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

КУРСОВАЯ РАБОТА
по дисциплине «Алгоритмы и структуры данных»
Тема: Исследование AVL и RB деревьев

Студент гр. 0382 ____________________ Кондратов Ю.А.

Преподаватель ____________________ Берленко Т.А.

Санкт-Петербург
2021

ЗАДАНИЕ

НА КУРСОВУЮ РАБОТУ

Студент Кондратов Ю.А

Группа 0382

Тема работы: исследование AVL и RB деревьев

Задание: "Исследование" - реализация требуемых структур данных/алгоритмов; генерация входных данных (вид входных данных определяется студентом); использование входных данных для измерения количественных характеристик структур данных, алгоритмов, действий; сравнение экспериментальных результатов с теоретическими. Вывод промежуточных данных не является строго обязательным, но должна быть возможность убедиться в корректности алгоритмов.

Студент	____________________	Кондратов Ю.А.
Преподаватель	____________________	Берленко Т.А.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

AVL и RB деревья являются схожими структурами данных. И то и другое являются сбалансированными бинарными деревьями поиска, однако, в то время как AVL дерево является идеально сбалансированным, RB дерево «почти сбалансировано». Очевидно, что идеальная балансировка требует больше времени, чем неидеальная, однако есть ли у идеально сбалансированных деревьев преимущества, нивелирующие этот недостаток? Это и предстоит узнать в ходе исследования.

1. РЕАЛИЗАЦИЯ СТРУКТУР ДАННЫХ

1.1. Бинарное дерево поиска и вращения

Основой для AVL и RB деревьев является такая структура данных, как *бинарное дерево поиска*. Для дальнейших рассуждений необходимо определить что это за структура данных.

Бинарным деревом называется иерархическая структура данных, в которой каждый узел имеет не более двух потомков (детей)[1]. *Корнем* дерева называется узел, не имеющий родителя, *листами* называются узлы, не имеющие потомков. Все остальные узлы имеют одного родителя и одного - двух потомков.

Бинарным деревом поиска (рисунок 1) называется бинарное дерево, для которого выполнены следующие условия [2]:

- оба поддерева — левое и правое — являются двоичными деревьями поиска;
- у всех узлов левого поддерева произвольного узла X значения ключей данных меньше, нежели значение ключа данных самого узла X;
- у всех узлов правого поддерева произвольного узла X значения ключей данных больше, нежели значение ключа данных самого узла X.

В данной работе используется реализация, не предусматривающая одинаковых ключей для разных узлов, поэтому все неравенства в условиях строгие.

Алгоритм поиска в бинарном дереве поиска аналогичен бинарному поиску по отсортированному массиву, за тем лишь отличием, что в качестве левого или правого подмассива берётся соответственно левое или правое поддерево. Пример применения алгоритма поиска изображён на рисунке 2.

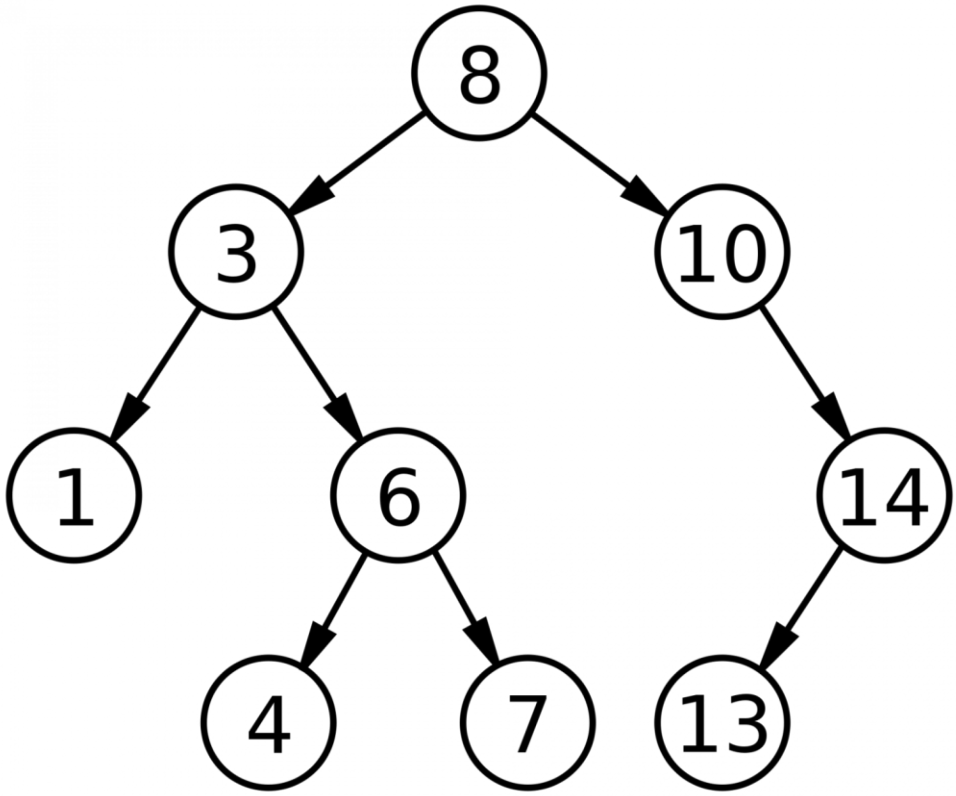

Рисунок 1 – Бинарное дерево поиска.

