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 _____ Ягудин Д.Р.

Преподаватель _____ Иванов Д.В.

Санкт-Петербург
2024

Цель работы

Целью данной лабораторной работы - изучение структуры данных АВЛ-дерева, его свойств и методов обработки.

Для достижения поставленной цели нужно выполнить следующие задачи:

1. Реализовать основные операции, связанные с АВЛ-деревьями (вставка и удаление узлов, включая удаление максимального и минимального элемента, а также проверка сбалансированности дерева).

2. Провести исследование производительности реализованных операций на различных объемах данных, сравнить их с теоретическими оценками временной сложности.

3. Подготовить визуализацию АВЛ-дерева, чтобы наглядно продемонстрировать его структуру 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

Реализация AVL-дерева:

Вот описание функций в данном коде:

1. Класс `Node`:
   - Создает узел для дерева с полями:
     - `val` — значение узла.
     - `left` — левый потомок (по умолчанию `None`).
     - `right` — правый потомок (по умолчанию `None`).
     - `height` — высота узла, изначально равная 1.

2. `height(root: Node) -> int`:

- Возвращает высоту узла. Если узел `root` равен `None`, возвращает 0 (для отсутствующих узлов).

3. `balance_factor(root: Node) -> int`:

- Вычисляет и возвращает балансировочный фактор узла: разницу между высотой правого и левого поддеревьев. Используется для оценки сбалансированности узла.

4. `fix_height(root: Node)`:

- Пересчитывает высоту узла `root`, как максимальное значение между высотой его левого и правого потомков, увеличенное на 1.

5. `right_rotation(node)`:

- Выполняет правое вращение дерева относительно узла `node`. Левый потомок становится новым корнем, а правый потомок этого нового корня перемещается на место левого потомка узла `node`. Высоты обновляются. Возвращает новый корень поддерева.

6. `left_rotation(node)`:

- Выполняет левое вращение дерева относительно узла `node`. Правый потомок становится новым корнем, а левый потомок этого нового корня перемещается на место правого потомка узла `node`. Высоты обновляются. Возвращает новый корень поддерева.

7. `lr_rotation(node)`:

- Выполняет сначала левое вращение для левого потомка узла, а затем правое вращение относительно узла `node`. Используется для балансировки, когда левое поддерево перекошено вправо.

8. `rl_rotation(node)`:

- Выполняет сначала правое вращение для правого потомка узла, а затем левое вращение относительно узла `node`. Используется для балансировки, когда правое поддерево перекошено влево.

9. `insert(data, node)`:

- Вставляет новый узел с данными `data` в дерево, начиная с узла `node`. Если узел пустой, создается новый узел.

- Выполняется проверка баланса поддеревьев после вставки. Если разница в высоте поддеревьев превышает 1, выполняются соответствующие вращения (правое, левое, lr или rl вращения). Высота узла пересчитывается.

10. `balance(root: Node) -> Node`:

- Балансирует узел `root`. Если балансировочный фактор показывает, что узел перекошен, выполняются соответствующие вращения (правое или левое).

- Вызывает корректировку высоты узла перед балансировкой и возвращает новый корень.

11. `find_min(root)`:

- Находит узел с минимальным значением в дереве, рекурсивно переходя к левому потомку. Возвращает узел с минимальным значением.

12. `delete_min(root)`:

- Удаляет узел с минимальным значением. Если у узла нет левого потомка, возвращает правый. Если левый потомок есть, рекурсивно вызывает удаление. После удаления выполняется балансировка.

13. `delete_max(root)`:

- Удаляет узел с максимальным значением. Аналогично функции `delete_min`, но работает с правым поддеревом. После удаления выполняется балансировка дерева.

14. `delete_value(val, root)`:

- Удаляет узел с заданным значением `val`. Если значение меньше текущего узла, рекурсивно удаляется в левом поддереве. Если больше, в правом. Если найден узел, он заменяется минимальным узлом в правом поддереве (если правое поддерево существует). После удаления происходит балансировка.

15. `get_high(root: Node)`:

- Вычисляет высоту дерева от узла `root`. Рекурсивно вычисляет высоту левого и правого поддеревьев и возвращает максимальное значение плюс 1.

