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 ___ Кузьминых Е.М

Преподаватель ___ Иванов Д.В.

Санкт-Петербург
2023

Цель работы.

Написать реализацию развернутого АВЛ-дерева и основные функции для работы с АВЛ-деревом.

Задачи.

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)

Выполнение работы.

Код состоит из класса *Node* и функций для работы с АВЛ-деревом.

Узел содержит следующие поля:

val: Значение самого узла

left: Значение следующей ветви слева.

right: значение следующей ветви справа.

Методы, реализованные для работы с АВЛ-деревом:

calculate_height(node): Эта рекурсивная функция вычисляет высоту узла node. Она посещает каждый узел и определяет его высоту,

вычисляя 1 плюс максимальную высоту его левого и правого поддеревьев. Функция возвращает высоту узла.

Check(node): Эта рекурсивная функция проверяет, является ли данное дерево сбалансированным, то есть разница между высотами его правого и левого поддеревьев не превышает 1. Функция вызывает calculate_height для каждого узла и проверяет условие сбалансированности для этого узла. Если условие не выполняется, функция возвращает False. Иначе, она продолжает рекурсивно проверять сбалансированность для каждого потомка узла. Если все условия выполняются, функция возвращает True, что означает, что дерево сбалансировано.

calculate_diff(root): Эта функция рекурсивно вычисляет минимальную разницу в значениях между родительским и потомственными узлами в дереве. Вначале функция проверяет, существуют ли значения у левого и правого потомков узла. Затем она рекурсивно вызывает себя для каждого потомка и вычисляет разницу между значениями родителя и потомка, сохраняя наименьшее значение. Функция возвращает минимальную разницу значений.

diff(root): та функция использует calculate_diff для вычисления минимальной разницы значений между родительским и потомственными узлами и возвращает это значение.

refresh_height(root): Эта функция обновляет высоту узла, основываясь на его текущем состоянии. Вначале функция проверяет, существуют ли значения у левого и правого потомков узла. Затем она вызывает calculate_height для каждого потомка и обновляет значение высоты узла, равное 1 плюс максимальная высота его потомков.

imbalance_factor(root): Эта функция возвращает разницу между высотой правого и левого поддерева узла. Она использует высоту узлов, которая уже была рассчитана с помощью refresh_height..

rotate_anti_clockwise(root) и rotate_clockwise(root): Эти функции

выполняют операции поворота, которые используются для восстановления баланса в AVL-дереве. Повороты узлов выполняются путем изменения ссылок на потомков и обновления высот вершин.

Balance_tree (root): Эта рекурсивная функция балансирует дерево, если оно становится несбалансированным после вставки или удаления узлов. Она проверяет разницу между высотами правого и левого поддерева узла и выполняет соответствующие повороты, чтобы восстановить баланс.

insert (root) и delete_node(root, val) : Эти функции позволяют вставлять и удалять узлы в AVL-дереве соответственно. Функция insert проверяет, если дерево пустое, создает новый узел с заданным значением и возвращает его как новое дерево. Иначе, она находит правильную позицию для вставки нового узла, обновляет высоты и баланс дерева, а затем вызывает функцию balance, чтобы восстановить сбалансированность. Функция delete также находит узел с заданным значением, удаляет его из дерева, обновляет высоты и баланс и вызывает функцию balance, чтобы восстановить сбалансированность.

seek_minimum(root): Эта функция рекурсивно осуществляет поиск минимального значения в AVL-дереве. Самый маленький узел находится по самому левому пути от корня, поэтому функция переходит по левым ссылкам от корня до тех пор, пока не достигнет узла, у которого нет левого потомка. Этот узел будет являться узлом с минимальным значением в дереве.

extract_minimum(root): Эта функция "изымает" узел с наименьшим значением из дерева и балансирует дерево после этого изменения. Если узел с минимальным значением не имеет левого потомка (то есть это самый маленький узел), она возвращает его правого потомка. Это приведет к удалению минимального узла, так как он больше не будет ссылаться на другие узлы. Функция затем рекурсивно обновляет ссылки на левых потомков других узлов и

балансирует дерево.

in_order(root): Эта рекурсивная функция выполняет in-order (симметричный) обход дерева. Она сначала обходит левое поддерево, затем посещает корневой узел, а затем обходит правое поддерево. Функция возвращает списком все узлы в порядке, заданном симметричным обходом.