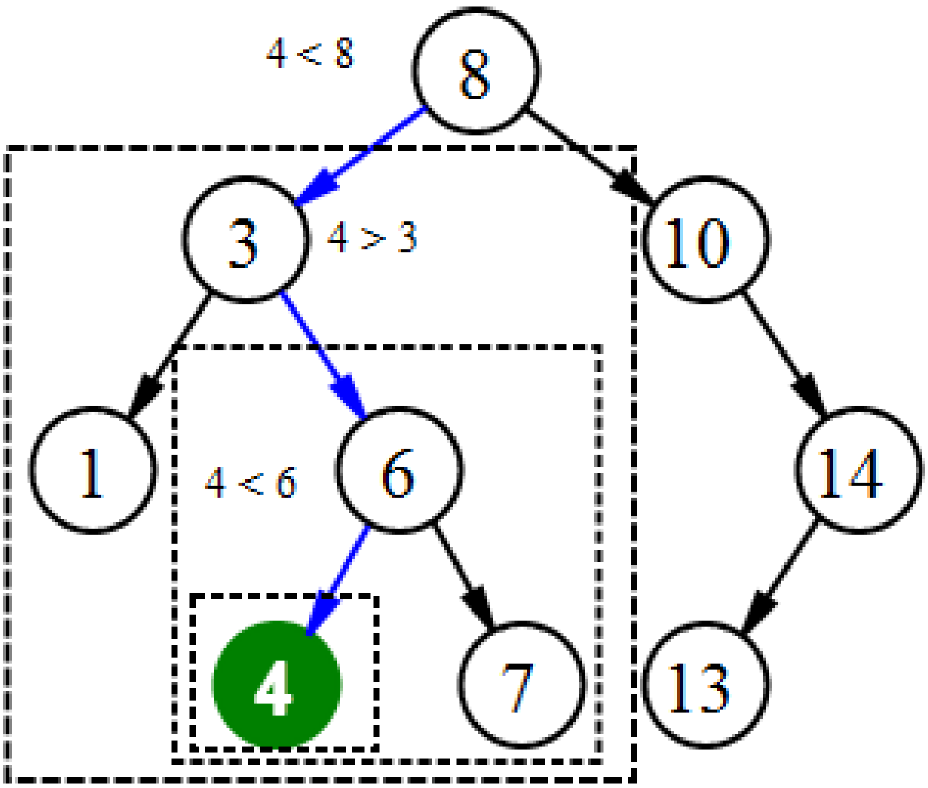

Рисунок 2 – Поиск в бинарном дереве поиска

Операцией вставки называется процесс добавления нового узла в дерево. Для реализации данной операции удобно использовать алгоритм поиска, который в случае отсутствия искомого узла в дереве, возвращает узел с предыдущего шага поиска, то есть родителя искомого узла, если бы тот находился в дереве. Таким образом, зная родителя узла, легко определить куда именно вставить узел: если ключ узла меньше ключа родителя – вставляем в качестве левого потомка, иначе – в качестве правого.

Операцией удаления называется процесс, при котором из дерева убирается узел с заданным ключом. Совершенно очевидно, как удалить узел являющийся листовым, либо узел, у которого только один потомок – назначить потомка узла потомком родителя узла, а родителя потомка (если потомок имеется) назначить родителем узла. Однако возникает вопрос – как удалить узел с двумя потомками? Для этого сначала среди листовых узлов дерева находится *замена* для удаляемого узла. Заменой является узел со следующим по величине ключом (то есть заменой для узла с ключом X является узел с ключом Y, где Y – минимальный из ключей узлов, которые больше X). Такой узел найти легко – это самый левый узел в правом поддереве удаляемого узла. После того, как замена найдена, значение из неё копируется в значение удаляемого узла, после чего удаляемым узлом становится замена и мы возвращаемся к случаю удаления листового узла, или узла с одним потомком, так как у самого левого узла в дереве не может быть левого потомка.

Левым вращение вокруг узла X называется операция, которая выполняет операцию, изображённую на рисунке 3[3]. Симметрично определяется правое вращение. Эти операции не необходимы для реализации бинарного дерева поиска, однако используются балансировке обоих деревьев.

