**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: АВЛ-деревья.**

|  |  |  |
| --- | --- | --- |
| Студент гр. 3344 |  | Фоминых Е.Г. |
| Преподаватель |  | Иванов Д.В. |

Санкт-Петербург 2024

# Цель работы

Изучение теоретической информации об устройстве АВЛ-деревьев и работа с ними. Реализация АВЛ-дерева с функциями вставки, удаления данного, максимального и минимального элементов, анализ эффективности работы работы на различных объемах данных.

# 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

Класс Node – узел АВЛ-дерева, имеет следующие поля:

* + - **val** – значение узла.
    - **left** – левый дочерний узел.
    - **right** – правый дочерний узел.
    - **height** – высота текущего узла.

Были реализованы следующие методы:

* + - **get\_height**– позволяет узнать высоту данного узла. В случае ***None*** узла возвращает 0.
    - **get\_balance** – позволяет узнать сбалансирован ли узел
    - **get\_min\_value\_node** – позволяет найти узел с минимальным значением в поддереве.
    - **get\_max\_value\_node** – позволяет найти узел с максимальным значением в поддереве.
    - **left\_rotate, right\_rotate, left\_right\_rotate, right\_left\_rotate** – производит поворот в необходимую стороону;
    - **insert** – позволяет вставить элемент со значением ***val***. Автоматически обновляет высоту дерева и, при необходимости, производит балансировку дерева.
    - **delete** – позволяет удалить элемент со значением. Автоматически обновляет высоту дерева и, при необходимости, производит балансировку дерева.
    - **diff** – позволяет вычислить минимальную абсолютную разницу между значением корня и значениями всех узлов в поддереве.
    - **visualization** – позволяет отобразить дерево.

# Тестирование

Были проведены исследования для оценки эффективности реализованных функций. Исследования проводились на различных объемах данных (100, 10000 и 100000), проверялась вставка в дерево, основанное на возрастающей, случайной, убывающей последовательностях, а также удаление из дерева на случайной выборке. Результаты исследования представлены на рисунке 1, а их графическое представление – на рисунке 2.

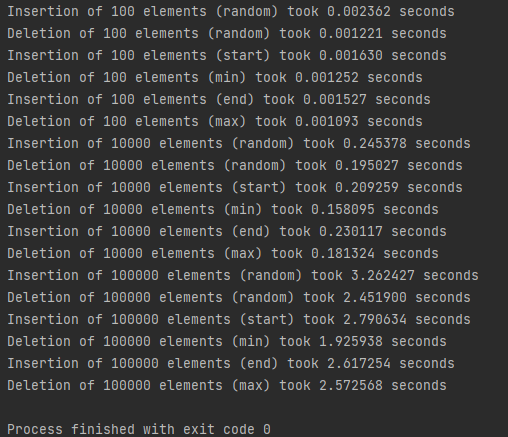


Рисунок 1.

# Тестирование.

Тесты для проверки корректности работы каждой функции находятся в файле tests.py. Они охватывают ключевые операции, включая вставку,

удаление, удаление максимального и минимального элементов, а также повороты дерева.

Результаты тестирования: Размер данных: 100

* Вставка:
  + Восходящая последовательность: 0.0016 секунд
  + Случайная последовательность: 0.0015секунд
  + Нисходящая последовательность: 0.0023 секунд
* Удаление:
  + Случайная последовательность: 0.0012 секунд

Вставка и удаление на небольших наборах данных выполняются практически мгновенно. Различия в производительности между различными типами

входных данных минимальны, что ожидаемо при малых объемах данных.

Размер данных: 10000

* Вставка:
  + Восходящая последовательность: 0.2092секунд
  + Случайная последовательность: 0.2453 секунд
  + Нисходящая последовательность: 0.2356 секунд
* Удаление:
  + Случайная последовательность 0.1950 секунд

При увеличении объема данных время выполнения операций возрастает, но остается в пределах логарифмической сложности (O(log n)), благодаря балансировке дерева. Для определенных наборов данных наблюдается незначительное увеличение времени из-за большего количества поворотов.