16.`compare_high(root):

- Сравнивает высоты левого и правого поддеревьев узла `root`. Если разница больше 1, возвращает `False`, иначе — `True`.

17. `check(root: Node) -> bool`:

- Проверяет, сбалансировано ли всё дерево. Рекурсивно проверяет балансировку всех узлов. Если хотя бы один узел несбалансирован, возвращает `False`.

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(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: 0.004 s

Для 100000 элементов: 0.7 s

2. void remove(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: 0.004 s

Для 100000 элементов: 1.03 s

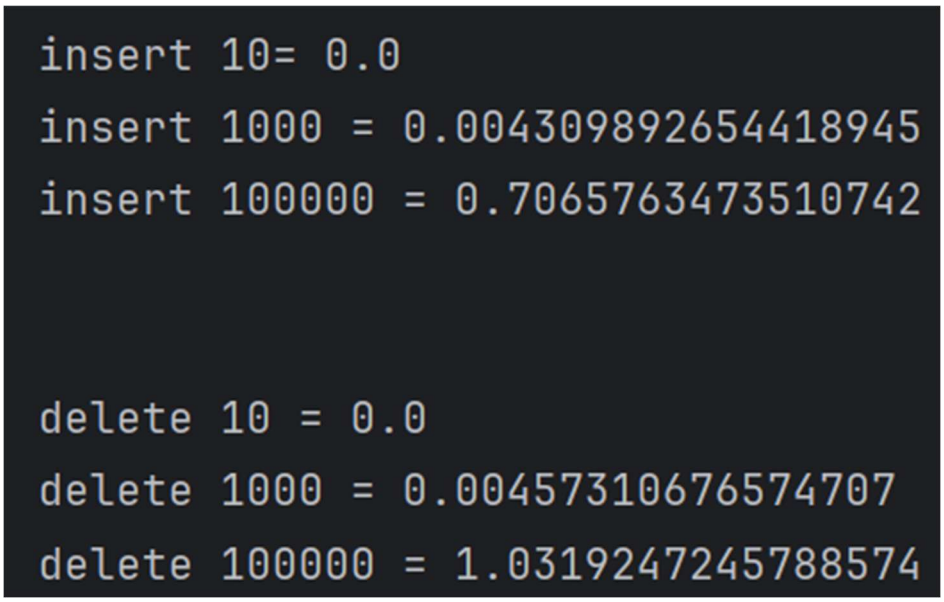

Рисунок 1 – Данные при тестировании операций на различных оьъёмах данных

Выводы

Результатом лабораторной работы стала реализация структуры данных АВЛ-дерева, также были проделаны замеры основных операций на различных объёмах данных 10, 1000, 100000 элементах.

После чего можно сделать выводы:

1. Эффективность AVL-дерева: Результаты исследования показали, что AVL-дерево является высокоэффективной структурой данных для выполнения операций вставки и удаления. Среднее и худшее время выполнения операций не превышает O (log n), что подтверждает теоретические ожидания.

2. Скорость выполнения операций: Вставка и удаление по значению выполнялись за миллисекунды даже при объемах данных (до 100000 элементов).

