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. 3342 Мохамед.М.Х.

Преподаватель Иванов Д.В.

Санкт-Петербург
2024

Цель работы

Целью данной лабораторной работы является разработка различных функций, которые взаимодействует с АВЛ-деревом: Реализовать функцию **check**,

которая возвращает булевое значение **true**, если дерево сбалансированное и **false** в противном случае. Реализовать функцию **diff** которая возвращает минимальную абсолютную разницу между значениями связанных узлов в этом дереве. Реализовать функцию insert, которая на вход принимает корень дерева и значение которое нужно добавить в это дерево.

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.

Выполнение работы

Разработан добавлен класс Node, который является элементом дерева, он содержит в себе значение, указатели на корни левого и правого поддеревьев, а так же свою высоту.

Метод push(item) добавляет элемент в голову стека, и так же при условии того, что в стеке находится не менее двух элементов начинает проверку инвариантов, и их объединение при необходимости.

Функция get_height(node: Node) Возвращает высоту узла. Если узел равен None, возвращает 0.

Функция get_balance(node: Node) Возвращает None, если поданный узел не существует, иначе возвращает баланс узла, который определяется как разность высот левого и правого поддеревьев.

Функция update_height(node: Node) Обновляет высоту узла на основе высот его левого и правого поддеревьев. Если узел не существует, ничего не делает.

Функция rotate_right(y: Node) Выполняет правый поворот вокруг узла y. Обновляет высоты узлов y и x после поворота. Необходим для перебалансировки дерева после удаления узла или его добавления.

Функция rotate_left(x: Node) Выполняет левый поворот вокруг узла y. Обновляет высоты узлов y и x после поворота. Необходим для перебалансировки дерева после удаления узла или его добавления.

Функция diff(root: Node) Вычисляет минимальную разницу между значениями узлов в дереве. Использует обход дерева в порядке in-order для нахождения всех пар соседних элементов и вычисления их разницы.

Функция insert(val, node: Node) Вставляет значение val в дерево, начиная с узла node. После вставки обновляет высоты и балансирует дерево, если это необходимо.

Функция delete(val, node: Node) Удаляет значение val из дерева, начиная с узла node. После удаления обновляет высоты и балансирует дерево, если это необходимо.

Функция get_min_value_node(node: Node) Возвращает узел с минимальным значением в поддереве, начиная с узла node.

Функция delete_min(node: Node) Удаляет узел с минимальным значением в поддереве, начиная с узла node. После удаления обновляет высоты и балансирует дерево, если это необходимо.

Функция get_max_value_node(node: Node) Возвращает узел с максимальным значением в поддереве, начиная с узла node.

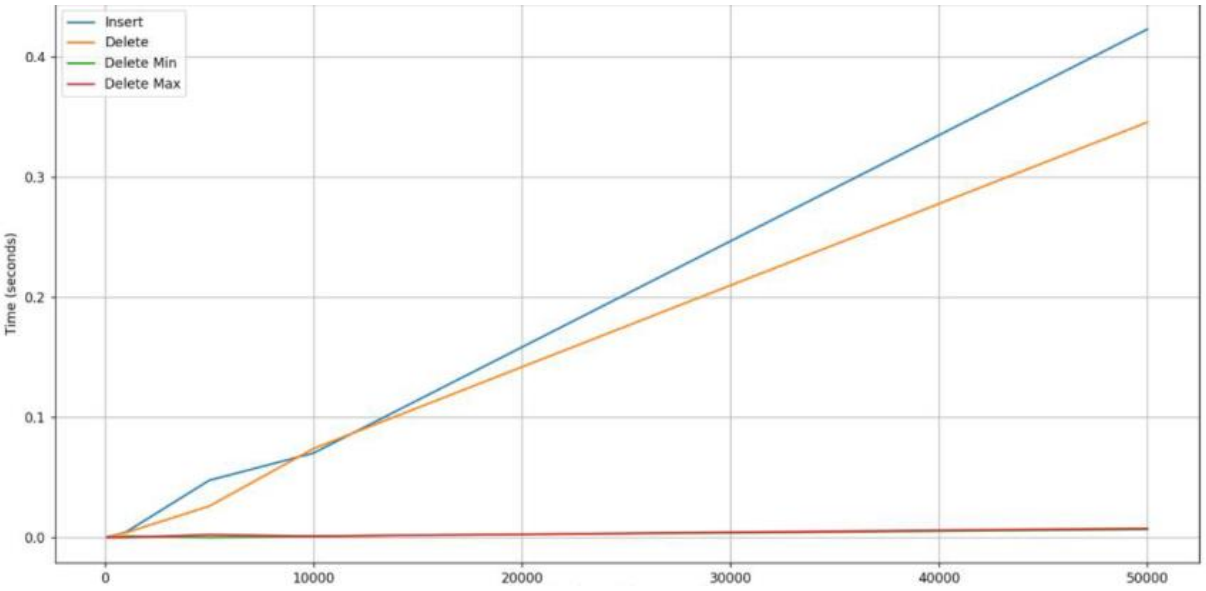
Функция delete_max(node: Node) Удаляет узел с максимальным значением в поддереве, начиная с узла node. После удаления обновляет высоты и балансирует дерево, если это необходимо.

