МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №1

по дисциплине «Алгоритмы и структуры данных»

Тема: АВЛ-деревья.

Студент гр. 3388 _____ Снигирев А.А.

Преподаватель _____ Шалагинов И.В.

Санкт-Петербург

2024

Цель работы

Изучить структуру данных АВЛ-дерево, реализовать основные методы к нему и алгоритм балансировки. Провести исследование времени работы и используемой памяти.

Задание

Реализация

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

```
Class Node:
     left: pointer
     right: pointer
     parent: pointer
     value: AnyType
     height: int
     balance: int
```

Хранит в себе все нужные узлу поля. В конструкторе принимает только значение `value` и создает ноду без связей.

`Class AVLTree` – основной класс структуры данных. Содержит все нужные поля и методы.

Поля:

`root: Node` – корень авл-дерева

Методы:

`search_place(key)` – один из важнейших методов. Спускается вниз по дереву, сравнивая значения каждого встреченного узла с `key`. Если узел найден — возвращается указатель на него. Если не найден, то возвращается указатель на узел, ребенком которого может стать узел со значением `key`.

`insert(key)` – метод вставки. Создает узел new со значением `key`. Обрабатывает несколько возможных случаев:

- Если корня дерева не существует, то узел new им становится.
- Если корень есть, вызывает метод `search_place`
    1. Если метод вернул узел, значение которого совпадает с new, то ничего не происходит, чтобы не дублировать узлы.
    2. Иначе происходит присваивание полю `left` или `right` указателя new

Далее происходит балансировка.

`delete(key)` – удаление. Вызывает метод `search_place` для поиска узла. Если узел найден, обрабатывается несколько случаев:

- Если хотя бы одного узла не существует, вызывается метод `__replace_node_in_parent()`, который заменяет узел на его сына, правого или левого.
- Если есть оба сына, происходит поиск минимального элемента в правом поддереве удаляемого. Значение удаляемого узла заменяется на значение минимального. Минимальный узел-лист удаляется по-первому варианту алгоритма.

Вызывается балансировка для всех узлов, начиная с родителя удаленного элемента.

- __update_balances_iterative(node) - вспомогательная функция. В ней цикл, проходящий вверх по дереву, обновляющий поля balance и height встречаемых нод функцией __update_height_balances() и восстанавливающий нарушенный баланс функцией __restore_balance__().
- __update_height_balance(node) – принимает ноду и обновляет ее высоту, беря максимум от высот дочерних нод плюс один.
- __restore_balance__(node) – принимает ноду. Если ее баланс <- 1, что означает превосходство в длине левого поддерева, а баланс левого сына <=0, вызывает функцию правого поворота __rotate_r__, если же баланс левого сына больше нуля, то просиходит двойной поворот: сначала левый для node.left, а после правый для node.

  Случай, когда баланс node >1, симметричен вышеописанному.
- __rotate_r__(node) – проверяет наличие дочерних нод и производит правый поворот, а затем балансировку.

  __rotate_l__(node) – симметричен.
- in_order() - совершает прямой обход.
- render_avl() и create_graph()- визуализация

Исходный код см. в Приложении А.

Исследование

Результаты запусков на разных объемах данных:

Элементов	**Вставка**	**Удаление**
100	0.000030	0.000023
1000	0.000032	0.000034
10000	0.000042	0.000038
100000	0.000075	0.000074

Оценка сложности AVL-Tree:

Вставка — O(log(n))

Удаление — O(log(n))

Как видно по таблице, с ростом числа элементов 10-кратно каждый раз, время выполнения программы растет намного медленнее, чем линейно.

Следовательно, реализованная структура данных по времени выполнения соответствует теоретическим показателям.