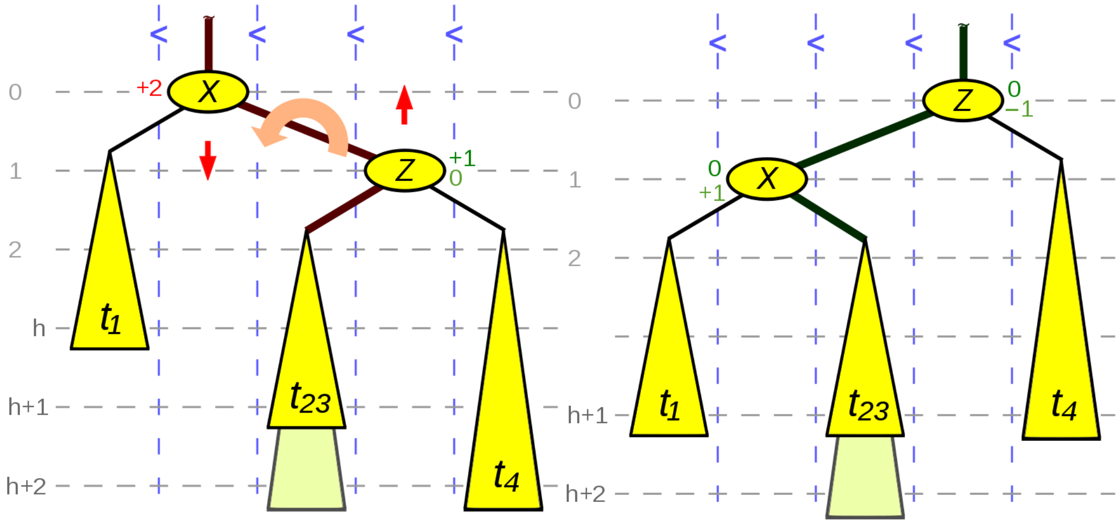

Рисунок 3 – Результат операции левого вращения

В данной работе реализация основных операций бинарного дерева поиска содержится в классе Base.

Узлом в данном дереве является объект класса Node. За родителя, левого потомка, правого потомка и ключ узла отвечают поля node.parent, node.left, node.right и node.key соответственно.

Операции левого поворота, правого поворота, поиска (с возвратом родителя), поиска с возвратом None в случае, если узел не был найден, поиска замены при удалении реализованы в методах _rotate_to_left, _rotate_to_right, _search, search, next класса Base соответственно. Операции вставки и удаления реализованы в классах AVL и RB деревьев отдельно.

1.2. Вставка и удаление в AVL дереве

AVL дерево (рисунок 4) – это сбалансированное по высоте двоичное дерево поиска: для каждой его вершины высота её двух поддеревьев различается не более чем на 1[3].

Операции вставки и удаления в AVL дереве реализуются аналогично вставке и удалению в бинарном дереве поиска, однако после их выполнения может нарушится баланс дерева, поэтому вызывается *функция балансировки*.

Функция балансировки работает следующим образом: если высота левого поддерева балансируемого узла больше высоты правого дерева минимум на два, то вызывается *функция левой балансировки.* В противном случае вызывается функция правой балансировки. В конце рекурсивно вызывается функция балансировки для родителя текущего узла.

Функция левой балансировки делает поворот влево вокруг переданного ей узла, если высота правого его поддерева больше высоты левого и независимо от этого условия делает правый поворот вокруг родителя переданного узла.

Симметрично работает *функция правой балансировки.*

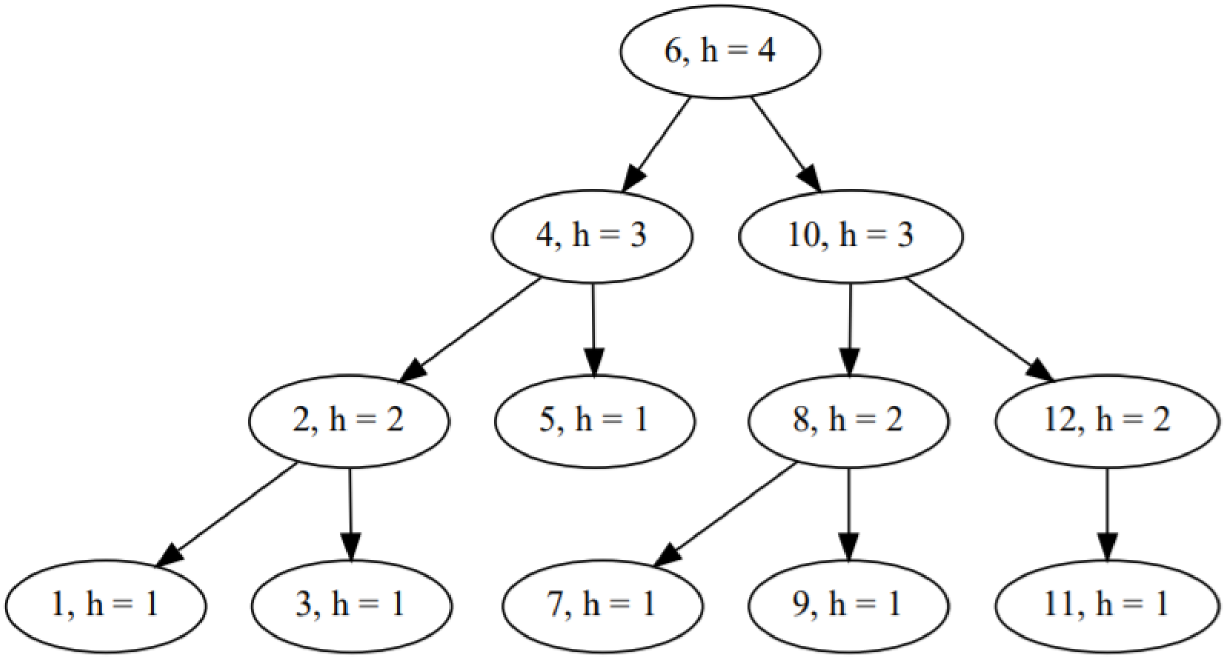

Рисунок 4 – Пример AVL дерева

Узлы авл дерева реализованы при помощи класса AVLNode, который наследуется от класса Node. Объекты класса AVLNode содеражт дополнительное поле – высоту узла. В AVLNode реализована функция подсчёта высоты узла по формуле: max(высота правого узла, высота левого узла) + 1. Данная функция используется при балансировке.

