIN(MIN_BALANCE, NODE.BALANCE_FACTOR)
                TRAVERSE(NODE.LEFT)
                TRAVERSE(NODE.RIGHT)

        TRAVERSE(ROOT)
        RETURN F"MAX |BALANCE|: {MAX_BALANCE}, MIN BALANCE: {MIN_BALANCE}"


CLASS RBT:
    DEF __INIT__(SELF):
        SELF.NIL = TREENODE(NONE)
        SELF.NIL.COLOR = 'BLACK'
        SELF.NIL.LEFT = SELF.NIL.RIGHT = SELF.NIL.PARENT = SELF.NIL
        SELF.ROOT = SELF.NIL

    DEF SEARCH(SELF, KEY):
        RETURN SELF._SEARCH(SELF.ROOT, KEY)

    DEF _SEARCH(SELF, NODE, KEY):
        IF NODE == SELF.NIL OR KEY == NODE.KEY:
```

```
        RETURN NODE
    IF KEY < NODE.KEY:
        RETURN SELF._SEARCH(NODE.LEFT, KEY)
    RETURN SELF._SEARCH(NODE.RIGHT, KEY)

DEF GET_MIN(SELF):
    IF SELF.ROOT == SELF.NIL:
        RETURN NONE
    NODE = SELF.ROOT
    WHILE NODE.LEFT != SELF.NIL:
        NODE = NODE.LEFT
    RETURN NODE.KEY

DEF GET_MAX(SELF):
    IF SELF.ROOT == SELF.NIL:
        RETURN NONE
    NODE = SELF.ROOT
    WHILE NODE.RIGHT != SELF.NIL:
        NODE = NODE.RIGHT
    RETURN NODE.KEY

DEF PREORDER_TRAVERSAL(SELF):
    RESULT = []
    SELF._PREORDER_HELPER(SELF.ROOT, RESULT)
    RETURN RESULT

DEF _PREORDER_HELPER(SELF, NODE, RESULT):
    IF NODE != SELF.NIL:
        RESULT.APPEND(NODE.KEY)
        SELF._PREORDER_HELPER(NODE.LEFT, RESULT)
        SELF._PREORDER_HELPER(NODE.RIGHT, RESULT)

DEF INORDER_TRAVERSAL(SELF):
    RESULT = []
    SELF._INORDER_HELPER(SELF.ROOT, RESULT)
    RETURN RESULT

DEF _INORDER_HELPER(SELF, NODE, RESULT):
    IF NODE != SELF.NIL:
        SELF._INORDER_HELPER(NODE.LEFT, RESULT)
        RESULT.APPEND(NODE.KEY)
        SELF._INORDER_HELPER(NODE.RIGHT, RESULT)

DEF POSTORDER_TRAVERSAL(SELF):
    RESULT = []
    SELF._POSTORDER_HELPER(SELF.ROOT, RESULT)
    RETURN RESULT

DEF _POSTORDER_HELPER(SELF, NODE, RESULT):
    IF NODE != SELF.NIL:
        SELF._POSTORDER_HELPER(NODE.LEFT, RESULT)
        SELF._POSTORDER_HELPER(NODE.RIGHT, RESULT)
```

```
        RESULT.APPEND(NODE.KEY)

DEF LEVEL_ORDER_TRAVERSAL(SELF):
    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

DEF INSERT(SELF, KEY):
    NODE = TREENODE(KEY)
    NODE.LEFT = NODE.RIGHT = SELF.NIL
    NODE.COLOR = 'RED'
    PARENT, CURRENT = NONE, SELF.ROOT
    WHILE CURRENT != SELF.NIL:
        PARENT = CURRENT
        CURRENT = CURRENT.LEFT IF KEY < CURRENT.KEY ELSE CURRENT.RIGHT
    NODE.PARENT = PARENT
    IF PARENT IS NONE OR PARENT == SELF.NIL:
        SELF.ROOT = NODE
        NODE.PARENT = NONE
    ELIF KEY < PARENT.KEY:
        PARENT.LEFT = NODE
    ELSE:
        PARENT.RIGHT = NODE
    SELF._FIX_INSERT(NODE)

