**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. 3341 |  | Лодыгин И.А. |
| Преподаватель |  | Иванов Д.В. |

Санкт-Петербург

2024

## Цель работы

Целью данной лабораторной работы является изучение механизмов работы сбалансированных бинарных деревьев, их особенностей и методов взаимодействия с ними.

Для достижения этой цели необходимо выполнить следующие задачи:

1. Реализовать ключевые операции для работы со сбалансированными деревьями, такие как добавление и удаление узлов, включая удаление наименьшего и наибольшего элементов, а также проверку поддержания сбалансированности структуры.
2. Провести анализ производительности реализованных функций на наборах данных различного объема, сравнив результаты с теоретическими оценками временной сложности.
3. Выполнить визуализацию структуры дерева, чтобы продемонстрировать его форму и изменения при выполнении операций.

**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**

В ходе выполнения работы был реализован набор функций для работы со сбалансированными бинарными деревьями поиска (AVL-деревьями), обеспечивающими автоматическую балансировку при добавлении и удалении узлов. Структура дерева представлена классом TreeNode, а функции поддерживают операции с высотой дерева, ротацией узлов и проверкой сбалансированности.

Описание класса и методов

Класс TreeNode: узел дерева представлен классом, имеющим свойства для значения узла (value), указателей на левого и правого потомка (left\_child и right\_child соответственно), а также высоты текущего узла (node\_height).

Функция compute\_height: вычисляет высоту узла. Если узел отсутствует (None), высота равна нулю, иначе используется значение свойства node\_height.

Функция get\_balance: определяет балансировку узла, вычисляя разницу между высотой правого и левого потомка. Баланс служит критерием для проверки необходимости ротации для восстановления сбалансированного состояния дерева.

Функция refresh\_height: пересчитывает высоту узла с учетом высоты его потомков, устанавливая максимальное значение высоты между левым и правым потомками плюс один для текущего узла.

Алгоритмы ротации

Для балансировки дерева используются две функции ротации:

rotate\_right: выполняет правую ротацию, перемещая левый дочерний узел вверх, а текущий узел – вниз вправо. Высота обоих узлов обновляется для соответствия новой структуре.

rotate\_left: аналогично выполняет левую ротацию, перемещая правый дочерний узел вверх, а текущий узел – вниз влево, также с обновлением высот.

Балансировка и вставка значений

Функция rebalance: проверяет баланс узла и применяет ротацию, если балансировка нарушена. Например, если баланс отрицательный и превышает допустимый предел, выполняется правая ротация, что восстанавливает сбалансированность.

Функция insert\_value: реализует добавление нового значения в дерево, начиная с корня. Она использует стек для отслеживания пути и правильного обновления высоты и баланса каждого узла на пути обратно к корню, выполняя при необходимости ротации. Если дерево пустое, создается новый узел как корень.

Удаление узлов

Функция find\_min: находит минимальное значение в дереве, проходя по левым дочерним узлам, что упрощает процесс удаления минимального узла.

Функция delete\_min: удаляет узел с минимальным значением. В случае наличия правого дочернего узла минимального узла, его место занимает правый потомок, а высота и балансировка дерева обновляются.

Функция delete\_value: удаляет конкретное значение из дерева. Если узел с данным значением имеет два потомка, используется функция find\_min для нахождения следующего по значению узла, который заменяет удаляемый. Если потомок всего один или отсутствует, узел удаляется, и баланс восстанавливается.

Проверка сбалансированности и расчет высоты

Функция calculate\_height: вычисляет высоту дерева, используя стек для обхода узлов в глубину.

Функция is\_balanced: рекурсивно проверяет сбалансированность дерева, сравнивая высоты левого и правого поддеревьев каждого узла. Если разница в высоте между потомками узла превышает единицу, возвращается False, что указывает на разбалансированность.

Визуализация дерева

Функция print\_tree: выводит значения дерева на консоль с отступами, соответствующими уровню узла, что позволяет визуально оценить структуру дерева.

**Тестирование**

Анализ производительности операций в AVL – дереве.