Размер данных: 100000

* Вставка:
  + Восходящая последовательность: 2.6172 секунд
  + Случайная последовательность: 3.2624 секунд
  + Нисходящая последовательность: 2.7906 секунд Удаление:
  + Случайная последовательность: 2.4519 секунд

Как можно заметить, с увеличением объема данных время вставки и удаления также возрастает. При сравнении вставки в деревья, основанные на отсортированных и неотсортированных данных, можно увидеть, что в случае неотсортированных данных вставка занимает больше времени. Это можно объяснить необходимостью часто выполнять повороты и, соответственно, балансировку дерева. Результаты исследования согласуются с теоретической сложностью O(log n), что позволяет сделать вывод о том, что логика работы АВЛ-дерева обеспечивает эффективную обработку как малых, так и больших объемов данн

# Выводы

Была изучена основная теоретическая информация об устройстве АВЛ- деревьев, было реализовано собственное АВЛ-дерево с функциями вставки, удаления данного, максимального и минимального элементов, была проведена теоретическая и эксперементальная оценка сложности вставки и удаления элементов на различных объемах данных.

**ПРИЛОЖЕНИЕ А ИСХОДНЫЙ КОД ПРОГРАММЫ**

# Название файла: main.py

# from insert import \*

# from delete import \*

# from visualization import visualization\_tree

# def main():

# root = None

# values = [i for i in range(1, 15)]

# for val in values:

# root = insert(val, root)

# delete(1, root)

# delete(5, root)

# delete(15, root)

# visualization\_tree(root)

# if \_\_name\_\_ == "\_\_main\_\_":

# main()

# Название файла: modules/balance.py

# from node import Node

# from height import get\_height

# def get\_balance(node: Node) -> int:

# if not node:

# return 0

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

# Название файла: modules/check.py

from node import Node

def tree\_height(root: Node) -> tuple:

if root == None:

return 0, True

left\_height, is\_left\_balanced = tree\_height(root.left)

right\_height, is\_right\_balanced = tree\_height(root.right)

current\_height = 1 + max(left\_height, right\_height)

is\_current\_balanced = is\_left\_balanced and is\_right\_balanced and abs(left\_height - right\_height) <= 1

return current\_height, is\_current\_balanced

def check(root: Node) -> bool:

return tree\_height(root)[1]

**modules/delete.py**

from node import Node

from height import get\_height

from balance import get\_balance

from rotate import left\_rotate

from rotate import right\_rotate

from rotate import left\_right\_rotate

from rotate import right\_left\_rotate

from get\_value\_node import get\_min\_value\_node

def delete(value: int, node: Node) -> Node:

if node is None:

return node

if value < node.val:

node.left = delete(value, node.left)

elif value > node.val:

node.right = delete(value, node.right)

else:

if node.left is None: return node.right

elif node.right is None: return node.left

temp = get\_min\_value\_node(node.right)

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:

if get\_balance(node.left) >= 0: return right\_rotate(node)

else: return left\_right\_rotate(node)

if balance < -1 and get\_balance(node.right) <= 0:

if get\_balance(node.right) <= 0: return left\_rotate(node)

else: return right\_left\_rotate(node)

return node

**modules/diff.py**

from node import Node

def diff(node:Node):

if not node:

return 0

res = float("inf")

if node.left:

res = diff(node.left)

res = min(res, abs(node.val - node.left.val))

if node.right:

res = min(diff(node.right), res)

res = min(res, abs(node.val - node.right.val))

return res

**modules/get\_value\_node.py**

from node import Node

def get\_max\_value\_node(node: Node) -> Node:

current = node

while current.right:

current = current.right

return current

def get\_min\_value\_node(node: Node) -> Node:

current = node

while current.left:

current = current.left

return current

**modules/heeight.py**

from node import Node

def get\_height(node: Node) -> int:

if not node:

return 0

return node.height

**modules/insert.py**

from node import Node

from height import get\_height

from balance import get\_balance

from rotate import left\_rotate

from rotate import right\_rotate

from rotate import left\_right\_rotate

from rotate import right\_left\_rotate

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

if not node:

return Node(val)

if val < node.val:

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

else:

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

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

balance = get\_balance(node)

if balance > 1:

if val > node.left.val: return left\_right\_rotate(node)

if val < node.left.val: return right\_rotate(node)

if balance < -1:

if val > node.right.val: return left\_rotate(node)

if val < node.right.val: return right\_left\_rotate(node)

return node

**modules/node.py**

from typing import Union

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

**modules/rotate.py**

from node import Node

from height import get\_height

def left\_rotate(x: Node) -> Node:

y = x.right

temp = y.left

y.left = x

x.right = temp

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

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

return y

def right\_rotate(y: Node) -> Node:

x = y.left

temp = x.right

x.right = y

y.left = temp

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

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

return x

def right\_left\_rotate(node: Node) -> Node:

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

return left\_rotate(node)

def left\_right\_rotate(node: Node) -> Node:

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

return right\_rotate(node)

**modules/vizualization.py**

from node import Node

def visualization\_tree(node: Node, level=0, prefix=""):

if node is not None:

visualization\_tree(node.right, level + 1, "R--- ")

print(" " \* (level \* 4) + prefix + str(node.val))

visualization\_tree(node.left, level + 1, "L--- "

**modules/tests.py**

from node import Node

from insert import insert

from delete import delete

from balance import get\_balance

from height import get\_height

from get\_value\_node import get\_min\_value\_node

from get\_value\_node import get\_max\_value\_node

def test\_insert\_root():

root = insert(10, None)

assert root.val == 10

assert root.height == 1

def test\_insert\_right\_child():

root = insert(10, None)

root = insert(20, root)

assert root.val == 10

assert root.right.val == 20

assert root.height == 2

def test\_insert\_left\_child():

root = insert(20, None)

root = insert(10, root)

assert root.val == 20

assert root.left.val == 10

assert root.height == 2

def test\_balance\_after\_inserts():

root = None

for val in [10, 20, 30]:

root = insert(val, root)

assert root.val == 20

assert root.left.val == 10

assert root.right.val == 30

assert get\_balance(root) <= 1

def test\_delete\_leaf():

root = None

for val in [10, 20, 30]:

root = insert(val, root)

root = delete(30, root)

assert root.val == 20

assert root.right is None

def test\_delete\_root():

root = None

for val in [10, 20, 30]:

root = insert(val, root)

root = delete(20, root)

assert root.val == 30

assert root.left.val == 10

assert root.right is None

def test\_tree\_height():

root = None

for val in [10, 5, 15, 3, 7, 13, 17]:

root = insert(val, root)

assert root.height == 3

def test\_tree\_balance\_complex():

root = None

values\_to\_insert = [10, 20, 30, 40, 50, 25]

for val in values\_to\_insert:

root = insert(val, root)

assert abs(get\_balance(root)) <= 1

root = delete(30, root)

assert abs(get\_balance(root)) <= 1

def test\_get\_min\_value\_node():

root = None

values\_to\_insert = [5, 10, 20, 30, 35]

for val in values\_to\_insert:

root = insert(val, root)

min\_node = get\_min\_value\_node(root)

assert min\_node.val == 5

def test\_get\_max\_value\_node():

root = None

values\_to\_insert = [5, 10, 20, 30, 35]

for val in values\_to\_insert:

root = insert(val, root)

max\_node = get\_max\_value\_node(root)

assert max\_node.val == 35

test\_insert\_root()

test\_insert\_right\_child()

test\_insert\_left\_child()

test\_balance\_after\_inserts()

test\_delete\_leaf()

test\_delete\_root()

test\_tree\_height()

test\_tree\_balance\_complex()

test\_get\_min\_value\_node()

test\_get\_max\_value\_node())