DEF _FIX_INSERT(SELF, K):
    WHILE K.PARENT AND GETATTR(K.PARENT, "COLOR", 'BLACK') == 'RED':
        IF K.PARENT == K.PARENT.PARENT.LEFT:
            UNCLE = K.PARENT.PARENT.RIGHT
            IF GETATTR(UNCLE, "COLOR", 'BLACK') == 'RED':
                K.PARENT.COLOR = 'BLACK'
                UNCLE.COLOR = 'BLACK'
                K.PARENT.PARENT.COLOR = 'RED'
                K = K.PARENT.PARENT
            ELSE:
                IF K == K.PARENT.RIGHT:
                    K = K.PARENT
                    SELF._LEFT_ROTATE(K)
                K.PARENT.COLOR = 'BLACK'
                K.PARENT.PARENT.COLOR = 'RED'
                SELF._RIGHT_ROTATE(K.PARENT.PARENT)
        ELSE:
            UNCLE = K.PARENT.PARENT.LEFT
```

```
            IF GETATTR(UNCLE, "COLOR", 'BLACK') == 'RED':
                K.PARENT.COLOR = 'BLACK'
                UNCLE.COLOR = 'BLACK'
                K.PARENT.PARENT.COLOR = 'RED'
                K = K.PARENT.PARENT
            ELSE:
                IF K == K.PARENT.LEFT:
                    K = K.PARENT
                    SELF._RIGHT_ROTATE(K)
                K.PARENT.COLOR = 'BLACK'
                K.PARENT.PARENT.COLOR = 'RED'
                SELF._LEFT_ROTATE(K.PARENT.PARENT)
    IF SELF.ROOT:
        SELF.ROOT.COLOR = 'BLACK'

DEF DELETE(SELF, KEY):
    NODE = SELF._SEARCH(SELF.ROOT, KEY)
    IF NODE == SELF.NIL:
        RETURN
    SELF._DELETE_NODE(NODE)

DEF _DELETE_NODE(SELF, NODE):
    Y = NODE
    Y_ORIGINAL_COLOR = Y.COLOR
    IF NODE.LEFT == SELF.NIL:
        X = NODE.RIGHT
        SELF._TRANSPLANT(NODE, NODE.RIGHT)
    ELIF NODE.RIGHT == SELF.NIL:
        X = NODE.LEFT
        SELF._TRANSPLANT(NODE, NODE.LEFT)
    ELSE:
        Y = SELF._MINIMUM(NODE.RIGHT)
        Y_ORIGINAL_COLOR = Y.COLOR
        X = Y.RIGHT
        IF Y.PARENT == NODE:
            IF X:
                X.PARENT = Y
        ELSE:
            SELF._TRANSPLANT(Y, Y.RIGHT)
            Y.RIGHT = NODE.RIGHT
            Y.RIGHT.PARENT = Y
        SELF._TRANSPLANT(NODE, Y)
        Y.LEFT = NODE.LEFT
        Y.LEFT.PARENT = Y
        Y.COLOR = NODE.COLOR
    IF Y_ORIGINAL_COLOR == 'BLACK':
        SELF._FIX_DELETE(X)

DEF _MINIMUM(SELF, NODE):
    WHILE NODE.LEFT != SELF.NIL:
        NODE = NODE.LEFT
    RETURN NODE
```

```
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
    IF V IS NOT NONE:
        V.PARENT = U.PARENT