3. Влияние размера данных на производительность: При увеличении количества элементов время выполнения операций увеличивалось, но оставалось в рамках приемлемых значений. Например, время удаления минимального и максимального значений резко возросло при объемах данных (до 100000 элементов), что указывает на важность учета структуры дерева при выполнении таких операций, также возможно на такой рост могли повлиять реализация балансировки и рекурсивная реализация.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

```
     class Node:
    def __init__(self, val, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
        self.height = 1

def height(root: Node) -> int:
    if root is None:
        return 0
    return root.height

def balance_factor(root: Node) -> int:
    return height(root.right) - height(root.left)

def fix_height(root: Node):
    root.height = max(height(root.left), height(root.right)) + 1

def right_rotation(node):
    ret = node.left
    assert ret is not None

    node.left = ret.right
    ret.right = node

    node.height = max((node.left.height if node.left else 0),
(node.right.height if node.right else 0)) + 1
    ret.height = max((ret.left.height if ret.left else 0),
(ret.right.height if ret.right else 0)) + 1

    return ret


def left_rotation(node):
    ret = node.right
    assert ret is not None

    node.right = ret.left
    ret.left = node

    node.height = max((node.left.height if node.left else 0),
(node.right.height if node.right else 0)) + 1
    ret.height = max((ret.left.height if ret.left else 0),
(ret.right.height if ret.right else 0)) + 1

    return ret


def lr_rotation(node):
    node.left = left_rotation(node.left)
```

```
        return right_rotation(node)


    def rl_rotation(node):
        node.right = right_rotation(node.right)
        return left_rotation(node)


    def insert(data, node):
        if node is None:
            return Node(data)

        if data < node.val:
            node.left = insert(data, node.left)
            if    (node.left.height    if    node.left    else    0)    -
(node.right.height if node.right else 0) == 2:
                if data < node.left.val:
                    node = right_rotation(node)
                else:
                    node = lr_rotation(node)
        else:
            node.right = insert(data, node.right)
            if    (node.right.height    if    node.right    else    0)    -
(node.left.height if node.left else 0) == 2:
                if data < node.right.val:
                    node = rl_rotation(node)
                else:
                    node = left_rotation(node)

        node.height  =  max((node.left.height  if  node.left  else  0),
(node.right.height if node.right else 0)) + 1
        return node

    def balance(root: Node) -> Node:
        fix_height(root)
        if balance_factor(root) == -2:
            if balance_factor(root.left) > 0:
                root.left = left_rotation(root.left)
            return right_rotation(root)

        elif balance_factor(root) == 2:
            if balance_factor(root.right) < 0:
                root.right = right_rotation(root.right)
            return left_rotation(root)

        return root

    def find_min(root):
        if root.left is None:
            return root
        return find_min(root.left)

    def delete_min(root):
        if root.left is None:
            return root.right
        root.left = delete_min(root.left)
        return balance(root)
```

```
 def delete_max(root):
     if root.right is None:
         return root.left
     root.right = delete_max(root.right)
     return balance(root)

 def delete_value(val, root):
     if root is None:
         return None
     if val < root.val:
         root.left = delete_value(val, root.left)
     elif val > root.val:
         root.right = delete_value(val, root.right)
     else:
         l = root.left
         r = root.right
         if r is not None:
             min_node = find_min(r)
             min_node.right = delete_min(r)
             min_node.left = l
             return balance(min_node)
         else:
             return l
     return balance(root)

 def get_high(root:Node):
     if root is None:
         return 0
     if root.right is None and root.left is None:
         return 1
     high_l, high_r = 0, 0

     if root.left is not None:
         high_l = get_high(root.left)

     if root.right is not None:
         high_r = get_high(root.right)

     return max(high_r, high_l) + 1


 def compare_high(root):
     if abs(get_high(root.left) - get_high(root.right)) > 1:
         return False
     return True

 def check(root:Node) -> bool:
     if root is None:
         return True
     if compare_high(root) is False:
         return False
return check(root.left) and check(root.right)
```

Название файла: test.py

```
from main import insert, delete_value, check
from random import randint
import time


arr10 = [randint(0, 100) for x in range(10)]
arr1000 = [randint(0, 100) for x in range(1000)]
arr100000 = [randint(0, 100) for x in range(100000)]


def time_insert(arr):
    tree = None
    start = time.time()
    for item in arr:
        tree = insert(item, tree)
    end = time.time()
    assert (check(tree) == True)
    return end - start

def time_delete(arr):
    tree = None
    for item in arr:
        tree = insert(item, tree)

    start = time.time()
    for i in range(len(arr)):
        delete_value(arr[i], tree)
    end = time.time()
    assert (check(tree) == True)
    return end - start

print(f"insert 10= {time_insert(arr10)}")
print(f"insert 1000 = {time_insert(arr1000)}")
print(f"insert 100000 = {time_insert(arr100000)}\n\n")

print(f"delete 10 = {time_delete(arr10)}")
```

```
print(f"delete 1000 = {time_delete(arr1000)}")
print(f"delete 100000 = {time_delete(arr100000)}\n\n")
```