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. 3384 _____ Пьянков М.Ф.

Преподаватель _____ Шестопалов Р.П.

Санкт-Петербург
2024

Цель работы

Изучение и реализация структуры данных: АВЛ-дерево, а также исследование время работы различных методов этой СД. Выполнение задания.

Задание

Дано авл-дерево. Реализуйте функцию insert, которая на вход принимает корень дерева и значение которое нужно добавить в это дерево.

Ограничения:

2 <= N <= 1000

-1000 <= значения узлов <= 1000

сигнатура функции insert на python:

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

Определение класса Node:

```
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
```

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

Все методы выбранной структуры данных будут реализованы на основе циклических алгоритмов, а не рекурсивных в целях уменьшения затрат памяти и повышении стабильности работы.

Метод __get_height__(node) – статический метод, необходимый для корректного получения высоты узла.

Метод __height_update__(self, node) — метод, обновляющий веса данного узла.

Методы поворотов:

__small_left_rotate__(self, node) — малы левый поворот

__small_right_rotate__(self, node) — малый правый поворот

__large_left_rotate__(self, node) — большой левый поворот

__large_right_rotate__(self, node) — большой правый поворот

Методы больших поворотов были реализованы с помощью малых поворотов.

Метод __balance_factor__(self, node) - метод позволяющий определить, нужно ли начинать балансировку в данном узле.

Метод __make_rotations__(self, node) - метод вызывающий методы вращения на основе значений метода __balance_factor__

Метод __balance__(self, node) - метод начинающий балансировку от текущего узла к корню дерева.

Метод search(self, value) — циклический поиск элемента в дереве.

Метод insert(self, value) — циклическая вставка нового элемента в дерево.

Метод pop(self, value) — циклическое удаление элемента по значению из дерева.

Методы pop_min(self) и pop_max(self) реализуют удаление минимального и максимального элемента дерева соответственно.

Метод check_from_node(root: AVLTreeNode) -> bool — статический метод, проверяющий дерево с корнем в данном узле на соответствие свойствам АВЛ-дерева.

Метод check_global(self) — метод, запускающий check_from_node в корне дерева.

Метод diff_from_node(root: AVLTreeNode, minimal=1e10) -> int — статический метод, вычисляющий минимальную разницу между соседними элементами дерева, корнем которого является данный узел.

Метод diff_global(self) — метод, вызывающий diff_from_node в корне дерева.

Метод in_order(self, current) — метод, осуществляющий прямой обход дерева, необходим для тестирования и визуализации элементов дерева.

Метод create_graph(self) — специальный метод, для визуализации дерева с помощью библиотеки graphviz.

Метод render_avl_tree(self) — метод, генерирующий изображение дерева.

Разработанный программный код см. в приложении А.

Исследование

Проведём исследование скорости работы методов вставки, и методов удаления элементов из дерева.