DEF _FIX_DELETE(SELF, X):
    WHILE X != SELF.ROOT AND GETATTR(X, "COLOR", 'BLACK') == 'BLACK':
        IF X == X.PARENT.LEFT:
            SIBLING = X.PARENT.RIGHT
            IF GETATTR(SIBLING, "COLOR", 'BLACK') == 'RED':
                SIBLING.COLOR = 'BLACK'
                X.PARENT.COLOR = 'RED'
                SELF._LEFT_ROTATE(X.PARENT)
                SIBLING = X.PARENT.RIGHT
            IF GETATTR(SIBLING.LEFT, "COLOR", 'BLACK') == 'BLACK' AND GETATTR(SIBLING.RIGHT, "COLOR",
                                                    'BLACK') == 'BLACK':
                SIBLING.COLOR = 'RED'
                X = X.PARENT
            ELSE:
                IF GETATTR(SIBLING.RIGHT, "COLOR", 'BLACK') == 'BLACK':
                    SIBLING.LEFT.COLOR = 'BLACK'
                    SIBLING.COLOR = 'RED'
                    SELF._RIGHT_ROTATE(SIBLING)
                    SIBLING = X.PARENT.RIGHT
                SIBLING.COLOR = X.PARENT.COLOR
                X.PARENT.COLOR = 'BLACK'
                SIBLING.RIGHT.COLOR = 'BLACK'
                SELF._LEFT_ROTATE(X.PARENT)
                X = SELF.ROOT
        ELSE:
            SIBLING = X.PARENT.LEFT
            IF GETATTR(SIBLING, "COLOR", 'BLACK') == 'RED':
                SIBLING.COLOR = 'BLACK'
                X.PARENT.COLOR = 'RED'
                SELF._RIGHT_ROTATE(X.PARENT)
                SIBLING = X.PARENT.LEFT
            IF GETATTR(SIBLING.RIGHT, "COLOR", 'BLACK') == 'BLACK' AND GETATTR(SIBLING.LEFT, "COLOR",
                                                    'BLACK') == 'BLACK':
                SIBLING.COLOR = 'RED'
                X = X.PARENT
            ELSE:
                IF GETATTR(SIBLING.LEFT, "COLOR", 'BLACK') == 'BLACK':
                    SIBLING.RIGHT.COLOR = 'BLACK'
                    SIBLING.COLOR = 'RED'
                    SELF._LEFT_ROTATE(SIBLING)
                    SIBLING = X.PARENT.LEFT
```

```
                SIBLING.COLOR = X.PARENT.COLOR
                X.PARENT.COLOR = 'BLACK'
                SIBLING.LEFT.COLOR = 'BLACK'
                SELF._RIGHT_ROTATE(X.PARENT)
                X = SELF.ROOT
        IF X:
            X.COLOR = 'BLACK'

    DEF _LEFT_ROTATE(SELF, X):
        Y = X.RIGHT
        X.RIGHT = Y.LEFT
        IF Y.LEFT != SELF.NIL:
            Y.LEFT.PARENT = X
        Y.PARENT = X.PARENT
        IF X.PARENT IS NONE:
            SELF.ROOT = Y
            Y.PARENT = NONE
        ELIF X == X.PARENT.LEFT:
            X.PARENT.LEFT = Y
        ELSE:
            X.PARENT.RIGHT = Y
        Y.LEFT = X
        X.PARENT = Y

    DEF _RIGHT_ROTATE(SELF, X):
        Y = X.LEFT
        X.LEFT = Y.RIGHT
        IF Y.RIGHT != SELF.NIL:
            Y.RIGHT.PARENT = X
        Y.PARENT = X.PARENT
        IF X.PARENT IS NONE:
            SELF.ROOT = Y
            Y.PARENT = NONE
        ELIF X == X.PARENT.RIGHT:
            X.PARENT.RIGHT = Y
        ELSE:
            X.PARENT.LEFT = Y
        Y.RIGHT = X
        X.PARENT = Y

    DEF GET_HEIGHT(SELF):
        RETURN SELF._HEIGHT(SELF.ROOT)

    DEF _HEIGHT(SELF, NODE):
        IF NODE == SELF.NIL OR NODE IS NONE:
            RETURN 0
        RETURN 1 + MAX(SELF._HEIGHT(NODE.LEFT), SELF._HEIGHT(NODE.RIGHT))


DEF RUN_EXPERIMENT(TREE_TYPE, MAX_KEYS=20000, STEP=2000, SORTED_KEYS=FALSE, TREE_NAME=""):
    HEIGHTS, SIZES = [], []
    FOR SIZE IN RANGE(STEP, MAX_KEYS + 1, STEP):
```

```
        IF TREE_TYPE == RBT:
            TREE = RBT()
            IF SORTED_KEYS:
                KEYS = LIST(RANGE(1, SIZE + 1))
            ELSE:
                KEYS = RANDOM.SAMPLE(RANGE(MAX_KEYS * 100), SIZE)
            FOR KEY IN KEYS:
                TREE.INSERT(KEY)
            HEIGHTS.APPEND(TREE.GET_HEIGHT())
        ELSE:
            TREE = TREE_TYPE()
            ROOT = NONE
            IF SORTED_KEYS:
                KEYS = LIST(RANGE(1, SIZE + 1))
            ELSE:
                KEYS = RANDOM.SAMPLE(RANGE(MAX_KEYS * 100), SIZE)
            FOR KEY IN KEYS:
                ROOT = TREE.INSERT(ROOT, KEY)
            HEIGHTS.APPEND(TREE.GET_HEIGHT(ROOT))
        SIZES.APPEND(SIZE)
    RETURN SIZES, HEIGHTS