Функция convert_to_anytree(node: Node) Преобразует дерево, представленное узлом node, в структуру данных AnyTreeNode, которая может быть использована для визуализации дерева.

Функция print_tree(root) Выводит дерево, начиная с корня root, в виде иерархической структуры. Использует функцию convert_to_anytree для преобразования дерева в формат, подходящий для визуализации Разработанный программный код см. в приложении А.

Тестирование

Тесты для проверки корректности работы функций по работе с деревом представлены в файле tests.py



Выводы

В ходе исследования были реализованы и проанализированы функции для работы с АВЛ-деревом. Вставка и удаление элементов имеют постоянную сложность O(log n), поскольку каждый узел добавляется или удаляется за фиксированное время, а последующая перебалансировка также занимает постоянное время. Однако удаление минимального и максимального значений выполняется очень быстро. Это объясняется тем, что АВЛ-дерево, будучи бинарным деревом поиска, имеет наименьший элемент в самом нижнем левом узле, а наибольший – в самом нижнем правом узле.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

```
 from typing import Union
from anytree import Node as AnyTreeNode, RenderTree

class Node:
    def __init__(self, val, left: Union[Node, None] = None, right:
Union[Node, None] = None):
        self.val = val
        self.left = left
        self.right = right
        self.height = 1

def get_height(node: Union[Node, None]) -> int:
    if not node:
        return 0
    return node.height

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

def get_balance(node: Union[Node, None]) -> int:
    if not node:
        return 0
    return get_height(node.left) - get_height(node.right)

def rotate_right(y: Node) -> Node:
    x = y.left
    T2 = x.right
    x.right = y
    y.left = T2
    update_height(y)
    update_height(x)
    return x

def rotate_left(x: Node) -> Node:
    y = x.right
    T2 = y.left
    y.left = x
    x.right = T2
    update_height(x)
    update_height(y)
    return y

def insert(val, node: Union[Node, None]) -> Node:
    if not node:
```

```
        return Node(val)

    if val < node.val:
        node.left = insert(val, node.left)
    elif val > node.val:
        node.right = insert(val, node.right)
    else:
        # Ignore duplicate values
        return node

    update_height(node)

    balance = get_balance(node)

    if balance > 1 and val < node.left.val:
        return rotate_right(node)

    if balance < -1 and val > node.right.val:
        return rotate_left(node)

    if balance > 1 and val > node.left.val:
        node.left = rotate_left(node.left)
        return rotate_right(node)

    if balance < -1 and val < node.right.val:
        node.right = rotate_right(node.right)
        return rotate_left(node)

    return node

def get_min_value_node(node: Node) -> Node:
    current = node
    while current.left is not None:
        current = current.left
    return current

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

    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
        elif not node.right:
            return node.left
```

```
        temp = get_min_value_node(node.right)
        node.val = temp.val
        node.right = delete(temp.val, node.right)

    update_height(node)

    balance = get_balance(node)

    if balance > 1 and get_balance(node.left) >= 0:
        return rotate_right(node)

    if balance > 1 and get_balance(node.left) < 0:
        node.left = rotate_left(node.left)
        return rotate_right(node)

    if balance < -1 and get_balance(node.right) <= 0:
        return rotate_left(node)

    if balance < -1 and get_balance(node.right) > 0:
        node.right = rotate_right(node.right)
        return rotate_left(node)

    return node

def delete_min(node: Union[Node, None]) -> Union[Node, None]:
    if node is None:
        return node

    if node.left is None:
        return node.right

    node.left = delete_min(node.left)

    update_height(node)

    balance = get_balance(node)

    if balance > 1 and get_balance(node.left) >= 0:
        return rotate_right(node)

    if balance > 1 and get_balance(node.left) < 0:
        node.left = rotate_left(node.left)
        return rotate_right(node)

    if balance < -1 and get_balance(node.right) <= 0:
        return rotate_left(node)

    if balance < -1 and get_balance(node.right) > 0:
```

```
        node.right = rotate_right(node.right)
        return rotate_left(node)

    return node

def delete_max(node: Union[Node, None]) -> Union[Node, None]:
    if node is None:
        return node

    if node.right is None:
        return node.left

    node.right = delete_max(node.right)

    update_height(node)

    balance = get_balance(node)

    if balance > 1 and get_balance(node.left) >= 0:
        return rotate_right(node)

    if balance > 1 and get_balance(node.left) < 0:
        node.left = rotate_left(node.left)
        return rotate_right(node)

    if balance < -1 and get_balance(node.right) <= 0:
        return rotate_left(node)

    if balance < -1 and get_balance(node.right) > 0:
        node.right = rotate_right(node.right)
        return rotate_left(node)

    return node

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

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

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

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

