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 _____ Песчатский С. Д.

Преподаватель _____ Иванов Д.В.

Санкт-Петербург
2024

Цель работы

Цель данной лабораторной работы — разработка функций для работы с АВЛ-деревом: реализовать функцию check, которая проверяет, является ли дерево сбалансированным, и возвращает true, если дерево сбалансировано, и false в противном случае; реализовать функцию diff, которая возвращает минимальную абсолютную разницу между значениями соседних узлов в дереве; реализовать функцию insert, которая принимает на вход корень дерева и значение, которое нужно добавить в 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 также свою высоту.

Функция check(root: Node) Проверяет является ли дерево АВЛ деревом или нет. Использует функцию is_balanced(node) для рекурсивной проверки каждого узла.

Функция get_tree_height(node) Рекурсивно ищет высоту дерева.

Функция get_max(node) Ищет максимальное значение в дереве

Функция get_min(node) Ищет минимальное значение в дереве

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 get_height(node: Node) Возвращает высоту узла. Если узел имеет тип не Node то 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 right_rotate(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 left_rotate(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 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 delete_max(node: Node) Удаляет узел с минимальным значением в поддереве, начиная с узла node. После удаления обновляет высоты и балансирует дерево, если это необходимо.

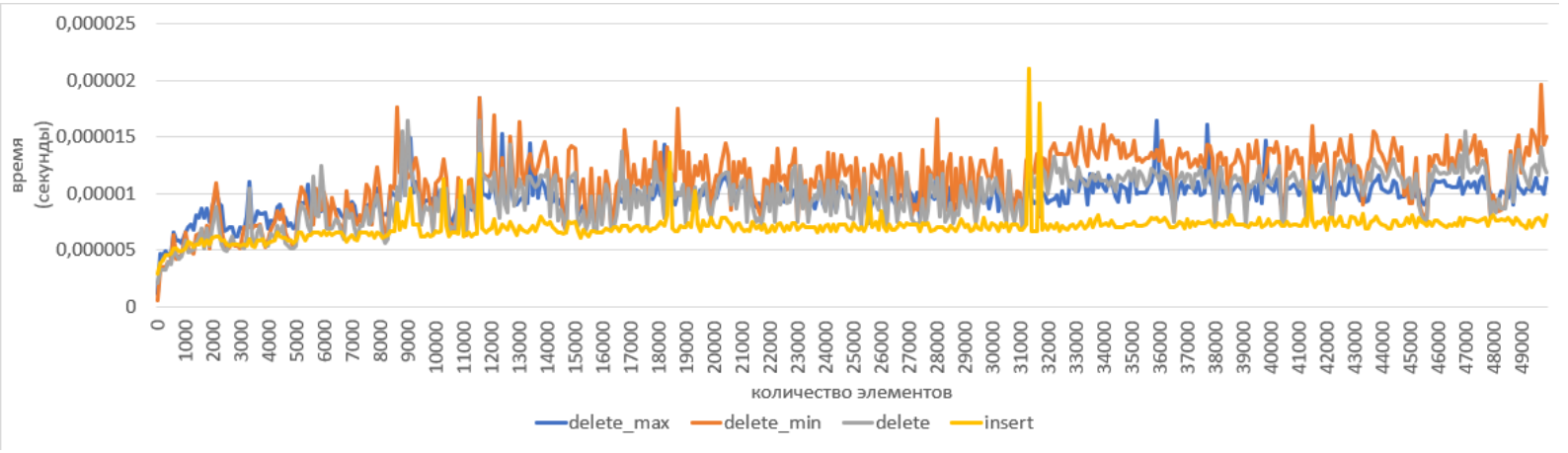
Функция delete_max(node: Node) Удаляет узел с максимальным значением в поддереве, начиная с узла node. После удаления обновляет высоты и балансирует дерево, если это необходимо.

Функция showTree(root) Выводит дерево, начиная с корня root, в виде иерархической структуры. Использует функцию build_tree_string для построения каждой строчки дерева, которые затем объединяются и выводятся.