DEF SHOW_TREE_DEMO():
    RANDOM.SEED(42)
    KEYS = RANDOM.SAMPLE(RANGE(1, 100), 15)
    PRINT(F"СЛУЧАЙНЫЕ КЛЮЧИ: {SORTED(KEYS)}")
    BST = BST()
    AVL = AVL()
    RBT = RBT()
    BST_ROOT = NONE
    AVL_ROOT = NONE
    FOR KEY IN KEYS:
        BST_ROOT = BST.INSERT(BST_ROOT, KEY)
        AVL_ROOT = AVL.INSERT(AVL_ROOT, KEY)
        RBT.INSERT(KEY)

    PRINT("=" * 70)
    PRINT("ДЕМОНСТРАЦИЯ ПОИСКА")
    PRINT("=" * 70)

    EXISTING_KEY = KEYS[7]
    NON_EXISTING_KEY = 999

    PRINT(F"\NПОИСК КЛЮЧА {EXISTING_KEY}:")
    BST_RESULT = BST.SEARCH(BST_ROOT, EXISTING_KEY)
    AVL_RESULT = AVL.SEARCH(AVL_ROOT, EXISTING_KEY)
    RBT_RESULT = RBT.SEARCH(EXISTING_KEY)
    PRINT(F"BST: {'НАЙДЕН' IF BST_RESULT ELSE 'НЕ НАЙДЕН'}")
    PRINT(F"AVL: {'НАЙДЕН' IF AVL_RESULT ELSE 'НЕ НАЙДЕН'}")
    PRINT(F"RBT: {'НАЙДЕН' IF RBT_RESULT != RBT.NIL ELSE 'НЕ НАЙДЕН'}")
```

```
PRINT(F"\NПОИСК КЛЮЧА {NON_EXISTING_KEY}:")
BST_RESULT = BST.SEARCH(BST_ROOT, NON_EXISTING_KEY)
AVL_RESULT = AVL.SEARCH(AVL_ROOT, NON_EXISTING_KEY)
RBT_RESULT = RBT.SEARCH(NON_EXISTING_KEY)
PRINT(F"BST: {'НАЙДЕН' IF BST_RESULT ELSE 'НЕ НАЙДЕН'}")
PRINT(F"AVL: {'НАЙДЕН' IF AVL_RESULT ELSE 'НЕ НАЙДЕН'}")
PRINT(F"RBT: {'НАЙДЕН' IF RBT_RESULT != RBT.NIL ELSE 'НЕ НАЙДЕН'}")

PRINT("\N" + "=" * 70)
PRINT("ДЕМОНСТРАЦИЯ УДАЛЕНИЯ")
PRINT("=" * 70)

DELETE_KEY = KEYS[5]
NON_DELETE_KEY = 888

PRINT(F"\NУДАЛЕНИЕ СУЩЕСТВУЮЩЕГО КЛЮЧА {DELETE_KEY}:")
BST_ROOT = BST.DELETE(BST_ROOT, DELETE_KEY)
AVL_ROOT = AVL.DELETE(AVL_ROOT, DELETE_KEY)
RBT.DELETE(DELETE_KEY)
PRINT("КЛЮЧ УДАЛЕН ИЗ ВСЕХ ДЕРЕВЬЕВ")

PRINT(F"\NУДАЛЕНИЕ НЕСУЩЕСТВУЮЩЕГО КЛЮЧА {NON_DELETE_KEY}:")
BST_ROOT = BST.DELETE(BST_ROOT, NON_DELETE_KEY)
AVL_ROOT = AVL.DELETE(AVL_ROOT, NON_DELETE_KEY)
RBT.DELETE(NON_DELETE_KEY)
PRINT("ТАКОГО КЛЮЧА НЕТ В ДЕРЕВЬЯХ")

PRINT("\N" + "=" * 70)
PRINT("ОБХОДЫ ДЕРЕВЬЕВ ПОСЛЕ ОПЕРАЦИЙ")
PRINT("=" * 70)