```
    dfs(root)
    return min_diff

def convert_to_anytree(node: Union[Node, None]) -> Union[AnyTreeNode, None]:
    if node is None:
        return None

    anytree_node = AnyTreeNode(str(node.val))

    if node.left:
        anytree_node.children += (convert_to_anytree(node.left),)
    if node.right:
        anytree_node.children += (convert_to_anytree(node.right),)

    return anytree_node

def print_tree(root):
    anytree_root = convert_to_anytree(root)
    for pre, fill, node in RenderTree(anytree_root):
        print(f"{pre}{node.name}")

root = None
values = [10, 20, 30, 40, 50, 25, 30, 40, 100, 23, 11]

for value in values:
    root = insert(value, root)

print("Original tree:")
print_tree(root)

root = delete(20, root)
print("\nTree after deleting node with value 20:")
print_tree(root)

root = delete_min(root)
print("\nTree after deleting minimum node:")
print_tree(root)

root = delete_max(root)
print("\nTree after deleting maximum node:")
print_tree(root)

print(f"\nMinimum difference between node values: {diff(root)}")
```

Название файла: tests.py

```
from main import Node, insert, delete, delete_min,
delete_max
```

```python
def test_insert():
    root = None
    values = [10, 20, 30, 40, 50, 25]
    for value in values:
        root = insert(value, root)

    assert root.val == 30
    assert root.left.val == 20
    assert root.right.val == 40
    assert root.left.left.val == 10
    assert root.left.right.val == 25
    assert root.right.right.val == 50

    print("Test insert passed!")

def test_delete():
    root = None
    values = [10, 20, 30, 40, 50, 25]
    for value in values:
        root = insert(value, root)

    root = delete(20, root)
    assert root.val == 30
    assert root.left.val == 25
    assert root.left.left.val == 10

    print("Test delete passed!")

def test_delete_min():
    root = None
    values = [10, 20, 30, 40, 50, 25]
    for value in values:
        root = insert(value, root)

    root = delete_min(root)
    assert root.val == 30
    assert root.left.val == 20
    assert root.left.left is None

    print("Test delete_min passed!")

def test_delete_max():
    root = None
    values = [10, 20, 30, 40, 50, 25]
    for value in values:
        root = insert(value, root)

    root = delete_max(root)
```

```
    assert root.val == 30
    assert root.right.val == 40
    assert root.right.right is None

    print("Test delete_max passed!")

test_insert()
test_delete()
test_delete_min()
test_delete_max()
```