Исследование работы АВЛ-дерева:

Ниже предоставлены графики с сравнением времени работы реализованного АВЛ дерева (метод вставки и удаления), в случае вставки одного и того-же элемента (худший случай).

Количество элементов	10	1000	10000	100000
Время работы	0.0	0.7397620677947998	81.89252543449402	1862.34523338

Таблица 1 – время работы функции insert

Количество элементов	10	1000	10000	100000
Время работы	0.0	0.6449642181396484	83.82408404350281	1864.9304

Таблица 2 – время работы функции delete_node

Тестирование.

Тестирование работоспособности написанного кода проводилось с помощью pytest

№	Входные данные	Выходные данные	Комментарий
1	def test_new_tree(): val = random.randint(1, 100) my_tree = insert(val, None) result = [val] my_answer = in_order(my_tree) assert result == my_answer, f'Error: {result} ≠ {my_answer}'	result == my_answer	Ответ верный

2	def test_remove_min():
	my_tree = None
	insert_values = random.sample(range(1, 100), 6)
	for v in insert_values:
	my_tree = insert(v, my_tree)
	
	my_tree = delete_node(min(insert_values), my_tree)
	my_answer = in_order(my_tree)
	result = sorted(insert_values)[1:] # exclude min value
	assert result == my_answer, f'Error: {result} ≠ {my_answer}'
		result == my_answer	Ответ верный

3	def test_remove_some_node():
	my_tree = None
	insert_values = random.sample(range(1, 100), 6)
	for v in insert_values:
	my_tree = insert(v, my_tree)
	
	remove_values = random.sample(insert_values, 2)
	for v in remove_values:
	my_tree =
		result == my_answer	Ответ верный

	delete_node(v, my_tree) my_answer = in_order(my_tree) result = sorted(list(set(insert_values) - set(remove_values))) assert result == my_answer, f'Error: {result} ≠ {my_answer}'		

Вывод.

В ходе выполнения лабораторной работы была написана реализация АВЛ-дерева и функций для работы с ним. Было произведено тестирование с измерением времени работы функций.

Приложение А.

Исходный код программы.