Функции балансировки, левой балансировки, правой балансировки реализованы, вставки, удаления в соответствующих методах __balance, __balance_left, __balance_right, insert, remove класса AVLTree.

1.2. Вставка и удаление в RB дереве

RB дерево (рисунок 5) – это двоичное дерево поиска, в котором каждый узел окрашен либо в чёрный либо в красный цвет. Также для каждого узла должны быть выполнены следующие условия [4]:

- узел может быть либо красным, либо чёрным и имеет двух потомков;
- корень — как правило чёрный. Это правило слабо влияет на работоспособность модели, так как цвет корня всегда можно изменить с красного на чёрный;
- все листья, не содержащие данных — чёрные;
- оба потомка каждого красного узла — чёрные;
- любой простой путь от узла-предка до листового узла-потомка содержит одинаковое число чёрных узлов.

Операции вставки и удаления в AVL дереве реализуются аналогично вставке и удалению в бинарном дереве поиска, однако после их выполнения могут нарушится условия красно-черного дерева, поэтому после операций вызываются *функция перекраски при вставке* и *функция перекраски при удалении*.

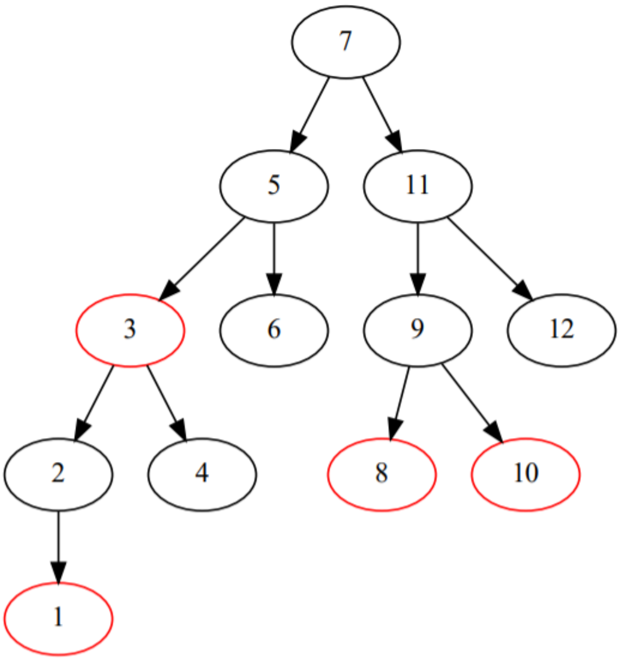

Рисунок 5 – Пример RB дерева

Реализация функций перекраски дерева предусматривает рассмотрение всех возможных случаев и, в некоторых из них, рекурсивное применение процедуры перекраски к следующим узлам, а также использование поворотов. Подробное описание реализации представлено в книге “Алгоритмы. Построение и анализ.” [6]

Узлы RB дерева реализованы при помощи класса RBNode, который наследуется от класса Node. Объекты класса AVLNode содеражт дополнительное поле – цвет узла (1 – красный, 0 – чёрный).

Функции вставки, удаления, перекраски после вставаки и перекраски после удаления реализованы в соответствующих методах insert, remove, __ins_repaint, __rm_repaint класса RBTree.

2. ИССЛЕДОВАНИЯ ВРЕМЕНИ РАБОТЫ ОСНОВНЫХ ОПЕРАЦИЙ

2.1. Исследование на случайных данных

Для чистоты эксперимента генерация случайны данных происходит при помощи HTTPS запроса к сайту random.org, который генерирует истинно случайные числа.

Исследование операции вставки проводилось следующим образом: в пустое дерево по очереди вставлялись элементы, каждый раз замерялось время, затраченное на вставку узла. Было проведено 100 тестов по 10000 случайно расположенных элементов в каждом. На графике (рисунок 6) представлены арифметически усреднённые за 100 тестов результаты.

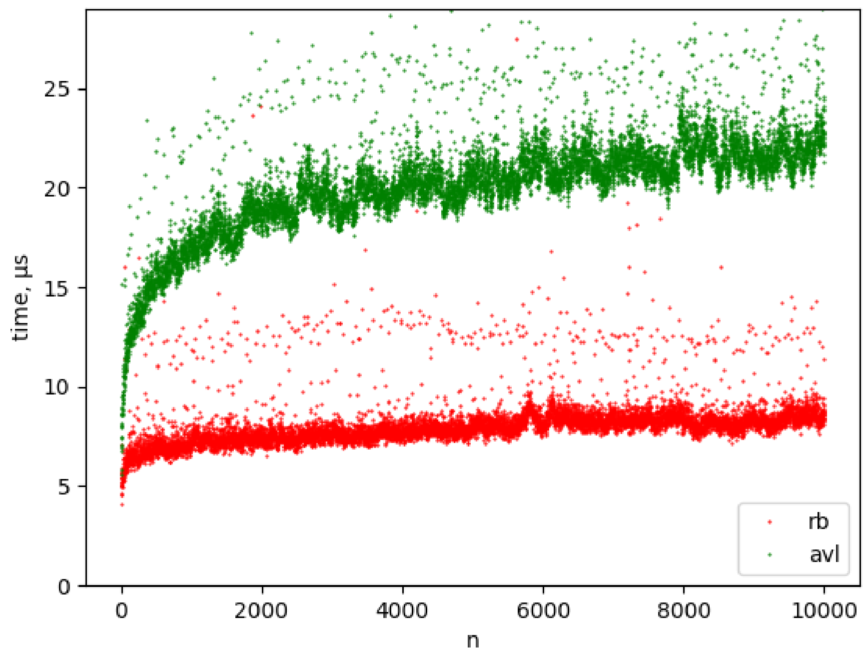

Рисунок 6 – Реузльтаты тестирования вставки случайных элементов

Исследование операции удаления проводилось следующим образом: в дерево вставлялся массив случайных элементов, после чего удалялось по одному элементу и замерялось время удаления. Параметры тестирования были как при тестировании вставки. Результаты преставлены на рисунке 7.

