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

ОТЧЕТ

по лабораторной работе №3

по дисциплине «Алгоритмы и структуры данных»

ТЕМА: РЕАЛИЗАЦИЯ И ИССЛЕДОВАНИЕ АВЛ-ДЕРЕВЬЕВ.

Студент гр. 2384 Валеева А.А.

Преподаватель Иванов Д.В.

Санкт-Петербург
2023

Цель работы

Определить концепцию и свойства АВЛ-дерева. Реализовать АВЛ-дерево, его основные методы(проверка дерева (является ли оно АВЛ деревом), нахождение разницы между связными узлами, вставка узла, удаление минимального, максимального и по значению узла, балансировка, повороты) и провести исследования, сравнив асимптотику работы с теоретическими показаниями.

Задание

В предыдущих лабораторных работах вы уже проводили исследования и эта не будет исключением. Как и в прошлые разы лабораторную работу можно разделить на две части:

1) решение задач на платформе moodle

2) исследование по заданной теме

В заданиях в качестве подсказки будет изложена основная структура данных (класс узла) и будет необходимо реализовать несколько основных функций: проверка дерева (является ли оно АВЛ деревом), нахождение разницы между связными узлами, вставка узла.

В качестве исследования нужно самостоятельно:

реализовать функции удаления узлов: любого, максимального и минимального

сравнить время и количество операций, необходимых для реализованных операций, с теоретическими оценками (очевидно, что проводить исследования необходимо на разных объемах данных)

В отчете помимо проведенного исследования необходимо приложить код всей получившей структуры: класс узла и функции.

Выполнение работы

Реализован класс *Node*, который представляет собой узел в AVL-дереве. Узел содержит значение *val,* указатели на левый и правый дочерний узел *left/right* и высоту узла *height.*

Реализованы следующие функции:

1. *height(root)* – функция принимает на вход узел AVL-дерева и возвращает высоту данного узла(количество ребер), если *root = None*, то возвращает 0.

2. *override_height(root)* – функция принимает на вход узел AVL-дерева. Функция находит максимальную высоту из высот дочерних узлов, используя функцию *height(),* и возвращает это значение, увеличенное на 1. Если нода пустая - то возвращается 0.

3. *bfactor(root)* – функция принимает на вход узел AVL-дерева и возвращает разность между высотой левого дочернего узла и правого, используя функцию height.

4. *rotate_left(root)* - функция принимает на вход узел root AVL-дерева. Создается переменная *children,* которая хранит указатель на правый узел, данного нам узла *root*. Правому узлу *root* присваивается левый узел этого правого узла, а последнему сам узел *root*. Далее применяется функция *override_height* для изменение высоты узлов сначала к *children,* потом к *root*. Функция возвращает узел *children.*

5. *rotate_right(root)* - функция принимает на вход узел root AVL-дерева. Создается переменная *q,* которая хранит указатель на левый узел, данного нам узла *root*. Левому узлу *root* присваивается правый узел этого левого узла, а последнему сам узел *root*. Далее применяется функция *override_height* для изменение высоты узлов сначала к *q,* потом к *root*. Функция возвращает узел *q*.

6. *balance(root)* - функция принимает на вход узел *root* AVL-дерева. С помощью функции *override_height* присваивается новое значение высоты узла *root*. Применяется функция *bfactor().* Далее, если разность высот более 1, то если значение разности высот правого узла *root* меньше нуля, то применяется правый поворот вокруг правого узла. И возвращается левый поворот root. Далее, если разность высот менее -1, то если значение разности высот левого узла root больше нуля, то применяется левый поворот вокруг левого узла. И возвращается правый поворот *root*. Функция возвращает узел *root.*

7. *insert(val, root)* – функция на вход приминает значение, которое нужно поместить в дерево, и корень дерева. Если корень пустой, то создается и возвращается новая нода. Если новое значение меньше значения корня, то в левый узел корня рекурсивно вызывается функция вставки, но уже с левым узлом. Иначе в правый узел корня рекурсивно вызывается функция вставки, но уже с правым узлом. Функция возвращает функцию баланса для переданного корня.

