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. 3344 _____ Сербиновский Ю.М.

Преподаватель _____ Иванов Д.В.

Санкт-Петербург
2024

Цель работы

Исследовать и реализациовать АВЛ-дерево на языке программирования Python.

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)

Также для очной защиты необходимо подготовить визуализацию дерева. В отчете помимо проведенного исследования необходимо приложить код всей получившей структуры: класс узла и функции.

- Дано бинарное дерево поиска. Проверить является ли оно авл-деревом (т.е. для каждого узла разница высот левого и правого поддерева не больше 1). Реализуйте функцию check, которая возвращает булевое значение true, если дерево сбалансированное и false в противном случае.
- Дано бинарное дерево поиска. Реализуйте функцию diff, которая возвращает минимальную абсолютную разницу между значениями связанных узлов в этом дереве.
- Дано авл-дерево. Реализуйте функцию insert, которая на вход принимает корень дерева и значение которое нужно добавить в это дерево.

Выполнение работы

Описание классов:

- modules.btree.Node: Определяет узел в бинарном дереве.
- modules.avltree.Node: Определяет узел в АВЛ-дереве.

Описание функций:

- get_height: возвращает высоту узла (если узел None, то возвращает 0).
- get_balance: вычисляет балансирующий фактор узла (разница между высотой правого и левого поддеревьев).
- update_height: обновляет высоту указанного узла.
- left_rotation: выполняет левый поворот относительно узла.
- right_rotation: выполняет правый поворот относительно узла.
- insert: вставляет значение в АВЛ-дерево.
- find_min: находит узел с минимальным значением в АВЛ-дереве.
- find_max: находит узел с максимальным значением в АВЛ-дереве.
- find: ищет указанное значение в АВЛ-дереве.
- rebalance: проверяет и выполняет балансировку дерева, если есть различия в высоте ветвей.
- delete_min: удаляет узел с минимальным значением из АВЛ-дерева.
- delete_max: удаляет узел с максимальным значением из АВЛ-дерева.
- delete: удаляет узел с указанным значением из АВЛ-дерева.
- check: проверяет корректность АВЛ-дерева.
- diff: определяет минимальную разницу между связанными узлами в бинарном дереве.
- print_tree: отображает визуальное представление дерева в консоли.

Тестирование

Для тестирования операций с АВЛ-деревом были разработаны юнит-тесты. Запуск тестов выполняется с использованием pytest. Юнит-тестирование успешно пройдено. Сами тесты находятся в файле «tests_sample.py» (см. Приложение А). Результаты выполнения pytest представлены ниже:

```
========================= test session starts ==========================
platform linux -- Python 3.10.12, pytest-8.2.2, pluggy-1.5.0
rootdir: /home/kisyamane/leti/alg-2024-3344/Serbinovskii_Yurii_lb3
plugins: anyio-4.4.0
collected 7 items

test_sample.py .......
[100%]

========================== 7 passed in 0.02s ===========================
```

Исследование

Было проведено исследование времени выполнения операций с АВЛ-деревом. Результаты исследования представлены в таблице 1, где все значения времени указаны в секундах.

Таблица 1 – Результаты исследования времени выполнения операций с АВЛ-деревом

Операция	Количество элементов в дереве				
	100	1000	10000	100000	1000000
insert	0.000008	0.000008	0.000011	0.000014	0.000020
delete	0.000008	0.000008	0.000010	0.000012	0.000018
delete_min	0.000005	0.000007	0.000009	0.000012	0.000014
delete_max	0.000006	0.000008	0.000009	0.000011	0.000013

На основе полученных данных видно, что зависимость между количеством элементов в дереве и временем выполнения операций имеет логарифмический характер, что соответствует теоретическим ожиданиям.

Выводы

Реализация АВЛ-дерева выполнена на языке программирования Python. Исследование операций взаимодействия с АВЛ-деревом успешно проведено.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

```
from modules.avltree import delete_max, delete_min, insert, delete
from modules.btree import print_tree

if __name__ == "__main__":
    root = None
    for i in range(0, 75, 3):
        root = insert(i, root)
    root = delete_max(root)
    root = delete_min(root)
    root = delete(root.val, root)
    print_tree(root)
```