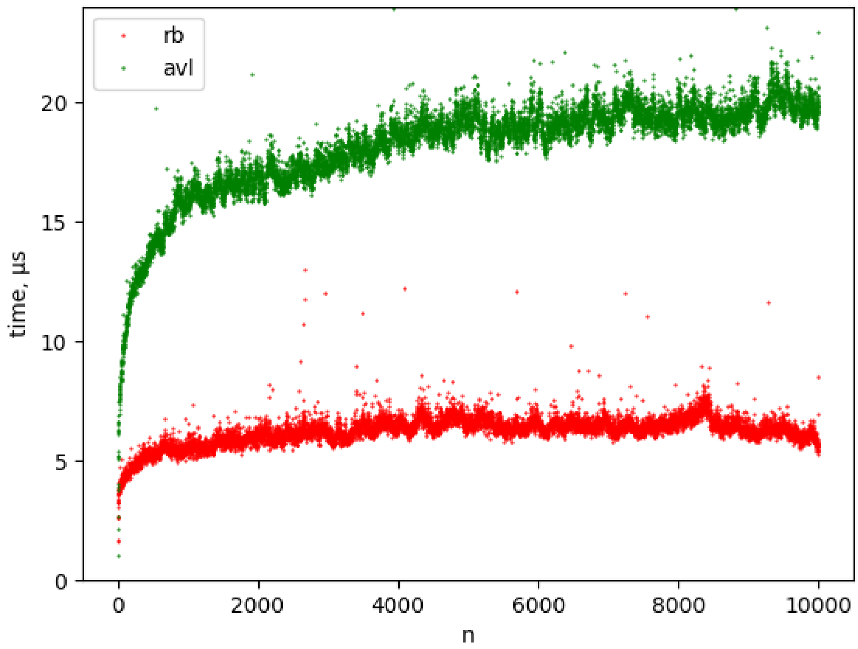

Рисунок 7 – Результаты тестирования удаления случайных элементов

Исследование поиска производилось следующим образом: в дерево вставлялся массив случайных элементов, после чего замерялось время поиска случайного элемента, которых потом удалялся. Параметры тестирования остались неизменными. Результат представлены на рисунке 8.

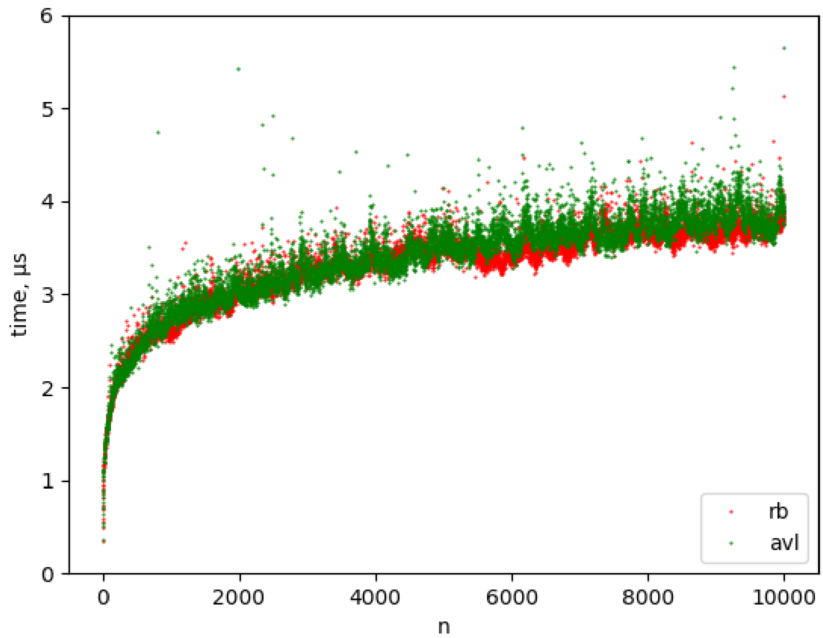

Рисунок 8 – Результаты тестирования поиска случайных элементов

2.2 Исследование на отсортированных данных

Параметры и алгоритм тестирования не отличаются от тестирования на случайных данных за тем лишь исключением, что массив чисел является отсортированным по возрастанию (в данном случае нет разницы в каком порядке сортировать массив). Результаты тестирования вставки, удаления и поиска представлены на рисунках 9, 10 и 11 соответственно.