Рисунок 1 — Вставка

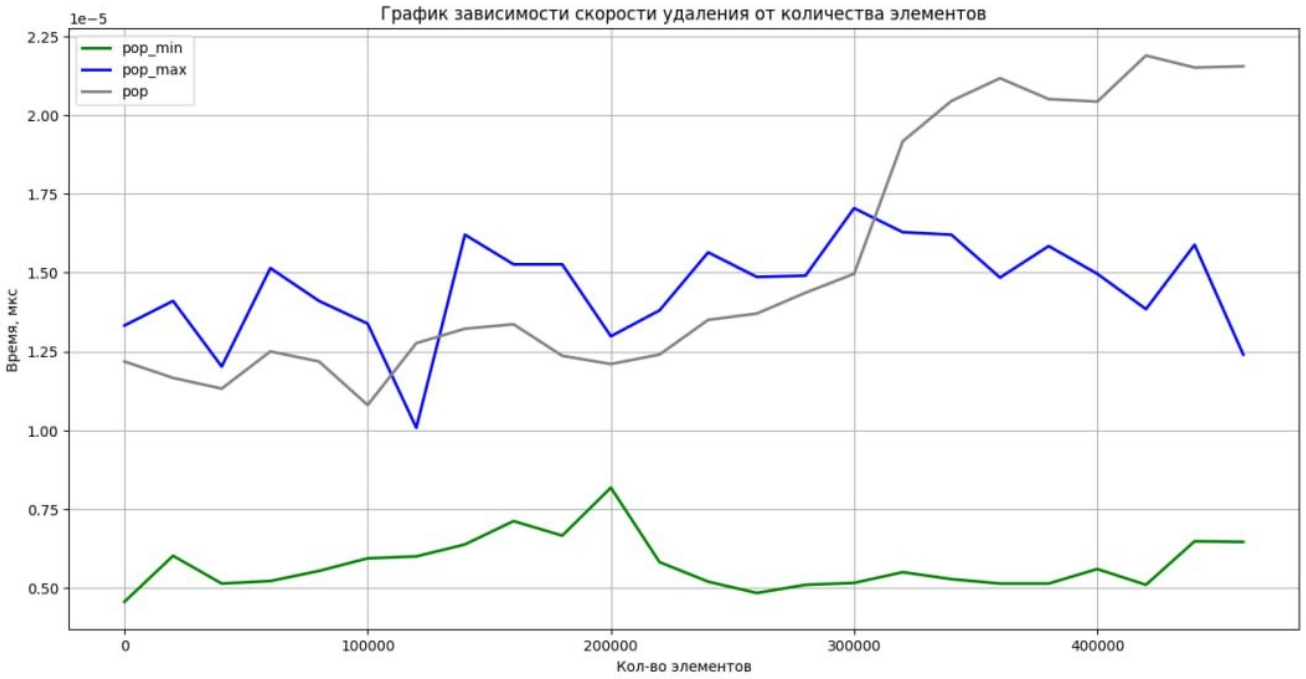

Рисунок 2 — Удаление

На графике скорости вставки (см рис. 1) видна сложность O(logn), на графике скорости удаления (см рис. 2)тоже видна сложность O(logn), однако не так очевидно. Следует отметить что трудности в измерении времени удаления элементов в дереве и сопоставлении этой зависимости какой-либо функции вызывает сложности, в следствии того, что после каждой итерации расположении элементов не определено из-за алгоритмов вращения. Но теоретические положения и реализация методов позволяют убедится в том что сложность достигает O(logn). Пояснение в случае удаления элемента необходимо каждый раз выбирать из двух детей узла , очевидно, что необходимо не более log2(n) операций. Однако трудности возникают в том, что после добавления или удаления возможно нужно провести балансировку, для этого от текущего элемента необходимо вернуться к корню и в случае надобности провести балансировку деревьев. Так как сложность балансировки O(1), то соответственно полное количество операций и время исполнения при вставке/удалении не может превышать 2log2(n). Но так как не представляется возможным протестировать АВЛ-дерево на больших данных на графиках наблюдаются аномалии.

Выводы

Была успешно изучена и реализована структура данных АВЛ-дерево. Также было проведено тестирование и исследование, в результате которого были получены в целом результаты соответствующие теоретическим представлениям. Также была успешно выполнена поставленная задача.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл main_lb3.py

```
def get_height(node):
    return node.height if node else 0

def update_height(node):
     node.height = max(get_height(node.left), get_height(node.right)) +
1

def balance_factor(node):
    return get_height(node.right) - get_height(node.left)

def small_right_rotate(node):
    left = node.left
    left_right = left.right
    left.right = node
    node.left = left_right
    update_height(node)
    update_height(left)
    return left

def small_left_rotate(node):
    right = node.right
    right_left = right.left
    right.left = node
    node.right = right_left
    update_height(node)
    update_height(right)
    return right

def __balance__(node):
    b_f = balance_factor(node)
    if b_f == -2:
        if balance_factor(node.left) <= 0:
            return small_right_rotate(node)
        else:
            node.left = small_left_rotate(node.left)
            return small_right_rotate(node)
    elif b_f == 2:
        if balance_factor(node.right) >= 0:
            return small_left_rotate(node)
        else:
            node.right = small_right_rotate(node.right)
            return small_left_rotate(node)
    return node

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)

    update_height(node)
    return __balance__(node)
```

