**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.

|  |  |  |
| --- | --- | --- |
| Студентка гр. 3342 |  | Корниенко А.Е. |
| Преподаватель |  | Иванов Д.В. |

Санкт-Петербург

2024

## Цель работы

Изучить разновидности деревьев и их особенности. С помощью языка программирования Python написать программу, которая создает АВЛ-деревья и может проводить с ними различные операции: балансировка, удаление и добавление элементов.

## 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

1) Создание класса узла дерева

Опишем класс Node и через конструктор определим следующие поля.

val – значение, лежащее в данном узле

left – ссылка на левый узел

right – ссылка на правый узел

height – высота узла.

2) Функции для АВЛ дерева

GetHeight(node) - получение высоты узла. Если передается None, то возвращается 0.

check(root) – проверка на то, является ли дерево АВЛ. Если во время обхода дерева находятся два поддерева высота, которых больше 1, возвращает False, иначе True.

right\_rotate(node) – правый поворот дерева. Реализован для балансировки дерева.

left\_rotate(node) – левый поворот дерева. Реализован для балансировки дерева.

insert(val, node) – вставка в дерево значения. Сравниваем данное значение со значением узла и рекурсиями ищем нужный элемент. Проводим вставку и балансировку.

delete\_node(val, node) – Сначала рекурсивно находится нужный узел. Если у него нет правого или левого поддерева, он заменяется на корень существующего. Если оба дерева присутствуют, находится минимальный элемент. Его значение записывается в узел, который нужно удалить, а сам минимальный элемент удаляется. Производится балансировка.

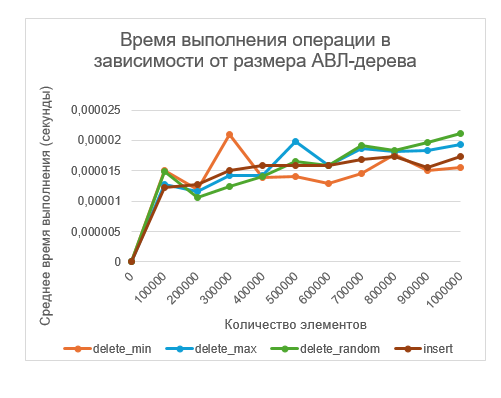
delete\_max(node) – удаление максимального элемента. Проходим рекурсией до самого правого элемента. Удаляем его. Проводим балансировку.

delete\_min(node) – удаление максимального элемента. Проходим рекурсией до самого левого элемента. Удаляем его. Проводим балансировку.

diff(root) – нахождение минимальной разницы между связанными узлами. Рекурсией проходим по всем узлам. Добавляем в массив разницу в значениях с левой и правой частями. Из массива берем минимальное значение.

## Тестирование

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



## Рисунок 1 – график зависимости времени выполнения операций от размера АВЛ-дерева

## Выводы

В ходе выполнения лабораторной работы была изучена и реализована структура данных «АВЛ-дерево». Был разработан необходимый функционал: вставка, удаление, балансировка: малый левый, малый правый, большой левый, большой правый. Данные функции были проверены на эффективность и работоспособность.

# Приложение А Исходный код программы

Название файла: main.py

class Node:

def \_\_init\_\_(self, val, left=None, right=None):

self.val = val

self.left = left

self.right = right

self.height = 1

def check(root):

if root.left is None and root.right is None:

return True

if root.left is None and (root.right.right is not None or root.right.left is not None):

return False

if root.right is None and (root.left.right is not None or root.left.left is not None):

return False

left = root.left

while left is not None:

if not check(left):

return False

left = left.left

right = root.right

while right is not None:

if not check(right):

return False

right = right.right

return True

def diff(root):

def passage\_of\_nodes(values, node):

if node.right is not None:

values.append(abs(node.val - node.right.val))

passage\_of\_nodes(values, node.right)

if node.left is not None:

values.append(abs(node.val - node.left.val))

passage\_of\_nodes(values, node.left)

values = []

passage\_of\_nodes(values, root)

return min(values)

def GetHeight(node):

if node is None:

return 0

return node.height

def update\_height(node):

if node:

node.height = 1 + max(GetHeight(node.left), GetHeight(node.right))

def right\_rotate(node):

left\_child = node.left

right\_child\_left = left\_child.right

left\_child.right = node

node.left = right\_child\_left

update\_height(node)

update\_height(left\_child)

return left\_child

def left\_rotate(node):

right\_child = node.right

left\_child\_right = right\_child.left

right\_child.left = node

node.right = left\_child\_right

update\_height(node)

update\_height(right\_child)

return right\_child

def insert(val, node: Node) -> Node:

if node is None:

return Node(val)

elif val < node.val:

node.left = insert(val, node.left)

else:

node.right = insert(val, node.right)

update\_height(node)

height\_difference = GetHeight(node.left) - GetHeight(node.right)