Main.py

```
class Node:
    def __init__(self, val, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right


def calculate_height(root):
    if root is None:
        return 0
    return 1 + max(calculate_height(root.left), calculate_height(root.right))


def check(root):
    if root is None:
        return True
    conditions = [
        abs(calculate_height(root.left) - calculate_height(root.right)) <= 1,
        check(root.left),
        check(root.right)
    ]
    return all(conditions)


def calculate_diff(current_root, min_diff):
    if current_root.left is not None:
        min_diff = min(min_diff, abs(current_root.val - current_root.left.val))
        min_diff = calculate_diff(current_root.left, min_diff)

    if current_root.right is not None:
        min_diff = min(min_diff, abs(current_root.val -
current_root.right.val))
        min_diff = calculate_diff(current_root.right, min_diff)

    return min_diff


def diff(root):
    if root is None:
        return float('inf')
    else:
        return calculate_diff(root, float('inf'))


def refresh_height(root):
    root.height = calculate_height(root)
    return root


def imbalance_factor(root):
```

```
    left_height = calculate_height(root.left)
    right_height = calculate_height(root.right)
    return right_height - left_height


def rotate_anti_clockwise(root):
    temp = root.left
    root.left = temp.right
    temp.right = refresh_height(root)
    return refresh_height(temp)


def rotate_clockwise(root):
    pivot = root.right
    root.right = pivot.left
    pivot.left = refresh_height(root)
    return refresh_height(pivot)


def balance_tree(root):
    root = refresh_height(root)
    if imbalance_factor(root) == 2:
        if imbalance_factor(root.right) < 0:
            root.right = rotate_anti_clockwise(root.right)
        return rotate_clockwise(root)
    if imbalance_factor(root) == -2:
        if imbalance_factor(root.left) > 0:
            root.left = rotate_clockwise(root.left)
        return rotate_anti_clockwise(root)

    return root


def insert(value, root):
    if root is None:
        return Node(value)

    if value < root.val:
        root.left = insert(value, root.left)
    else:
        root.right = insert(value, root.right)
    return balance_tree(root)


def seek_minimum(root):
    if root.left is None:
        return root
    else:
        return seek_minimum(root.left)

def extract_minimum(root):
    if root.left is None:
        return root.right
    root.left = extract_minimum(root.left)
    return balance_tree(root)

def delete_node(value, root): # удаление узла
    if root is None:
```

```
        return 0
    if value < root.val:
        root.left = delete_node(value, root.left)
    elif value > root.val:
        root.right = delete_node(value, root.right)
    else: # value = node.val
        temp_left = root.left
        temp_right = root.right
        if temp_right is None:
            return temp_left
        smallest_node = seek_minimum(temp_right)
        smallest_node.right = extract_minimum(temp_right)
        smallest_node.left = temp_left
        return balance_tree(smallest_node)
    return balance_tree(root)

def in_order(root: Node):
    if root is None:
        return []
    return [*in_order(root.left), root.val, *in_order(root.right)]
```

Tests.py

```
import random

from main import delete_node, insert, in_order


def test_new_tree():
    val = random.randint(1, 100)
    my_tree = insert(val, None)

    result = [val]
    my_answer = in_order(my_tree)
    assert result == my_answer, f'Error: {result} ≠
{my_answer}'

def test_insert():
    my_tree = None
    insert_values = random.sample(range(1, 100), 6)  # generate
6 unique random numbers
    for v in insert_values:
        my_tree = insert(v, my_tree)

    my_answer = in_order(my_tree)
    result = sorted(insert_values)
    assert result == my_answer, f'Error: {result} ≠
{my_answer}'
```

```
    def test_remove_min():
        my_tree = None
        insert_values = random.sample(range(1, 100), 6)
        for v in insert_values:
            my_tree = insert(v, my_tree)

        my_tree = delete_node(min(insert_values), my_tree)
        my_answer = in_order(my_tree)
        result = sorted(insert_values)[1:]  # exclude min value
        assert result == my_answer, f'Error: {result} ≠
    {my_answer}'

    def test_remove_max():
        my_tree = None
        insert_values = random.sample(range(1, 100), 6)
        for v in insert_values:
            my_tree = insert(v, my_tree)

        my_tree = delete_node(max(insert_values), my_tree)
        my_answer = in_order(my_tree)
        result = sorted(insert_values)[:-1]
        assert result == my_answer, f'Error: {result} ≠
    {my_answer}'

    def test_remove_some_node():
        my_tree = None
        insert_values = random.sample(range(1, 100), 6)
        for v in insert_values:
            my_tree = insert(v, my_tree)

        remove_values = random.sample(insert_values, 2)
        for v in remove_values:
            my_tree = delete_node(v, my_tree)

        my_answer = in_order(my_tree)
        result = sorted(list(set(insert_values) -
    set(remove_values)))
        assert result == my_answer, f'Error: {result} ≠
    {my_answer}' super().append(p_object)


def print_colors(self):
    for i in range(len(self)):
        print(f"{i+1} корабль: {self[i].color}")
```

```
def print_ship(self):
    for i in range(len(self)):
        if self[i].length > 150:
            print(f'Длина корабля №{i+1} больше 150 метров')
```