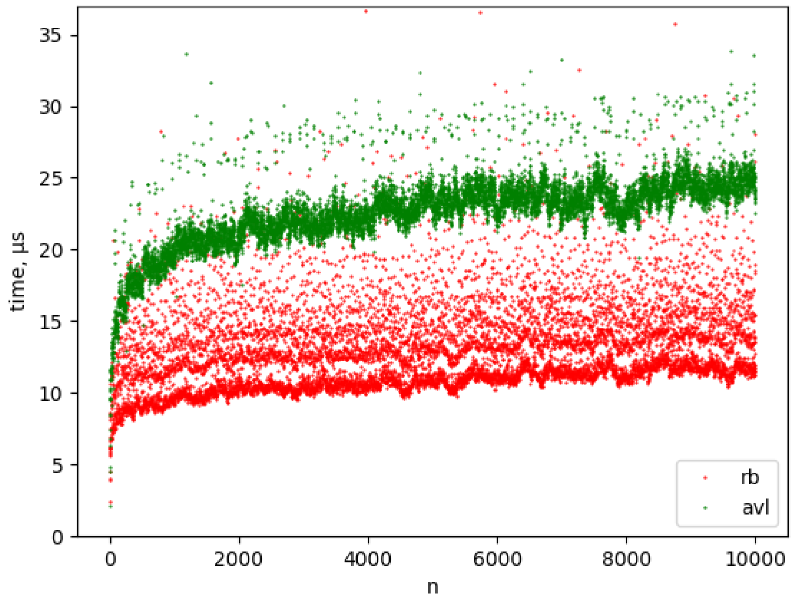

Рисунок 9 – Результат тестировани вставки отсортированных элементов

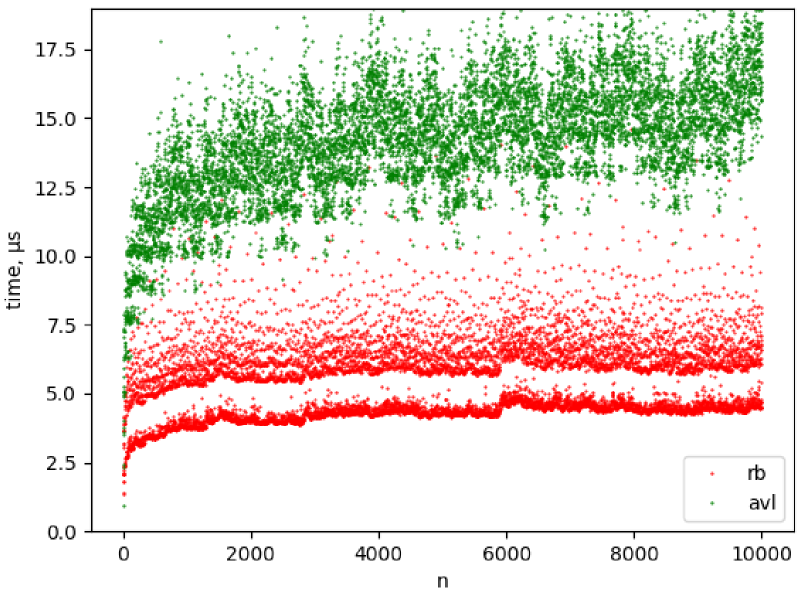

Рисунок 10 – Результат тестирования удаления отсортированных элементов

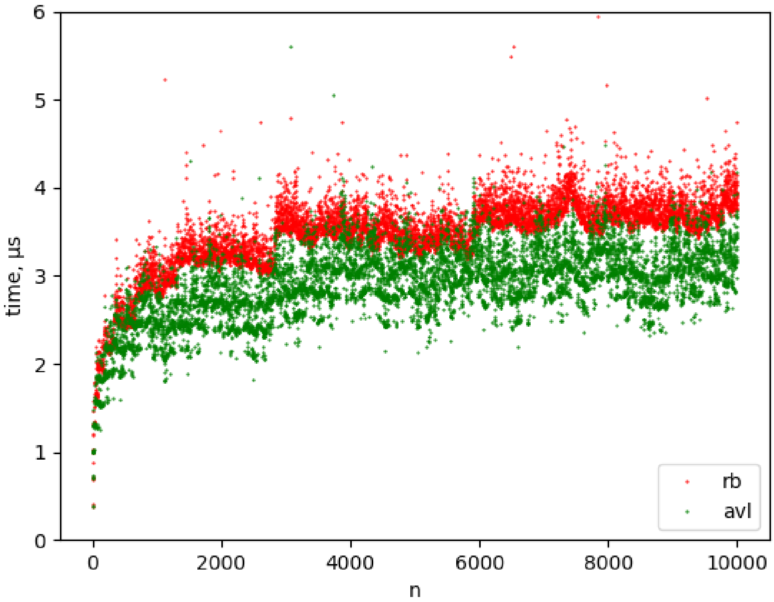

Рисунок 11 – Результат тестирования поиска отсортированных элементов

На графиках можно заметить «слоистую» структуру вставки и удаления в RB дерево. Таким образом можно проследить как меняется время вставки в зависимости от того, насколько глубоко заходит процедура перекраски. Также по «зигзагообразной» структуре графика для AVL дерева можно проследить насколько отличается время вставки и удаления при большом количестве и при малом количестве поворотов, необходимом для балансировки дерева.