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), так как каждый узел добавляется или удаляется за фиксированное время, а последующая перебалансировка также занимает одинаковое время. Удаление минимального и максимального значений выполняется очень быстро, поскольку АВЛ-дерево, будучи бинарным деревом поиска, позволяет быстро находить наименьший элемент (самый нижний левый) и наибольший элемент (самый нижний правый).

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

```
     from typing import Union
import time
import random
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: int = 1

def check(root: Node) -> bool:
    def is_balanced(node):
        if not node:
            return True, 0
        left_balanced, left_height = is_balanced(node.left)
        right_balanced, right_height = is_balanced(node.right)
        if not left_balanced or not right_balanced:
            return False, 0
        if abs(left_height - right_height) > 1:
            return False, 0
        return True, max(left_height, right_height) + 1
    balanced, _ = is_balanced(root)
    return balanced
def get_tree_height(node):
        if not node:
            return 0
        return 1 + max(get_tree_height(node.left),
get_tree_height(node.right))
def get_max(node):
    curr=node
    while curr.right:
        curr=curr.right
    return curr.val
def get_min(node):
```

```
    curr=node
    while curr.left:
        curr=curr.left
    return curr.val
def diff(root: Node) -> int:
    def in_order(diffs,node):
        if node.left is not None:
            diffs.append(abs(node.val-node.left.val))
            in_order(diffs,node.left)
        if node.right is not None:
            diffs.append(abs(node.val - node.right.val))
            in_order(diffs, node.right)
    diffs = []
    in_order(diffs,root)
    return min(diffs)
def get_height(node: Node) -> int:
    if not node:
        return 0
    return node.height

def get_balance(node: Node) -> int:
    if not node:
        return 0
    return get_height(node.left) - get_height(node.right)

def right_rotate(y: Node) -> Node:
    x = y.left
    T2 = x.right
    x.right = y
    y.left = T2
    y.height = max(get_height(y.left), get_height(y.right)) + 1
    x.height = max(get_height(x.left), get_height(x.right)) + 1
    return x

def left_rotate(x: Node) -> Node:
    y = x.right
    T2 = y.left
    y.left = x
    x.right = T2
```

```
    x.height = max(get_height(x.left), get_height(x.right)) + 1
    y.height = max(get_height(y.left), get_height(y.right)) + 1
    return y

def insert(val, node: Node) -> 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:
        return node
    node.height = 1 + max(get_height(node.left), get_height(node.right))
    balance = get_balance(node)
    if balance > 1 and val < node.left.val:
        return right_rotate(node)
    if balance < -1 and val > node.right.val:
        return left_rotate(node)
    if balance > 1 and val > node.left.val:
        node.left = left_rotate(node.left)
        return right_rotate(node)
    if balance < -1 and val < node.right.val:
        node.right = right_rotate(node.right)
        return left_rotate(node)
    return node

def delete(val, node: Node) -> Node:
    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 node.left is None:
            return node.right
        elif node.right is None:
            return node.left
```

```
        else:
            temp = node.right
            while temp.left:
                temp = temp.left
            node.val = temp.val
            node.right = delete(temp.val, node.right)
    node.height = 1 + max(get_height(node.left), get_height(node.right))
    balance = get_balance(node)
    if balance > 1 and get_balance(node.left) >= 0:
        return right_rotate(node)
    if balance > 1 and get_balance(node.left) < 0:
        node.left = left_rotate(node.left)
        return right_rotate(node)
    if balance < -1 and get_balance(node.right) <= 0:
        return left_rotate(node)
    if balance < -1 and get_balance(node.right) > 0:
        node.right = right_rotate(node.right)
        return left_rotate(node)

    return node
def delete_max(node: Node):
    target = get_max(node)
    delete(target, node)
def delete_min(node: Node):
    target = get_min(node)
    delete(target, node)
def showTree(node):
    def build_tree_string(node, curr_index):
        if node is None:
            return [], 0, 0, 0

        line1 = []
        line2 = []
        node_repr = str(node.val)

        new_root_width = gap_size = len(node_repr)

        l_box, l_box_width, l_root_start, l_root_end =
build_tree_string(node.left, 2 * curr_index + 1)
```

```
        r_box, r_box_width, r_root_start, r_root_end =
build_tree_string(node.right, 2 * curr_index + 2)

        if l_box_width > 0:
            l_root = (l_root_start + l_root_end) // 2 + 1
            line1.append(' ' * (l_root + 1))
            line1.append('_' * (l_box_width - l_root))
            line2.append(' ' * l_root + '/')
            line2.append(' ' * (l_box_width - l_root))
            new_root_start = l_box_width + 1
            gap_size += 1
        else:
            new_root_start = 0

        line1.append(node_repr)
        line2.append(' ' * new_root_width)

        if r_box_width > 0:
            r_root = (r_root_start + r_root_end) // 2
            line1.append('_' * r_root)
            line1.append(' ' * (r_box_width - r_root + 1))
            line2.append(' ' * r_root + '\\')
            line2.append(' ' * (r_box_width - r_root))
            gap_size += 1
        new_root_end = new_root_start + new_root_width - 1

        gap = ' ' * gap_size
        new_box = [''.join(line1), ''.join(line2)]
        for i in range(max(len(l_box), len(r_box))):
            l_line = l_box[i] if i < len(l_box) else ' ' * l_box_width
            r_line = r_box[i] if i < len(r_box) else ' ' * r_box_width
            new_box.append(l_line + gap + r_line)

        return new_box, len(new_box[0]), new_root_start, new_root_end

    tree_height = get_tree_height(node)
    if tree_height == 0:
        return
```

```
    tree_string, _, _, _ = build_tree_string(node, 0)
    for line in tree_string:
        print(line)

n=20
c=1
arr=[]
for i in range(n+1):
    arr.append(c)
    c+=1
root = Node(arr[0])
for i in range(1, n):
    root=insert(arr[i], root)
print("Исходное дерево:")
showTree(root)

delete(10, root)
print("\nДерево после удаления узла 10")
showTree(root)

delete_min(root)
print("\nДерево после удаления минимального узла")
showTree(root)

delete_max(root)
print("\nДерево после удаления максимального узла")
showTree(root)

insert(23, root)
print("\nДерево после добавления узла 23")
showTree(root)
```