8. *find_min(root)* - функция принимает на вход корень *root* AVL-дерева. Функция рекурсивно спускается по левому поддереву, пока не примет значение *None,* и возвращает минимальное значение.

9. *remove_min(root)* - функция принимает на вход корень *root* AVL-дерева. Если левый узел root равен *None*, то возвращается правый узел *root*. Иначе в левый узел рекурсивно вызывается эта же функция. Функция возвращает сбалансированное дерево вокруг узла *root*.

10. *remove(val, root)* - функция на вход приминает значение, которое нужно поместить в дерево, и *root* - корень дерева. Если *root* равен *None,* то возвращается 0. Далее функция находит узел, в котором находится нужное значение: если значение меньше или больше находящегося в *root,* то рекурсивно вызывается функция с переданой левой или правой нодой. Найден узел: запоминаются правая и левая ноды. Если правый указатель равен *None*, то возвращается *left*. Иначе, с помощью функции *find_min()* находится минимальный элемент, вокруг которого будет выполнена балансировка и возвращена.

11. *in_order(root)* – функция центрированного обхода дерева, возвращает массив, который был заполнен следующим образом: сначала вносим левого ребенка, если он не *None*, затем корень затем правого, если не *None*.

Исследование

Теоретическая оценка вставки элемента – O(log n). Эта оценка возникает, т.к. структура AVL-дерева построена таким образом, что данные представляют собой бинарное дерево поиска(сбалансированное), тем самым убыстряя поиск места. На рисунке 1 видно, что график схож с график функции log n.

Таблица 1. Результаты замеров вставки

	Insert

Набор данных	10	100	1000	5000	10000	50000	100000	500000	100 0000
время, мс	0,5	0,41	0,551	2,2468	5,5330	30,30758	68,68670	423,423496	850,850543

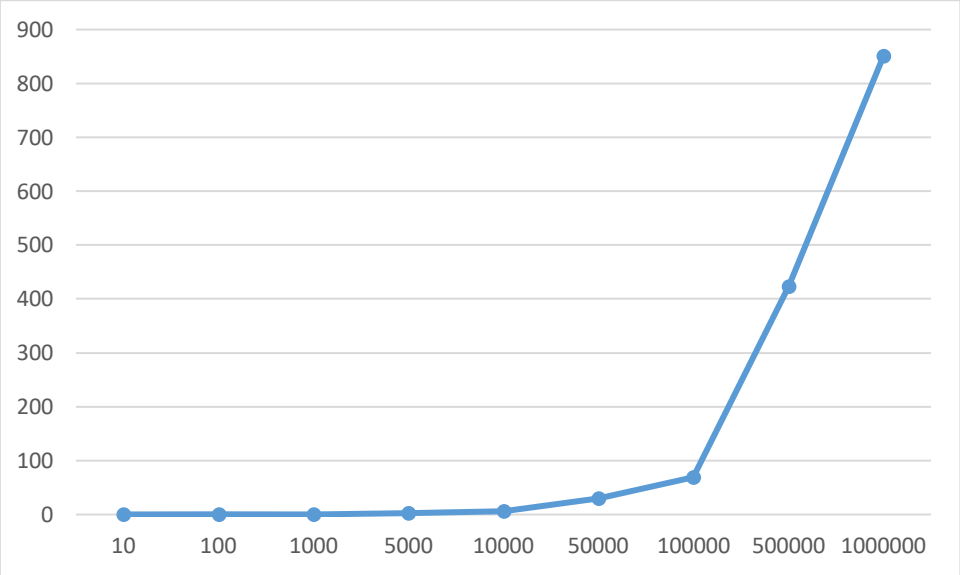

Рисунок 1 – Вставка в AVL-дерево

Было протестировано удаление минимального, максимального и случайного элемента дерева.