PRINT("\N=== BST ДЕРЕВО ===")
PRINT(F"ВЫСОТА: {BST.GET_HEIGHT(BST_ROOT)}")
PRINT("ОБХОД В ШИРИНУ:", ' '.JOIN(MAP(STR, BST.LEVEL_ORDER(BST_ROOT))))
PRINT("ПРЯМОЙ ОБХОД: ", END='')
BST.PREORDER(BST_ROOT)
PRINT("\NСИММЕТРИЧНЫЙ ОБХОД: ", END='')
BST.INORDER(BST_ROOT)
PRINT("\NОБРАТНЫЙ ОБХОД: ", END='')
BST.POSTORDER(BST_ROOT)
PRINT()

PRINT("\N=== AVL ДЕРЕВО ===")
PRINT(F"ВЫСОТА: {AVL.GET_HEIGHT(AVL_ROOT)}")
PRINT("ОБХОД В ШИРИНУ:", ' '.JOIN(MAP(STR, AVL.LEVEL_ORDER(AVL_ROOT))))
PRINT("ПРЯМОЙ ОБХОД: ", END='')
AVL.PREORDER(AVL_ROOT)
PRINT("\NСИММЕТРИЧНЫЙ ОБХОД: ", END='')
AVL.INORDER(AVL_ROOT)
PRINT("\NОБРАТНЫЙ ОБХОД: ", END='')
AVL.POSTORDER(AVL_ROOT)
PRINT()
```

```
        PRINT("БАЛАНС:", AVL.GET_BALANCE_INFO(AVL_ROOT))

        PRINT("\N=== RBT ДЕРЕВО ===")
        PRINT(F"ВЫСОТА: {RBT.GET_HEIGHT()}")
        PRINT("ОБХОД В ШИРИНУ:", ' '.JOIN(MAP(STR, RBT.LEVEL_ORDER_TRAVERSAL())))
        PRINT("ПРЯМОЙ ОБХОД:", ' '.JOIN(MAP(STR, RBT.PREORDER_TRAVERSAL())))
        PRINT("СИММЕТРИЧНЫЙ ОБХОД:", ' '.JOIN(MAP(STR, RBT.INORDER_TRAVERSAL())))
        PRINT("ОБРАТНЫЙ ОБХОД:", ' '.JOIN(MAP(STR, RBT.POSTORDER_TRAVERSAL())))


    IF __NAME__ == "__MAIN__":
        PRINT("=" * 80)
        PRINT("ЛАБОРАТОРНАЯ РАБОТА №2: АНАЛИЗ БИНАРНЫХ ДЕРЕВЬЕВ ПОИСКА")
        PRINT("=" * 80)

        SHOW_TREE_DEMO()

        SIZES_BST_RANDOM, HEIGHTS_BST_RANDOM = RUN_EXPERIMENT(BST, MAX_KEYS=20000, STEP=2000,
SORTED_KEYS=FALSE,
                                    TREE_NAME="BST")
        SIZES_AVL_RANDOM, HEIGHTS_AVL_RANDOM = RUN_EXPERIMENT(AVL, MAX_KEYS=20000, STEP=2000,
SORTED_KEYS=FALSE,
                                    TREE_NAME="AVL")
        SIZES_RBT_RANDOM, HEIGHTS_RBT_RANDOM = RUN_EXPERIMENT(RBT, MAX_KEYS=20000, STEP=2000,
SORTED_KEYS=FALSE,
                                    TREE_NAME="RBT")
        SIZES_AVL_SORTED, HEIGHTS_AVL_SORTED = RUN_EXPERIMENT(AVL, MAX_KEYS=20000, STEP=2000,
SORTED_KEYS=TRUE,
                                    TREE_NAME="AVL")
        SIZES_RBT_SORTED, HEIGHTS_RBT_SORTED = RUN_EXPERIMENT(RBT, MAX_KEYS=20000, STEP=2000,
SORTED_KEYS=TRUE,
                                    TREE_NAME="RBT")

        N = NP.LINSPACE(1, 20000, 200)
        Y_THEORY_AVL = 1.44 * NP.LOG2(N + 1)
        Y_THEORY_RBT = 2 * NP.LOG2(N + 1)
        Y_THEORY_BST_AVG = 1.39 * NP.LOG2(N)
        Y_THEORY_LOWER = NP.LOG2(N + 1)

        PLT.FIGURE(FIGSIZE=(15, 10))

