**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. 3384 |  | Пьянков М.Ф. |
| Преподаватель |  | Шестопалов Р.П. |

Санкт-Петербург

**2024**

## Цель работы

Изучение и реализация структуры данных: АВЛ-дерево, а также исследование время работы различных методов этой СД. Выполнение задания.

## Задание

Дано авл-дерево. Реализуйте функцию insert, которая на вход принимает корень дерева и значение которое нужно добавить в это дерево.

Ограничения:

2 <= N <= 1000

-1000 <= значения узлов <= 1000

сигнатура функции insert на python:

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

Определение класса Node:

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

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

Все методы выбранной структуры данных будут реализованы на основе циклических алгоритмов, а не рекурсивных в целях уменьшения затрат памяти и повышении стабильности работы.

Метод \_\_get\_height\_\_(node) – статический метод, необходимый для корректного получения высоты узла.

Метод \_\_height\_update\_\_(self, node) — метод, обновляющий веса данного узла.

Методы поворотов:

\_\_small\_left\_rotate\_\_(self, node) — малы левый поворот

\_\_small\_right\_rotate\_\_(self, node) — малый правый поворот

\_\_large\_left\_rotate\_\_(self, node) — большой левый поворот

\_\_large\_right\_rotate\_\_(self, node) — большой правый поворот

Методы больших поворотов были реализованы с помощью малых поворотов.

Метод \_\_balance\_factor\_\_(self, node) - метод позволяющий определить, нужно ли начинать балансировку в данном узле.

Метод \_\_make\_rotations\_\_(self, node) - метод вызывающий методы вращения на основе значений метода \_\_balance\_factor\_\_

Метод \_\_balance\_\_(self, node) - метод начинающий балансировку от текущего узла к корню дерева.

Метод search(self, value) — циклический поиск элемента в дереве.

Метод insert(self, value) — циклическая вставка нового элемента в дерево.

Метод pop(self, value) — циклическое удаление элемента по значению из дерева.

Методы pop\_min(self) и pop\_max(self) реализуют удаление минимального и максимального элемента дерева соответственно.

Метод check\_from\_node(root: AVLTreeNode) -> bool — статический метод, проверяющий дерево с корнем в данном узле на соответствие свойствам АВЛ-дерева.

Метод check\_global(self) — метод, запускающий check\_from\_node в корне дерева.

Метод diff\_from\_node(root: AVLTreeNode, minimal=1e10) -> int — статический метод, вычисляющий минимальную разницу между соседними элементами дерева, корнем которого является данный узел.

Метод diff\_global(self) — метод, вызывающий diff\_from\_node в корне дерева.

Метод in\_order(self, current) — метод, осуществляющий прямой обход дерева, необходим для тестирования и визуализации элементов дерева.

Метод create\_graph(self) — специальный метод, для визуализации дерева с помощью библиотеки graphviz.

Метод render\_avl\_tree(self) — метод, генерирующий изображение дерева.

Разработанный программный код см. в приложении А.

## Исследование

Проведём исследование скорости работы методов вставки, и методов удаления элементов из дерева.