Таблица 2. Результаты замеров удаления

	Delete								
Набор данных	10	100	1000	5000	10000	50000	100000	500000	1 000 000
Время, мс	0,1	0,2	0,2	0,2	0,5	0,8	0,9	0,1	0,11

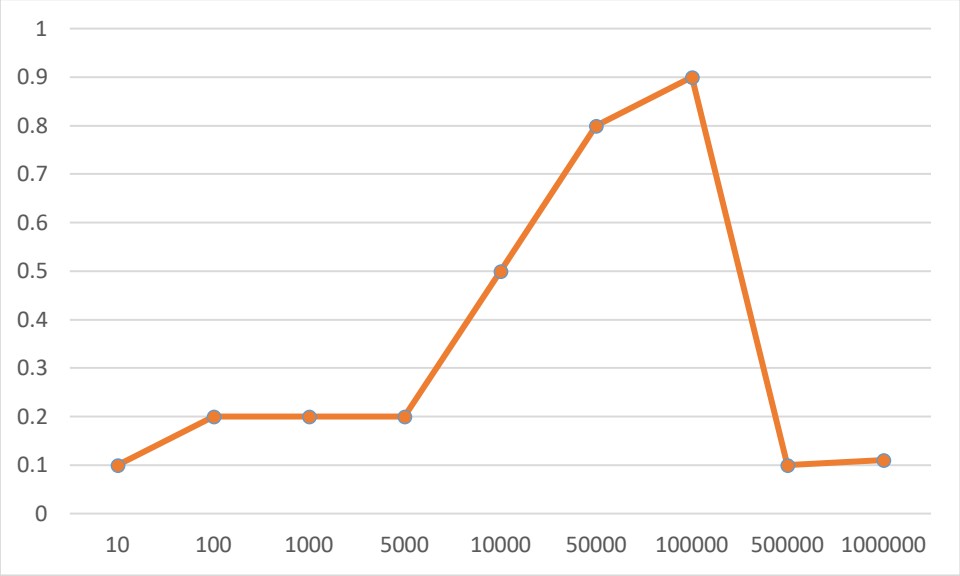

Рисунок 2 – Удаление трех элементов AVL-дерева

Разработанный программный код см. в приложении А. Тестирование реализовано с помощью *pytest*, см. в файле *tests.py*.

Выводы

Было реализовано AVL-дерево, а также его основные методы. Проведено исследование работы методов вставки и удаления на различных наборах данных. В ходе исследования была подтверждена, теоретическая оценка сложности операции (o(log(n)).

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл: node.py

```
from typing import Union
class Node:
    def __init__(self, val, left=None, right=None):
        self.val = val
        self.left: Union[Node, None] = left
        self.right: Union[Node, None] = right
        self.height = 1
def height(root): # высота дерева = макс высота из поддеревьев + 1 на "голову"
    if (root is None):
        return 0
    return (1 + max(height(root.left), height(root.right)))

def override_height(root): # переобпределение высоты
    root.height = height(root)
    return root
def bfactor(root): #проверка баланс-фактора
    hl = height(root.left)
    hr = height(root.right)
    return hr - hl
def rotate_right(root): #правый поворот
    q = root.left
    root.left = q.right
    q.right = override_height(root)
    return override_height(q)
def rotate_left(root): #левый поворот
    children = root.right
    root.right = children.left
    children.left = override_height(root)
    return override_height(children)
def balance(root): # балансировка
    root = override_height(root)
    if (bfactor(root) == 2):
        if (bfactor(root.right) < 0):
            root.right = rotate_right(root.right)
        return rotate_left(root)
    if (bfactor(root) == -2):
        if(bfactor(root.left) > 0):
            root.left = rotate_left(root.left)
        return rotate_right(root)

    return root # балансировка не нужна

def insert(val, root): # вставка ноды
    if root is None:
        return Node(val)
```

```
    if val < root.val:
        root.left = insert(val, root.left)
    else:
        root.right = insert(val, root.right)
    return balance(root)

def find_min(root): # поиск минимального узла в дереве
    if (root.left is None):
        return root
    else:
        return find_min(root.left)
def remove_min(root): # "выписывание" минимального эоемента
    if (root.left is None):
        return root.right # левый является минимумом -> возвращаем
его правого ребенка
    root.left = remove_min(root.left)
    return balance(root)
def remove(val, root): # удаление ноды
    if root is None:
        return 0
    if val < root.val:
        root.left = remove(val, root.left)
    elif val > root.val:
        root.right = remove(val, root.right)
    else: # val = root.val
        my_left = root.left
        my_right = root.right
        if my_right is None:
            return my_left
        node_min = find_min(my_right)
        node_min.right = remove_min(my_right)
        node_min.left = my_left
        return balance(node_min)
    return balance(root)
def check(root) : # проверка дерева на правильность
    if root is None:
        return True
    lh = height(root.left)
    rh = height(root.right)
    if (abs(lh - rh) <= 1 and check(root.left) and
check(root.right)):
        return True
    return False
def calculate_diffs(root):
    if root is None or (root.left is None and root.right is None):
        return []
    diffs = []
    if root.left is not None:
        diffs.append(root.val - root.left.val)
    if root.right is not None:
        diffs.append(root.right.val - root.val)
    return diffs + calculate_diffs(root.left) +
calculate_diffs(root.right)
```

```
def diff(root: Node): # мин разница между значениями связных нод
    return min(calculate_diffs(root))

def in_order(root): #объод дерева: левый->корень->правый
    if root is None:
        return []
    return [*in_order(root.left), root.val, *in_order(root.right)]
```