Анализ показал, что были выполнены тесты по вставке и удалению элементов в AVL-дерево для различных размеров входных данных. Приведены результаты для размеров 10, 1000, и 100000 элементов. Оценка сложности алгоритмов и результаты тестов следующие:

Сложность алгоритмов:

Лучшее время: O(log n) — при равномерном распределении данных и сбалансированном дереве.

Среднее время: O(log n) — дерево остаётся сбалансированным благодаря AVL-условию.

Худшее время: O(log n) — AVL-дерево предотвращает ухудшение производительности, всегда поддерживая балансировку.

1. void insert\_value(int value) - имеет низкое время выполнения даже при большом количестве элементов, что соответствует теоретической сложности O(log n).

Данные:

Для 10 элементов: 0 s

Для 1000 элементов: 0.006 s

Для 100000 элементов: 1.15 s

2. void delete\_value(int value) - удаление происходит быстро для большинства случаев, но, как видно, даже при 100000 элементов время остается в рамках ожидаемого O(log n).

Данные:

Для 10 элементов: 0 s

Для 1000 элементов: 0.005 s

Для 100000 элементов: 1.01 s

3. void remov\_min() - Время выполнения резко увеличивается при удалении минимального элемента в больших деревьях. Это связано с тем, что для поиска и удаления минимального элемента может потребоваться последовательный переход по левым узлам дерева.

Данные:

Для 10 элементов: 0 s

Для 1000 элементов: 0.0055 s

Для 100000 элементов: 0.76 s

4. void removeMax() – аналогично удалению минимального элемента, удаление максимального элемента требует обхода правых узлов до его нахождения, что также увеличивает время выполнения на больших входных данных.

Данные:

10 элементов: 0 s

1000 элементов: 0.005 s

100000 элементов: 0.77 s

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Рисунок 1 – Данные при тестировании операций на различных оьъёмах данных

## Выводы

Результатом данной лабораторной работы стала реализация структуры данных AVL-дерева, а также проведение замеров основных операций на различных объемах данных (10, 1000 и 100000 элементов). На основании полученных результатов можно сделать следующие выводы:

Эффективность AVL-дерева: Исследование подтвердило, что AVL-дерево представляет собой высокоэффективную структуру данных для выполнения операций вставки и удаления. Среднее и худшее время выполнения этих операций не превышает O(log n), что соответствует теоретическим ожиданиям.

Скорость выполнения операций: Операции вставки и удаления по значению выполнялись за миллисекунды, даже при объемах данных, достигающих 100000 элементов.

Влияние размера данных на производительность: При увеличении количества элементов время выполнения операций увеличивалось, однако оставалось в пределах приемлемых значений. Например, время удаления минимальных и максимальных значений значительно возросло при объеме данных до 100000 элементов. Это подчеркивает важность учета структуры дерева при выполнении подобных операций, а также может быть связано с особенностями реализации балансировки и рекурсивным характером выполнения операций.

# Приложение А Исходный код программы

Название файла: main.py

class TreeNode:

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

self.value = value

self.left\_child = left\_child

self.right\_child = right\_child

self.node\_height = 1

def compute\_height(node: TreeNode) -> int:

return 0 if node is None else node.node\_height

def get\_balance(node: TreeNode) -> int:

return compute\_height(node.right\_child) - compute\_height(node.left\_child)

def refresh\_height(node: TreeNode):

left\_height = compute\_height(node.left\_child)

right\_height = compute\_height(node.right\_child)

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

def rotate\_right(node: TreeNode) -> TreeNode:

pivot = node.left\_child

node.left\_child = pivot.right\_child

pivot.right\_child = node

refresh\_height(node)

refresh\_height(pivot)

return pivot

def rotate\_left(node: TreeNode) -> TreeNode:

pivot = node.right\_child

node.right\_child = pivot.left\_child

pivot.left\_child = node

refresh\_height(node)

refresh\_height(pivot)

return pivot

def rebalance(node: TreeNode) -> TreeNode:

refresh\_height(node)

balance = get\_balance(node)

if balance < -1:

if get\_balance(node.left\_child) > 0:

node.left\_child = rotate\_left(node.left\_child)

return rotate\_right(node)

if balance > 1:

if get\_balance(node.right\_child) < 0:

node.right\_child = rotate\_right(node.right\_child)

return rotate\_left(node)

return node

def insert\_value(value: int, node: TreeNode) -> TreeNode:

stack = []

current = node

while current:

stack.append(current)

if value < current.value:

if current.left\_child is None:

current.left\_child = TreeNode(value)

break

current = current.left\_child

else:

if current.right\_child is None:

current.right\_child = TreeNode(value)

break

current = current.right\_child

if not stack:

return TreeNode(value)

while stack:

current = stack.pop()

current = rebalance(current)

return current

def find\_min(node: TreeNode) -> TreeNode:

while node.left\_child:

node = node.left\_child

return node

def delete\_max(node: TreeNode) -> TreeNode:

parent, current = None, node

while current.right\_child:

parent, current = current, current.right\_child

if parent:

parent.right\_child = current.left\_child

return rebalance(node)

else:

return current.left\_child

def delete\_min(node: TreeNode) -> TreeNode:

parent, current = None, node

while current.left\_child:

parent, current = current, current.left\_child

if parent:

parent.left\_child = current.right\_child

return rebalance(node)

else:

return current.right\_child

def delete\_value(value: int, node: TreeNode) -> TreeNode:

parent = None

current = node

while current and current.value != value:

parent = current

if value < current.value:

current = current.left\_child

else:

current = current.right\_child

if current is None:

return node

if current.left\_child and current.right\_child:

min\_larger\_node = find\_min(current.right\_child)

current.value = min\_larger\_node.value

current.right\_child = delete\_min(current.right\_child)

elif current.left\_child or current.right\_child:

child = current.left\_child if current.left\_child else current.right\_child

if parent is None:

return child

if parent.left\_child == current:

parent.left\_child = child

else:

parent.right\_child = child

else:

if parent is None:

return None

if parent.left\_child == current:

parent.left\_child = None

else:

parent.right\_child = None

return rebalance(node)

def calculate\_height(node: TreeNode) -> int:

stack = [(node, 1)]

max\_height = 0

while stack:

node, height = stack.pop()

if node is not None:

max\_height = max(max\_height, height)

stack.append((node.left\_child, height + 1))

stack.append((node.right\_child, height + 1))

return max\_height

def is\_balanced(node: TreeNode) -> bool:

if node is None:

return True

left\_height = calculate\_height(node.left\_child)

right\_height = calculate\_height(node.right\_child)

if abs(left\_height - right\_height) > 1:

return False

return is\_balanced(node.left\_child) and is\_balanced(node.right\_child)

def print\_tree(node: TreeNode, level=0):

if node is not None:

print\_tree(node.right\_child, level + 1)

print(' ' \* 4 \* level + '->', node.value)

print\_tree(node.left\_child, level + 1)

Название файла: tests.py

from main import insert\_value, is\_balanced, delete\_value, print\_tree

from random import randint

import time

test\_data\_10 = [randint(0, 100) for \_ in range(10)]

test\_data\_1000 = [randint(0, 100) for \_ in range(1000)]

test\_data\_100000 = [randint(0, 100) for \_ in range(100000)]

def measure\_insert\_time(data):

tree = None

start\_time = time.time()

for value in data:

tree = insert\_value(value, tree)

elapsed\_time = time.time() - start\_time

assert is\_balanced(tree), "Tree is not balanced after insertion"

return elapsed\_time

def measure\_delete\_time(data):

tree = None

for value in data:

tree = insert\_value(value, tree)

start\_time = time.time()

for value in data:

tree = delete\_value(value, tree)

elapsed\_time = time.time() - start\_time

assert is\_balanced(tree), "Tree is not balanced after deletion"

return elapsed\_time

print(f"Insertion time for 10 elements: {measure\_insert\_time(test\_data\_10):.6f} seconds")

print(f"Insertion time for 1000 elements: {measure\_insert\_time(test\_data\_1000):.6f} seconds")

print(f"Insertion time for 100000 elements: {measure\_insert\_time(test\_data\_100000):.6f} seconds")

print(f"Deletion time for 10 elements: {measure\_delete\_time(test\_data\_10):.6f} seconds")

print(f"Deletion time for 1000 elements: {measure\_delete\_time(test\_data\_1000):.6f} seconds")

print(f"Deletion time for 100000 elements: {measure\_delete\_time(test\_data\_100000):.6f} seconds")