        PLT.SUBPLOT(2, 2, 1)
        PLT.PLOT(SIZES_BST_RANDOM, HEIGHTS_BST_RANDOM, 'G-', LINEWIDTH=2)
        PLT.PLOT(N, Y_THEORY_BST_AVG, 'G--', ALPHA=0.7, LABEL='СРЕДНЯЯ: 1.39·LOG₂N')
        PLT.PLOT(N, Y_THEORY_LOWER, 'B--', ALPHA=0.5, LABEL='НИЖНЯЯ: LOG₂(N+1)')
        PLT.TITLE('BST: СЛУЧАЙНЫЕ КЛЮЧИ')
        PLT.XLABEL('КОЛИЧЕСТВО КЛЮЧЕЙ')
        PLT.YLABEL('ВЫСОТА ДЕРЕВА')
        PLT.LEGEND()
        PLT.GRID(TRUE, ALPHA=0.3)

        PLT.SUBPLOT(2, 2, 2)
```

```
PLT.PLOT(SIZES_AVL_RANDOM, HEIGHTS_AVL_RANDOM, 'B-', LINEWIDTH=2, LABEL='AVL (ЭКСПЕРИМЕНТ)')
PLT.PLOT(SIZES_RBT_RANDOM, HEIGHTS_RBT_RANDOM, 'R-', LINEWIDTH=2, LABEL='RBT (ЭКСПЕРИМЕНТ)')
PLT.PLOT(N, Y_THEORY_AVL, 'B--', ALPHA=0.5, LABEL='AVL ВЕРХНЯЯ: 1.44·LOG₂(N+1)')
PLT.PLOT(N, Y_THEORY_RBT, 'R--', ALPHA=0.5, LABEL='RBT ВЕРХНЯЯ: 2·LOG₂(N+1)')
PLT.PLOT(N, Y_THEORY_LOWER, 'G--', ALPHA=0.3, LABEL='НИЖНЯЯ ГРАНИЦА')
PLT.TITLE('AVL И RBT: СЛУЧАЙНЫЕ КЛЮЧИ')
PLT.XLABEL('КОЛИЧЕСТВО КЛЮЧЕЙ')
PLT.YLABEL('ВЫСОТА ДЕРЕВА')
PLT.LEGEND()
PLT.GRID(TRUE, ALPHA=0.3)

PLT.SUBPLOT(2, 2, 3)
PLT.PLOT(SIZES_AVL_SORTED, HEIGHTS_AVL_SORTED, 'B-', LINEWIDTH=2, LABEL='ЭКСПЕРИМЕНТ')
PLT.PLOT(N, Y_THEORY_AVL, 'B--', ALPHA=0.7, LABEL='ВЕРХНЯЯ: 1.44·LOG₂(N+1)')
PLT.PLOT(N, Y_THEORY_LOWER, 'G--', ALPHA=0.7, LABEL='НИЖНЯЯ: LOG₂(N+1)')
PLT.FILL_BETWEEN(N, Y_THEORY_LOWER, Y_THEORY_AVL, ALPHA=0.1, COLOR='BLUE')
PLT.TITLE('AVL: МОНОТОННО ВОЗРАСТАЮЩИЕ КЛЮЧИ')
PLT.XLABEL('КОЛИЧЕСТВО КЛЮЧЕЙ')
PLT.YLABEL('ВЫСОТА ДЕРЕВА')
PLT.LEGEND()
PLT.GRID(TRUE, ALPHA=0.3)

PLT.SUBPLOT(2, 2, 4)
PLT.PLOT(SIZES_RBT_SORTED, HEIGHTS_RBT_SORTED, 'R-', LINEWIDTH=2, LABEL='ЭКСПЕРИМЕНТ')
PLT.PLOT(N, Y_THEORY_RBT, 'R--', ALPHA=0.7, LABEL='ВЕРХНЯЯ: 2·LOG₂(N+1)')
PLT.PLOT(N, Y_THEORY_LOWER, 'G--', ALPHA=0.7, LABEL='НИЖНЯЯ: LOG₂(N+1)')
PLT.FILL_BETWEEN(N, Y_THEORY_LOWER, Y_THEORY_RBT, ALPHA=0.1, COLOR='RED')
PLT.TITLE('RBT: МОНОТОННО ВОЗРАСТАЮЩИЕ КЛЮЧИ')
PLT.XLABEL('КОЛИЧЕСТВО КЛЮЧЕЙ')
PLT.YLABEL('ВЫСОТА ДЕРЕВА')
PLT.LEGEND()
PLT.GRID(TRUE, ALPHA=0.3)

PLT.TIGHT_LAYOUT()
PLT.SHOW()
```