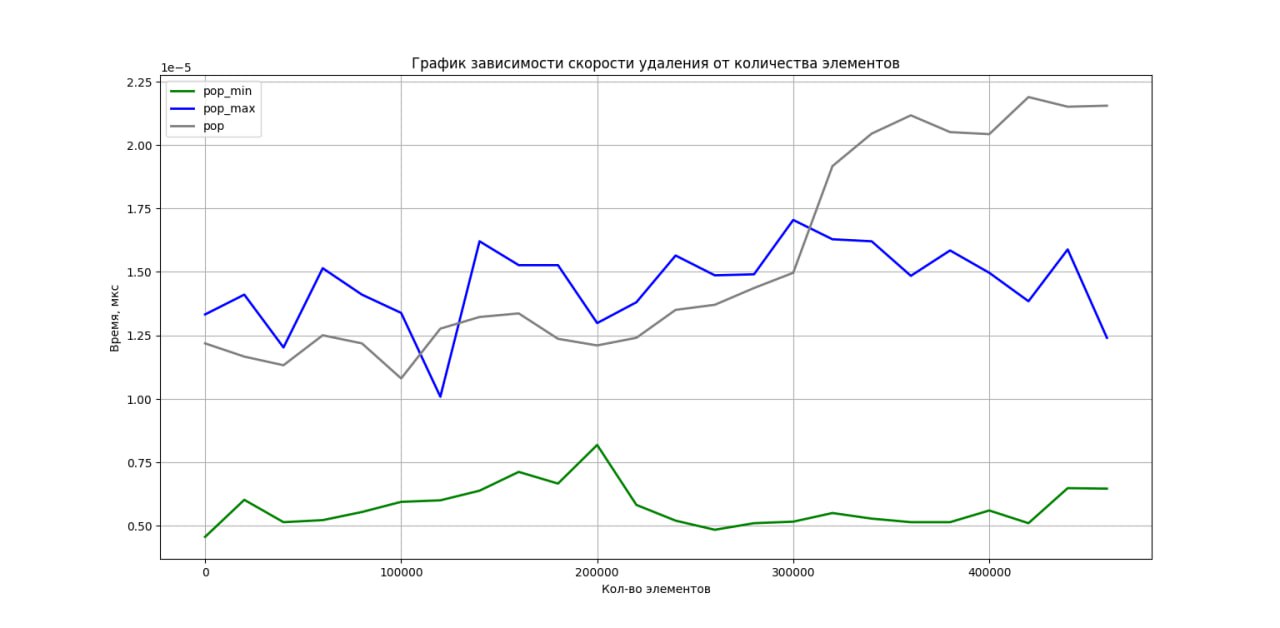
Рисунок 1 — Вставка

Рисунок 2 — Удаление

На графике скорости вставки (см рис. 1) видна сложность O(logn), на графике скорости удаления (см рис. 2)тоже видна сложность O(logn), однако не так очевидно. Следует отметить что трудности в измерении времени удаления элементов в дереве и сопоставлении этой зависимости какой-либо функции вызывает сложности, в следствии того, что после каждой итерации расположении элементов не определено из-за алгоритмов вращения. Но теоретические положения и реализация методов позволяют убедится в том что сложность достигает O(logn). Пояснение в случае удаления элемента необходимо каждый раз выбирать из двух детей узла , очевидно, что необходимо не более log2(n) операций. Однако трудности возникают в том, что после добавления или удаления возможно нужно провести балансировку, для этого от текущего элемента необходимо вернуться к корню и в случае надобности провести балансировку деревьев. Так как сложность балансировки O(1), то соответственно полное количество операций и время исполнения при вставке/удалении не может превышать 2log2(n). Но так как не представляется возможным протестировать АВЛ-дерево на больших данных на графиках наблюдаются аномалии.

## Выводы

Была успешно изучена и реализована структура данных АВЛ-дерево. Также было проведено тестирование и исследование, в результате которого были получены в целом результаты соответствующие теоретическим представлениям. Также была успешно выполнена поставленная задача.

# Приложение А Исходный код программы

Файл main\_lb3.py

def get\_height(node):

return node.height if node else 0

def update\_height(node):

node.height = max(get\_height(node.left), get\_height(node.right)) + 1

def balance\_factor(node):

return get\_height(node.right) - get\_height(node.left)

def small\_right\_rotate(node):

left = node.left

left\_right = left.right

left.right = node

node.left = left\_right

update\_height(node)

update\_height(left)

return left

def small\_left\_rotate(node):

right = node.right

right\_left = right.left

right.left = node

node.right = right\_left

update\_height(node)

update\_height(right)

return right

def \_\_balance\_\_(node):

b\_f = balance\_factor(node)

if b\_f == -2:

if balance\_factor(node.left) <= 0:

return small\_right\_rotate(node)

else:

