# 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.

*Сheck(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 = 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}' | result == my\_answer | Ответ верный |

**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')