Название файла: tests.py

```
from main import Node, insert, delete, delete_min, delete_max
def run_tests():
    test_insert()
    test_delete()
```

```
    test_delete_min()
    test_delete_max()
def test_insert():
    root = None
    values = [1, 2, 3, 4, 5, 6]
    for value in values:
        root = insert(value, root)

    assert root.val == 4
    assert root.left.val == 2
    assert root.right.val == 5
    assert root.left.left.val == 1
    assert root.left.right.val == 3
    assert root.right.right.val == 6

    print("Test insert passed!")

def test_delete():
    root = None
    values = [1, 2, 3, 4, 5, 6]
    for value in values:
        root = insert(value, root)

    root = delete(2, root)
    assert root.val == 4
    assert root.left.val == 3
    assert root.left.left.val == 1
    assert root.left.right is None

    print("Test delete passed!")

def test_delete_min():
    root = None
    values = [1, 2, 3, 4, 5, 6]
    for value in values:
        root = insert(value, root)
    root = delete_min(root)
    assert root.val == 4
    assert root.left.val == 2
```

```
    assert root.left.left is None
    assert root.left.right.val == 3

    print("Test delete_min passed!")

def test_delete_max():
    root = None
    values = [1, 2, 3, 4, 5, 6]
    for value in values:
        root = insert(value, root)

    root = delete_max(root)
    assert root.val == 4
    assert root.right.val == 5
    assert root.right.right is None

    print("Test delete_max passed!")


run_tests()
```