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

return small\_right\_rotate(node)

elif b\_f == 2:

if balance\_factor(node.right) >= 0:

return small\_left\_rotate(node)

else:

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

return small\_left\_rotate(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)

elif val > node.val:

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

update\_height(node)

return \_\_balance\_\_(node)

Файл avl\_tree.py

import graphviz as gv

class AVLTreeNode:

def \_\_init\_\_(self, value):

self.parent = None

self.left = None

self.right = None

self.height = 1

self.value = value

class AVLTree:

def \_\_init\_\_(self):

self.root = None

@staticmethod

def \_\_get\_height\_\_(node):

return node.height if node else 0

def \_\_height\_update\_\_(self, node):

node.height = max(self.\_\_get\_height\_\_(node.left), self.\_\_get\_height\_\_(node.right)) + 1

def \_\_small\_left\_rotate\_\_(self, node):

temp = node.right

temp\_left = temp.left

temp.parent = node.parent

node.parent = temp

if temp\_left is not None:

temp\_left.parent = node

node.right = temp\_left

temp.left = node

if temp.parent is not None:

if temp.parent.right == node:

temp.parent.right = temp

elif temp.parent.left == node:

temp.parent.left = temp

else:

self.root = temp

self.\_\_height\_update\_\_(node)

self.\_\_height\_update\_\_(temp)

def \_\_small\_right\_rotate\_\_(self, node):

temp = node.left

temp\_right = temp.right

temp.parent = node.parent

node.parent = temp

if temp\_right is not None:

temp\_right.parent = node

node.left = temp\_right

temp.right = node

if temp.parent is not None:

if temp.parent.left == node:

temp.parent.left = temp

elif temp.parent.right == node:

temp.parent.right = temp

else:

self.root = temp

self.\_\_height\_update\_\_(node)

self.\_\_height\_update\_\_(temp)

def \_\_large\_left\_rotate\_\_(self, node):

self.\_\_small\_right\_rotate\_\_(node.right)

self.\_\_small\_left\_rotate\_\_(node)

def \_\_large\_right\_rotate\_\_(self, node):

self.\_\_small\_left\_rotate\_\_(node.left)

self.\_\_small\_right\_rotate\_\_(node)

def \_\_balance\_factor\_\_(self, node):

left = self.\_\_get\_height\_\_(node.left) if node else 0

right = self.\_\_get\_height\_\_(node.right) if node else 0

return left - right

def \_\_make\_rotations\_\_(self, node):

balance\_factor = self.\_\_balance\_factor\_\_(node)

if balance\_factor == -2:

if self.\_\_balance\_factor\_\_(node.right) > 0:

self.\_\_large\_left\_rotate\_\_(node)

else:

self.\_\_small\_left\_rotate\_\_(node)

elif balance\_factor == 2:

if self.\_\_balance\_factor\_\_(node.left) < 0:

self.\_\_large\_right\_rotate\_\_(node)

else:

self.\_\_small\_right\_rotate\_\_(node)

def \_\_balance\_\_(self, node):

current = node

next\_parent = current.parent

self.\_\_height\_update\_\_(current)

while next\_parent is not None:

current = next\_parent

next\_parent = current.parent

self.\_\_height\_update\_\_(current)

self.\_\_make\_rotations\_\_(current)

def search(self, value):

current = self.root

while current is not None:

if current.value > value:

current = current.left

elif current.value < value:

current = current.right

else:

return True

return False

def insert(self, value):

if self.root is None:

self.root = AVLTreeNode(value)

return

current = self.root

new\_node = AVLTreeNode(value)

while current is not None:

if value <= current.value:

if current.left is None:

new\_node.parent = current

current.left = new\_node

break

current = current.left

else:

if current.right is None:

new\_node.parent = current

current.right = new\_node

break

current = current.right

self.\_\_balance\_\_(current)

def pop(self, value):

if self.root is None:

return

current = self.root

while current is not None:

if current.value > value:

current = current.left

elif current.value < value:

current = current.right

else:

break

if current is None:

return

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

self.root = None

return

if current is self.root and self.root.right is None:

self.root.left.parent = None

self.root = self.root.left

return

if current.right is None:

if current.left is not None:

current.left.parent = current.parent

if current == current.parent.left:

current.parent.left = current.left

elif current == current.parent.right:

current.parent.right = current.left

else:

new\_current = current.right

while new\_current.left is not None:

new\_current = new\_current.left

current.value = new\_current.value

if new\_current.right is not None:

new\_current.right.parent = new\_current.parent

if new\_current.parent != current:

if new\_current.right is not None:

new\_current.right.parent = new\_current.parent

new\_current.parent.left = new\_current.right

elif new\_current.parent == current:

if current.right.right is not None:

current.right.right.parent = current

current.right = current.right.right

self.\_\_balance\_\_(new\_current)

return

self.\_\_balance\_\_(current)

def pop\_min(self):

if self.root is None:

return

current = self.root

while current.left is not None:

current = current.left

if current == self.root:

if current.right is None:

self.root = None

else:

current.right.parent = None

self.root = current.right

else:

if current.right is not None:

current.right.parent = current.parent

current.parent.left = current.right

self.\_\_balance\_\_(current)

def pop\_max(self):

if self.root is None:

return

current = self.root

while current.right is not None:

current = current.right

if current == self.root:

if current.left is None:

self.root = None

else:

current.left.parent = None

self.root = current.left

else:

if current.left is not None:

current.left.parent = current.parent

current.parent.right = current.left

self.\_\_balance\_\_(current)

@staticmethod

def check\_from\_node(root: AVLTreeNode) -> bool:

if root is None:

return True

left\_height = AVLTree.\_\_get\_height\_\_(root.left)

right\_height = AVLTree.\_\_get\_height\_\_(root.right)

if abs(left\_height - right\_height) <= 1 and \

AVLTree.check\_from\_node(root.left) and AVLTree.check\_from\_node(root.right):

return True

return False

def check\_global(self):

return self.check\_from\_node(self.root)

@staticmethod

def diff\_from\_node(root: AVLTreeNode, minimal=1e10) -> int:

if root is None:

return int(minimal)

if root.left is not None:

minimal = min(abs(root.value - root.left.value), minimal)

if root.right is not None:

minimal = min(abs(root.value - root.right.value), minimal)

return min(AVLTree.diff\_from\_node(root.left, minimal), AVLTree.diff\_from\_node(root.right, minimal), minimal)

def diff\_global(self):

return self.diff\_from\_node(self.root)

def in\_order(self, current):

if current is not None:

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

return current.value

if current.left is not None and current.right is not None:

return f'{self.in\_order(current.left)} {current.value} {self.in\_order(current.right)}'

if current.left is not None:

return f'{self.in\_order(current.left)} {current.value}'

if current.right is not None:

return f'{current.value} {self.in\_order(current.right)}'

def create\_graph(self):

dot = gv.Digraph(format='png')

nodes, edges = [], []

stack = [(self.root, "")] if self.root is not None else []

while stack:

node, label = stack.pop()

if node:

dot.node(str(id(node)), str(node.value))

nodes.append((id(node), node.value))

if node.left:

dot.edge(str(id(node)), str(id(node.left)))

edges.append((node.value, node.left.value))

stack.append((node.left, "L"))

if node.right:

dot.edge(str(id(node)), str(id(node.right)))

edges.append((node.value, node.right.value))

stack.append((node.right, "R"))

return dot, nodes, edges

def render\_avl\_tree(self):

dot, \_, \_ = self.create\_graph()

dot.render('avl\_tree.gv', view=True)