**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.

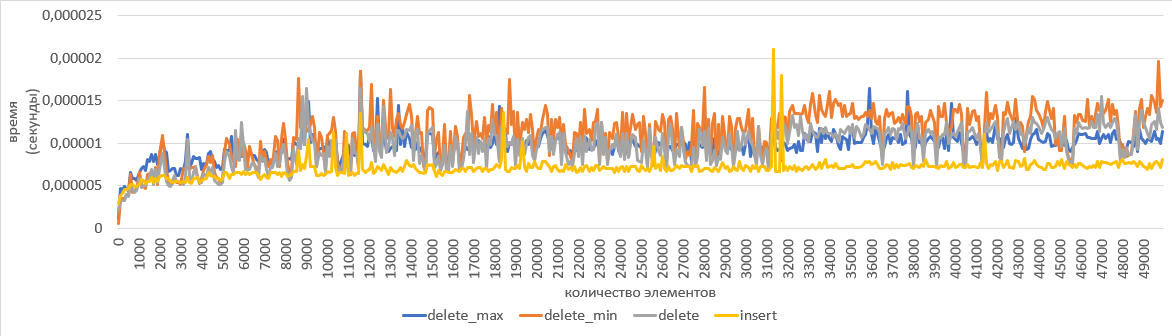
Функция delete\_max(node: Node) Удаляет узел с максимальным значением в поддереве, начиная с узла node. После удаления обновляет высоты и балансирует дерево, если это необходимо.

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()