# малый левый поворот

if height\_difference <= -2 and GetHeight(node.right.left) <= GetHeight(node.right.right):

return left\_rotate(node)

# малый правый поворот

if height\_difference >= 2 and GetHeight(node.left.right) <= GetHeight(node.left.left):

return right\_rotate(node)

# большой левый поворот

if height\_difference <= -2 and GetHeight(node.right.left) > GetHeight(node.right.right):

node.right = right\_rotate(node.right)

return left\_rotate(node)

# большой правый поворот

if height\_difference >= 2 and GetHeight(node.left.right) > GetHeight(node.left.left):

node.left = left\_rotate(node.left)

return right\_rotate(node)

return node

def delete\_node(val, node):

if node is None:

return node

if val < node.val:

node.left = delete\_node(val, node.left)

elif val > node.val:

node.right = delete\_node(val, node.right)

else:

if node.right is None:

temp = node.left

return temp

if node.left is None:

temp = node.right

return temp

temp = node.right

while temp.left is not None:

temp = temp.left

node.val = temp.val

node.right = delete\_node(temp.val, node.right)

update\_height(node)

height\_difference = GetHeight(node.left) - GetHeight(node.right)

# малый левый поворот

if height\_difference <= -2 and GetHeight(node.right.left) <= GetHeight(node.right.right):

return left\_rotate(node)

# малый правый поворот

if height\_difference >= 2 and GetHeight(node.left.right) <= GetHeight(node.left.left):

return right\_rotate(node)

# большой левый поворот

if height\_difference <= -2 and GetHeight(node.right.left) > GetHeight(node.right.right):

node.right = right\_rotate(node.right)

return left\_rotate(node)

# большой правый поворот

if height\_difference >= 2 and GetHeight(node.left.right) > GetHeight(node.left.left):

node.left = left\_rotate(node.left)

return right\_rotate(node)

return node

def delete\_min(node):

if node.left is None:

return None

node.left = delete\_min(node.left)

update\_height(node)

height\_difference = GetHeight(node.left) - GetHeight(node.right)

# малый левый поворот

if height\_difference <= -2 and GetHeight(node.right.left) <= GetHeight(node.right.right):

return left\_rotate(node)

# малый правый поворот

if height\_difference >= 2 and GetHeight(node.left.right) <= GetHeight(node.left.left):

return right\_rotate(node)

# большой левый поворот

if height\_difference <= -2 and GetHeight(node.right.left) > GetHeight(node.right.right):

node.right = right\_rotate(node.right)

return left\_rotate(node)

# большой правый поворот

if height\_difference >= 2 and GetHeight(node.left.right) > GetHeight(node.left.left):

node.left = left\_rotate(node.left)

return right\_rotate(node)

return node

def delete\_max(node):

if node.right is None:

return None

node.right = delete\_max(node.right)

update\_height(node)

height\_difference = GetHeight(node.left) - GetHeight(node.right)

# малый левый поворот

if height\_difference <= -2 and GetHeight(node.right.left) <= GetHeight(node.right.right):

return left\_rotate(node)

# малый правый поворот

if height\_difference >= 2 and GetHeight(node.left.right) <= GetHeight(node.left.left):

return right\_rotate(node)

# большой левый поворот

if height\_difference <= -2 and GetHeight(node.right.left) > GetHeight(node.right.right):

node.right = right\_rotate(node.right)

return left\_rotate(node)

# большой правый поворот

if height\_difference >= 2 and GetHeight(node.left.right) > GetHeight(node.left.left):

node.left = left\_rotate(node.left)

return right\_rotate(node)

return node

Название файла: tests.py

from main import \*

import pytest

def test\_insert():

root = None

values = [30, 40, 50, 25, 35, 60]

for value in values:

root = insert(value, root)

assert root.val == 40

assert root.left.val == 30

assert root.right.val == 50

assert root.right.right.val == 60

assert root.left.right.val == 35

assert root.left.left.val == 25

def test\_delete():

root = None

values = [30, 40, 50, 25, 35, 60]

for value in values:

root = insert(value, root)

root = delete\_node(30, root)

assert root.val == 40

assert root.left.val == 35

assert root.left.left.val == 25

def test\_delete\_min():

root = None

values = [30, 40, 50, 25, 35, 60]

for value in values:

root = insert(value, root)

root = delete\_min(root)

assert root.val == 40

assert root.left.val == 30

assert root.left.left is None

def test\_delete\_max():

root = None

values = [30, 40, 50, 25, 35, 60]

for value in values:

root = insert(value, root)

root = delete\_max(root)

assert root.val == 40

assert root.right.val == 50

assert root.right.right is None