Файл: main.py

```
from node import Node, insert, remove, in_order
def in_order_print(node): # выводит левого потомка, затем
родителя, затем правого потомка
    if node:
        in_order_print(node.left)
        print(node.val)
        in_order_print(node.right)

if __name__ == '__main__':
    my_tree = Node(55)
    my_tree = insert(50, my_tree)
    my_tree = insert(40, my_tree)
    my_tree = insert(80, my_tree)
    my_tree = insert(34, my_tree)
    my_tree = insert(12, my_tree)
    remove(34, my_tree)
    in_order_print(my_tree)
```

Файл: tests.py

```
from node import Node, insert, remove, in_order

def test_new_tree():
    my_tree = insert(55, None)

    result = [55]
    my_answer = in_order(my_tree)
    assert result == my_answer

def test_insert():
    my_tree = insert(55, None)
    my_tree = insert(50, my_tree)
    my_tree = insert(40, my_tree)
    my_tree = insert(80, my_tree)
    my_tree = insert(34, my_tree)
    my_tree = insert(12, my_tree)

    my_answer = in_order(my_tree)
    result = [12, 34, 40, 50, 55, 80]
    assert result == my_answer
```

```
def test_remove_min():
    my_tree = insert(55, None)
    my_tree = insert(50, my_tree)
    my_tree = insert(40, my_tree)
    my_tree = insert(80, my_tree)
    my_tree = insert(34, my_tree)
    my_tree = insert(12, my_tree)

    my_tree = remove(12, my_tree)
    my_answer = in_order(my_tree)
    result = [34, 40, 50, 55, 80]
    assert result == my_answer

def test_remove_max():
    my_tree = insert(55, None)
    my_tree = insert(50, my_tree)
    my_tree = insert(40, my_tree)
    my_tree = insert(80, my_tree)
    my_tree = insert(34, my_tree)
    my_tree = insert(12, my_tree)

    my_tree = remove(80, my_tree)
    my_answer = in_order(my_tree)
    result = [12, 34, 40, 50, 55]
    assert result == my_answer

def test_remove_some_node():
    my_tree = insert(55, None)
    my_tree = insert(50, my_tree)
    my_tree = insert(40, my_tree)
    my_tree = insert(80, my_tree)
    my_tree = insert(34, my_tree)
    my_tree = insert(12, my_tree)

    my_tree = remove(34, my_tree)
    my_tree = remove(55, my_tree)

    my_answer = in_order(my_tree)
    result = [12, 40, 50, 80]
    assert result == my_answer
```