Файл avl_tree.py

```
import graphviz as gv


class AVLTreeNode:
    def __init__(self, value):
        self.parent = None
        self.left = None
        self.right = None
        self.height = 1
        self.value = value


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

    @staticmethod
    def __get_height__(node):
        return node.height if node else 0

    def __height_update__(self, node):
                    node.height   =   max(self.__get_height__(node.left),
self.__get_height__(node.right)) + 1

    def __small_left_rotate__(self, node):
        temp = node.right
        temp_left = temp.left
        temp.parent = node.parent
        node.parent = temp
        if temp_left is not None:
            temp_left.parent = node
        node.right = temp_left
        temp.left = node
        if temp.parent is not None:
            if temp.parent.right == node:
                temp.parent.right = temp
            elif temp.parent.left == node:
                temp.parent.left = temp
        else:
            self.root = temp
        self.__height_update__(node)
        self.__height_update__(temp)

    def __small_right_rotate__(self, node):
        temp = node.left
        temp_right = temp.right
        temp.parent = node.parent
        node.parent = temp
        if temp_right is not None:
            temp_right.parent = node
        node.left = temp_right
        temp.right = node
        if temp.parent is not None:
            if temp.parent.left == node:
```

```
                temp.parent.left = temp
            elif temp.parent.right == node:
                temp.parent.right = temp
        else:
            self.root = temp
        self.__height_update__(node)
        self.__height_update__(temp)

    def __large_left_rotate__(self, node):
        self.__small_right_rotate__(node.right)
        self.__small_left_rotate__(node)

    def __large_right_rotate__(self, node):
        self.__small_left_rotate__(node.left)
        self.__small_right_rotate__(node)

    def __balance_factor__(self, node):
        left = self.__get_height__(node.left) if node else 0
        right = self.__get_height__(node.right) if node else 0
        return left - right

    def __make_rotations__(self, node):
        balance_factor = self.__balance_factor__(node)
        if balance_factor == -2:
            if self.__balance_factor__(node.right) > 0:
                self.__large_left_rotate__(node)
            else:
                self.__small_left_rotate__(node)

        elif balance_factor == 2:
            if self.__balance_factor__(node.left) < 0:
                self.__large_right_rotate__(node)
            else:
                self.__small_right_rotate__(node)

    def __balance__(self, node):
        current = node
        next_parent = current.parent
        self.__height_update__(current)
        while next_parent is not None:
            current = next_parent
            next_parent = current.parent
            self.__height_update__(current)
            self.__make_rotations__(current)

    def search(self, value):
        current = self.root
        while current is not None:
            if current.value > value:
                current = current.left
            elif current.value < value:
                current = current.right
            else:
                return True
        return False
```

```
    def insert(self, value):
        if self.root is None:
            self.root = AVLTreeNode(value)
            return
        current = self.root
        new_node = AVLTreeNode(value)
        while current is not None:
            if value <= current.value:
                if current.left is None:
                    new_node.parent = current
                    current.left = new_node
                    break
                current = current.left
            else:
                if current.right is None:
                    new_node.parent = current
                    current.right = new_node
                    break
                current = current.right
        self.__balance__(current)

    def pop(self, value):
        if self.root is None:
            return
        current = self.root
        while current is not None:
            if current.value > value:
                current = current.left
            elif current.value < value:
                current = current.right
            else:
                break
        if current is None:
            return
            if current is self.root and self.root.left is None and
self.root.right is None:
            self.root = None
            return
        if current is self.root and self.root.right is None:
            self.root.left.parent = None
            self.root = self.root.left
            return
        if current.right is None:
            if current.left is not None:
                current.left.parent = current.parent

            if current == current.parent.left:
                current.parent.left = current.left
            elif current == current.parent.right:
                current.parent.right = current.left
        else:
            new_current = current.right
            while new_current.left is not None:
                new_current = new_current.left
            current.value = new_current.value
            if new_current.right is not None:
```

```
                new_current.right.parent = new_current.parent

            if new_current.parent != current:
                if new_current.right is not None:
                    new_current.right.parent = new_current.parent
                new_current.parent.left = new_current.right
            elif new_current.parent == current:
                if current.right.right is not None:
                    current.right.right.parent = current
                current.right = current.right.right
            self.__balance__(new_current)
            return
        self.__balance__(current)

    def pop_min(self):
        if self.root is None:
            return
        current = self.root
        while current.left is not None:
            current = current.left
        if current == self.root:
            if current.right is None:
                self.root = None
            else:
                current.right.parent = None
                self.root = current.right
        else:
            if current.right is not None:
                current.right.parent = current.parent
            current.parent.left = current.right
        self.__balance__(current)

    def pop_max(self):
        if self.root is None:
            return
        current = self.root
        while current.right is not None:
            current = current.right
        if current == self.root:
            if current.left is None:
                self.root = None
            else:
                current.left.parent = None
                self.root = current.left
        else:
            if current.left is not None:
                current.left.parent = current.parent
            current.parent.right = current.left
        self.__balance__(current)

    @staticmethod
    def check_from_node(root: AVLTreeNode) -> bool:
        if root is None:
            return True
        left_height = AVLTree.__get_height__(root.left)
        right_height = AVLTree.__get_height__(root.right)
```

```
        if abs(left_height - right_height) <= 1 and \
                                    AVLTree.check_from_node(root.left)  and
AVLTree.check_from_node(root.right):
            return True
        return False

    def check_global(self):
        return self.check_from_node(self.root)

    @staticmethod
    def diff_from_node(root: AVLTreeNode, minimal=1e10) -> int:
        if root is None:
            return int(minimal)
        if root.left is not None:
            minimal = min(abs(root.value - root.left.value), minimal)
        if root.right is not None:
            minimal = min(abs(root.value - root.right.value), minimal)
               return  min(AVLTree.diff_from_node(root.left,  minimal),
AVLTree.diff_from_node(root.right, minimal), minimal)

    def diff_global(self):
        return self.diff_from_node(self.root)

    def in_order(self, current):
        if current is not None:
            if current.left is None and current.right is None:
                return current.value
            if current.left is not None and current.right is not None:
                 return f'{self.in_order(current.left)} {current.value}
{self.in_order(current.right)}'
            if current.left is not None:
                                    return  f'{self.in_order(current.left)}
{current.value}'
            if current.right is not None:
                                               return   f'{current.value}
{self.in_order(current.right)}'

    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.gv', view=True)
```