3. СРАВНЕНИЕ ЭКСПЕРИМЕНТАЛЬНЫХ ДАННЫХ С ТЕОРЕТИЧЕСКИМИ

Теоретически, все операции в сбалансированных деревьях выполняются за O(h) или, что то же самое за O(log(n)), где h – высота дерева, n – количество узлов в нём. Действительно, на экспериментальных графиках прослеживается логарифмеческая зависимость времени работы от количества элементов. Также теории соответствует и то, что поиск в деревьях происходит значительно быстрее чем вставка и удаление, так как при поиске не изменяется структура дерева.

Согласно теории, поиск в АВЛ деревьях, благодаря их идеальной сбалансированности, должен производиться быстрее чем в rb деревьях, однако, как можно заметить на графиках, разница во времени поиска практически отсутствует.

Идеальная балансировка, как видно из графиков, значительно замедляет время выполнения основных операций в AVL деревьях, что в теории должно нивелироваться быстротой поиска, однако на практике этого не происходит.

Что касается памяти, то обе структуры данных требуют одинаковое её количество, так как узлы каждого дерева являются узлами бинарного дерева поиска с дополнительным целочисленным полем. Однако, если требуется минимальный расход памяти, размер дополнительного поля в RB дереве можно сократить до одного бита информации.

Таким образом, RB деревья на практике оказываются гораздо более эффективны, чем AVL, если требуется большое количество вставок и удалений. AVL деревья могут оказаться незначительно более полезны разве что в ситуации, когда практически отсутствует изменение данных, но часто происходит их поиск.

ЗАКЛЮЧЕНИЕ

Таким образом исследование-сравнение двух структур данных – AVL и RB деревьев, проведённое в данной работе, показало, что идеальная балансировка дерева требует неоправданно больших затрат по времени, чем приведение дерева к «почти сбалансированному» состоянию. Эксперимент показал, что практически отсутствующее преимущество по времени поиска в идеально сбалансированном дереве неспособно компенсировать этот недостаток.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1.Статья о бинарных деревьях // Wikipedia. URL: https://en.wikipedia.org/wiki/Binary_tree (дата обращения: 20.12.2021).

2.Статья о бинарных деревьях поиска // Habr. URL: https://habr.com/ru/post/267855 (дата обращения: 20.12.2021).

3.Статья об AVL деревьях // Wikipedia. URL: https://en.wikipedia.org/wiki/AVL_tree (дата обращения: 20.12.2021).

4. Алгоритмы: построение и анализ // Кормен, Т., Лейзерсон, Ч., Ривест, Р., Штайн. М.: Вильямс, 2005. 1296 с.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД

Файл base.py:

```
import functools


@functools.total_ordering
class Node:
    def __init__(self, key, parent=None, left=None, right=None):
        self.key = key
        self.parent = parent
        self.left = left
        self.right = right

    def __eq__(self, other):
        return self.key == other.key

    def __lt__(self, other):
        return self.key < other.key

    def __bool__(self):
        return self.key is not None


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

    def _rotate_to_left(self, node):
        right = node.right
        if node.parent:
            if node > node.parent:
                node.parent.right = right
            else:
                node.parent.left = right
        else:
            self.root = node.right
        right.parent = node.parent
```

```
        node.parent = right
        node.right = right.left
        if right.left:
            right.left.parent = node
        right.left = node

    def _rotate_to_right(self, node):
        left = node.left
        if node.parent:
            if node > node.parent:
                node.parent.right = node.left
            else:
                node.parent.left = node.left
        else:
            self.root = node.left
        left.parent = node.parent
        node.parent = left
        node.left = left.right
        if left.right:
            left.right.parent = node
        left.right = node

    def _search(self, key, root):
        if key == root.key:
            return root
        if key > root.key:
            if root.right:
                return self._search(key, root.right)
            return root
        if key < root.key:
            if root.left:
                return self._search(key, root.left)
            return root

    def search(self, key):
        res = self._search(key, self.root)
        if res.key == key:
            return res
```

```
        return None

    @staticmethod
    def next(node):
        if node.right:
            node = node.right
            while node.left:
                node = node.left
            return node
        while node.parent:
            if node.parent > node:
                break
            node = node.parent
        return node.parent
```