Вывод

Были исследованы АВЛ-деревья и реализованы основные функции к ним. Проведены исследования и сравнения с теоретическими показаниями на разных объемах данных. АВЛ-дерево показало себя весьма быстрой и эффективной структурой данных.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

```
from modules.avl_tree import AVLTree

tree = AVLTree()
arr = [int(number) for number in input().split()]
a = datetime.now()
for i in arr:
    tree.insert(i)
print(tree.pre_order(tree.root, []))
b = datetime.now()
print(b-a)
tree.render_avl_tree()
```

Название файла avl_tree.py:

```
import graphviz as gv

class Node:
    def __init__(self, key) -> None:
        self.balance = 0
        self.height = 0
        self.right = None
        self.left = None
        self.parent = None
        self.value = key

class AVLTree:
    def __init__(self):
        self.root = None

    def search_place(self, key) -> Node:
        curr = self.root
        parent = None
        while curr is not None:
            parent = curr
            if key < curr.value:
                curr = curr.left
            elif key > curr.value:
                curr = curr.right
            else:
                return curr
        return parent

    def insert(self, key):
        new = Node(key)
        if self.root is None:
            self.root = new
            return

        parent = self.search_place(key)
```

```
            if parent and parent.value == key:
                return

            new.parent = parent
            if key < parent.value:
                parent.left = new
            else:
                parent.right = new

            self.__update_balances_iterative(new)

        def delete(self, key):
            node_to_delete = self.search_place(key)
            if node_to_delete is None or node_to_delete.value != key:
                return

            if node_to_delete.left is None or node_to_delete.right is
None:
                        self.__replace_node_in_parent(node_to_delete,
node_to_delete.left or node_to_delete.right)
            else:
                                                        successor    =
self._get_min_value_node(node_to_delete.right)
                node_to_delete.value = successor.value
                            self.__replace_node_in_parent(successor,
successor.left)

            parent = node_to_delete.parent
            self.__update_balances_iterative(parent)

        def __replace_node_in_parent(self, node, new_child):
            if node.parent:
                if node == node.parent.left:
                    node.parent.left = new_child
                else:
                    node.parent.right = new_child
            else:
                self.root = new_child
            if new_child:
                new_child.parent = node.parent

        def _get_min_value_node(self, node):
            current = node
            while current.left is not None:
                current = current.left
            return current

        def __update_balances_iterative(self, node):
            while node:
                self.__update_height_balance(node)
                self.__restore_balance__(node)
                node = node.parent

        def __restore_balance__(self, node):
            if node is None:
                return node
```

```
        if node.balance < -1:
            if node.left and node.left.balance <= 0:
                return self.__rotate_r__(node)
            elif node.left and node.left.balance > 0:
                node.left = self.__rotate_l__(node.left)
                return self.__rotate_r__(node)
        elif node.balance > 1:
            if node.right and node.right.balance >= 0:
                return self.__rotate_l__(node)
            elif node.right and node.right.balance < 0:
                node.right = self.__rotate_r__(node.right)
                return self.__rotate_l__(node)

        return node

    def __rotate_r__(self, node):
        if node is None or node.left is None:
            return node

        y = node.left
        node.left = y.right
        if y.right is not None:
            y.right.parent = node
        y.parent = node.parent
        if node.parent is None:
            self.root = y
        elif node == node.parent.right:
            node.parent.right = y
        else:
            node.parent.left = y
        y.right = node
        node.parent = y
        self.__update_height_balance(node)
        self.__update_height_balance(y)
        return y

    def __rotate_l__(self, node):
        if node is None or node.right is None:
            return node

        y = node.right
        node.right = y.left
        if y.left is not None:
            y.left.parent = node
        y.parent = node.parent
        if node.parent is None:
            self.root = y
        elif node == node.parent.left:
            node.parent.left = y
        else:
            node.parent.right = y
        y.left = node
        node.parent = y
        self.__update_height_balance(node)
        self.__update_height_balance(y)
```

```
            return y

        def __update_height_balance(self, node):
            left_height = node.left.height if node.left else -1
            right_height = node.right.height if node.right else -1
            node.height = max(left_height, right_height) + 1
            node.balance = right_height - left_height

        def in_order(self, curr, res):
            if curr is not None:
                self.in_order(curr.left, res)
                res.append(curr.value)
                self.in_order(curr.right, res)
            return res


        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.gv1', view=True)


    tree = AVLTree()
    arr = [212, 69, 369, 390, 145,51, 350, 350 ]
    for i in range(50):
        tree.insert(i)
    tree.delete(15)
    print(tree.in_order(tree.root, []))
    tree.render_avl_tree()
```