Название файла: tests-sample.py

```
from modules.avltree import (
    insert,
    find_max,
    find_min,
    find,
    delete,
    delete_max,
    delete_min
)

def fill_tree():
    root = None

    for i in range(10, 500, 3):
        root = insert(i, root)
    root = insert(0, root)
    root = insert(777, root)
    root = insert(333, root)
    return root

def test_insert():
    root = fill_tree()
    root = insert(51, root)
    assert find(51, root) is not None

def test_find():
    root = fill_tree()
    assert find(0, root) is not None
    assert find(1000, root) is None

def test_find_min():
    root = fill_tree()
    node = find_min(root)
```

```
        assert node is not None and node.val == 0

    def test_find_max():
        root = fill_tree()
        node = find_max(root)

        assert node is not None and node.val == 777

    def test_delete():
        root = fill_tree()
        root = delete(333, root)

        assert root is not None and find(333, root) is None

    def test_delete_min():
        root = fill_tree()
        root = delete_min(root)

        assert root is not None and find(0, root) is None

    def test_delete_max():
        root = fill_tree()
        root = delete_max(root)

        assert root is not None and find(777, root) is None
```

Название файла: avltree.py

```
class Node:
    def __init__(self, val, left=None, right=None):
        self.val = val
        self.left: Node | None = left
        self.right: Node | None = right
        self.height: int = 1


def update_height(node: Node):
    if node:
        node.height=max(get_height(node.left),  get_height(node.right))
+ 1


def get_height(node: Node):
    return node.height if node else 0


def get_balance(node: Node) -> int:
    return get_height(node.left) - get_height(node.right)
```

```
def right_rotation(old_root: Node) -> Node:
    new_root = old_root.left
    t = new_root.right

    new_root.right = old_root
    old_root.left = t

    update_height(old_root)
    update_height(new_root)

    return new_root

def left_rotation(old_root: Node) -> Node:
    new_root = old_root.right
    t = new_root.left

    new_root.left = old_root
    old_root.right = t

    update_height(old_root)
    update_height(new_root)

    return new_root

def rebalance(node: Node) -> Node:
    balance = get_balance(node)
    if balance > 1:  # Left heavy
        if get_balance(node.left) < 0:  # Left-Right case
            node.left = left_rotation(node.left)
        return right_rotation(node)
    if balance < -1:  # Right heavy
        if get_balance(node.right) > 0:  # Right-Left case
            node.right = right_rotation(node.right)
        return left_rotation(node)
    return node

def insert(val, node: Node) -> Node:
    if not node:
```

```
        return Node(val)

    if val < node.val:
        node.left = insert(val, node.left)
    else:
        node.right = insert(val, node.right)

    update_height(node)
    return rebalance(node)

def find_min(node: Node) -> Node:
    while node.left:
        node = node.left
    return node

def find_max(node: Node) -> Node:
    while node.right:
        node = node.right
    return node

def find(val, node: Node) -> Node | None:
    if not node:
        return None

    if val == node.val:
        return node
    elif val < node.val:
        return find(val, node.left)
    else:
        return find(val, node.right)

def delete_min(node: Node) -> Node:
    if not node.left:
        return node.right
    node.left = delete_min(node.left)
    update_height(node)
    return rebalance(node)
```

```
def delete_max(node: Node) -> Node:
    if not node.right:
        return node.left
    node.right = delete_max(node.right)
    update_height(node)
    return rebalance(node)

def delete(val, node: Node) -> Node:
    if not node:
        return None

    if val < node.val:
        node.left = delete(val, node.left)
    elif val > node.val:
        node.right = delete(val, node.right)
    else:
        if not node.left:
            return node.right
        if not node.right:
            return node.left

        min_node = find_min(node.right)
        node.val = min_node.val
        node.right = delete(min_node.val, node.right)

    update_height(node)
    return rebalance(node)
```

Название файла: btree.py

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

def check(root: Node) -> bool:
    def height_and_balance(node):
        if not node:
            return 0, True

        left_height, left_balanced = height_and_balance(node.left)
        right_height, right_balanced = height_and_balance(node.right)
```

```
        current_height = 1 + max(left_height, right_height)

        current_balanced = abs(left_height - right_height) <= 1

        return current_height, left_balanced and right_balanced and
current_balanced

    _, is_balanced = height_and_balance(root)
    return is_balanced


def diff(root: Node) -> int:
    min_diff = float('inf')

    def find_min_diff(node):
        nonlocal min_diff
        if not node:
            return

        if node.left:
            min_diff = min(min_diff, abs(node.val - node.left.val))
            find_min_diff(node.left)

        if node.right:
            min_diff = min(min_diff, abs(node.val - node.right.val))
            find_min_diff(node.right)

    find_min_diff(root)

    return min_diff


def print_tree(node: Node, level=0, prefix="Root: "):
    if node is not None:
        print(" " * (level * 4) + prefix + f"({node.val})")

        if node.left or node.right:
            if node.left:
                print_tree(node.left, level + 1, prefix="L--- ")
            else:
                print(" " * ((level + 1) * 4) + "L--- (None)")

            if node.right:
                print_tree(node.right, level + 1, prefix="R--- ")
            else:
                print(" " * ((level + 1) * 4) + "R--- (None)")
```