Файл avl_tree.py:

```
from base import Node, Base


class AVLNode(Node):
    def __init__(self, key, height=0, parent=None, left=None, right=None):
        super().__init__(key, parent, left, right)
        self.height = height

    def count_height(self):
        h_r = h_l = 0
        if self.right:
            h_r = self.right.height
        if self.left:
            h_l = self.left.height
        self.height = 1 + max(h_r, h_l)


class AVLTree(Base):
    def __init__(self, arr=None):
        super().__init__()
        if arr is None:
            arr = []
```

```
        for key in arr:
            self.insert(key)

    def insert(self, key):
        if not self.root:
            self.root = AVLNode(key)
            return

        node = self._search(key, self.root)
        if key == node.key:
            raise KeyError(f'{key} already inserted')

        if key > node.key:
            node.right = AVLNode(key, 0, node)
            node = node.right
        elif key < node.key:
            node.left = AVLNode(key, 0, node)
            node = node.left

        self.__balance(node)

    def remove(self, key):
        if not self.root:
            raise IndexError('removing from an empty tree')

        node = self._search(key, self.root)
        if node.key != key:
            raise KeyError(key)

        if not node.right and node.left:
            node.key = node.left.key
            node.left, node = None, node.left

        elif node.right:
            subst = self.next(node)
            if subst.right:
                subst.right.parent = subst.parent
            if subst > subst.parent:
```

```
                subst.parent.right = subst.right
            else:
                subst.parent.left = subst.right
            node.key = subst.key
            node = subst

        else:
            if node.parent:
                if node > node.parent:
                    node.parent.right = None
                else:
                    node.parent.left = None
            else:
                self.root = None

        if node:
            self.__balance(node.parent)

    def __balance(self, node):
        if not node:
            return

        left_h = right_h = 0
        if node.left:
            left_h = node.left.height
        if node.right:
            right_h = node.right.height

        if left_h > right_h + 1:
            self.__balance_left(node)
        if right_h > left_h + 1:
            self.__balance_right(node)
        node.count_height()

        self.__balance(node.parent)

    def __balance_left(self, node):
        node = node.left
```

```
        p = node.parent
        right_h = left_h = 0
        if node.right:
            right_h = node.right.height
        if node.left:
            left_h = node.left.height

        if right_h > left_h:
            self._rotate_to_left(node)
            node.count_height()
        self._rotate_to_right(p)

    def __balance_right(self, node):
        node = node.right
        p = node.parent
        right_h = left_h = 0
        if node.right:
            right_h = node.right.height
        if node.left:
            left_h = node.left.height

        if left_h > right_h:
            self._rotate_to_right(node)
            node.count_height()
        self._rotate_to_left(p)
```

Файл rb_tree.py:

```
from base import Node, Base


class RBNode(Node):
    def __init__(self, key=None, color=0, parent=None, left=None, right=None):
        super().__init__(key, parent, left, right)
        self.color = color

    def grandparent(self):
        if self.parent:
            return self.parent.parent
```

```
            return None

        def uncle(self):
            gp = self.grandparent()
            if not gp:
                return None
            if gp.left and self.parent == gp.left:
                return gp.right
            if gp.right and self.parent == gp.right:
                return gp.left



    # red = 1, black = 0
    class RBTree(Base):
        def __init__(self, arr=None):
            super().__init__()
            self.nil = RBNode()
            if arr is None:
                arr = []
            for key in arr:
                self.insert(key)

        def insert(self, key):
            if not self.root:
                self.root = RBNode(key, 0, None, self.nil, self.nil)
                return

            node = self._search(key, self.root)
            if key == node.key:
                raise KeyError(f'{key} already inserted')

            if key > node.key:
                node.right = RBNode(key, 1, node, self.nil, self.nil)
                node = node.right
            elif key < node.key:
                node.left = RBNode(key, 1, node, self.nil, self.nil)
                node = node.left
```

```
        self.__ins_repaint(node)

    def remove(self, key):
        if not self.root:
            raise IndexError('removing from an empty tree')

        node = self._search(key, self.root)
        if node.key != key:
            raise KeyError(key)

        if not node.left or not node.right:
            y = node
        else:
            y = self.next(node)

        if y.left:
            x = y.left
        else:
            x = y.right
        x.parent = y.parent
        if y.parent:
            if y == y.parent.left:
                y.parent.left = x
            else:
                y.parent.right = x
        else:
            self.root = x
        if y != node:
            node.key = y.key
        if not y.color:
            self.__rm_repaint(x)

    def __ins_repaint(self, node):
        p = node.parent
        if not p:
            node.color = 0
            return
        u, g = node.uncle(), node.grandparent()
```

```
        if not p.color:
            return
        if u and u.color:
            p.color = 0
            u.color = 0
            g.color = 1
            self.__ins_repaint(g)
            return
        if p.right and p.right == node and g.left and p == g.left:
            self._rotate_to_left(p)
            node = node.left
        elif p.left and p.left == node and g.right and p == g.right:
            self._rotate_to_right(p)
            node = node.right
        p, g = node.parent, node.grandparent()
        p.color = 0
        g.color = 1
        if p.left and p.left == node and g.left and p == g.left:
            self._rotate_to_right(g)
        else:
            self._rotate_to_left(g)

    def __rm_repaint(self, node):
        while self.root and node.parent and not node.color:
            if node == node.parent.left:
                brother = node.parent.right
                if brother.color:
                    brother.color = 0
                    node.parent.color = 1
                    self._rotate_to_left(node.parent)
                    brother = node.parent.right
                if not brother.left.color and not brother.right.color:
                    brother.color = 1
                    node = node.parent
                else:
                    if not brother.right.color:
```

```
                        brother.left.color = 0
                        brother.color = 1
                        self._rotate_to_right(brother)
                        brother = node.parent.right
                    brother.color = node.parent.color
                    node.parent.color = 0
                    brother.right.color = 0
                    self._rotate_to_left(node.parent)
                    node = self.root
            elif node == node.parent.right:
                brother = node.parent.left
                if brother.color:
                    brother.color = 0
                    node.parent.color = 1
                    self._rotate_to_right(node.parent)
                    brother = node.parent.left
                if not brother.right.color and not brother.left.color:
                    brother.color = 1
                    node = node.parent
                else:
                    if not brother.left.color:
                        brother.right.color = 0
                        brother.color = 1
                        self._rotate_to_left(brother)
                        brother = node.parent.left
                    brother.color = node.parent.color
                    node.parent.color = 0
                    brother.left.color = 0
                    self._rotate_to_right(node.parent)
                    node = self.root
        node.color = 0
```