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 Рудаков А.Л.

Преподаватель Шестопалов Р.П.

Санкт-Петербург
2024

Цель работы.

Изучить, что из себя представляет АВЛ-дерево, и реализовать данную структуру данных. Также необходимо провести исследование его работы.

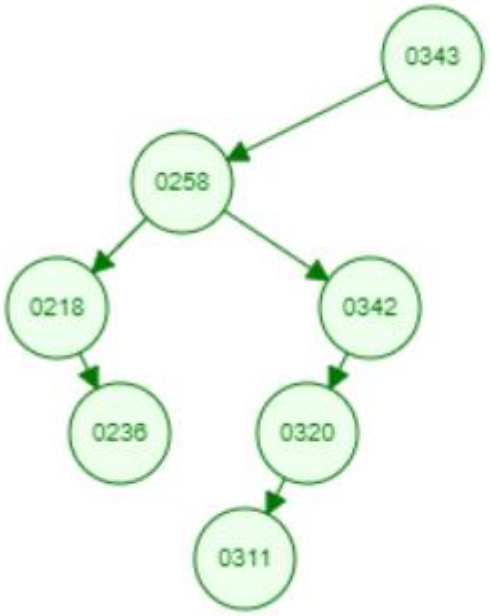
Задание.

Дано бинарное дерево поиска. Проверить является ли оно авл-деревом (т.е. для каждого узла разница высот левого и правого поддерева не больше 1)

Реализуйте функцию check, которая возвращает булевое значение true, если дерево сбалансированное и false в противном случае.

Пример:

Пусть дерево выглядит следующим образом:



Ответ: false

На узлах 342 и 343 разница левого правого поддерева больше 1.

Сигнатура функции на python:

```
def check(root: Node) -> bool:
```

класса:

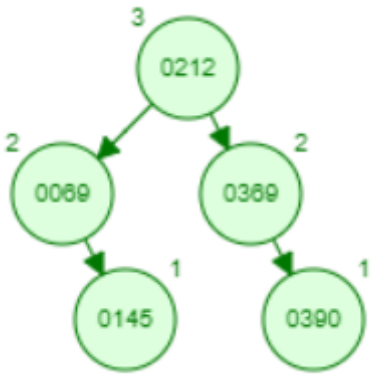
```
class Node:
    def __init__(self, val, left=None, right=None):
        self.val = val
        self.left = left
```

self.right = right

Дано бинарное дерево поиска. Реализуйте функцию diff, которая возвращает минимальную абсолютную разницу между значениями связанных узлов в этом дереве.

Пример:

Пусть дерево выглядит следующим образом:



Посчитаем абсолютную разницу между связанными узлами:

abs(145 - 69) = 76

abs(69 - 212) = 143

abs(212 - 369) = 157

abs(369 - 390) = 21

Минимальная абсолютная разница получилась между узлами 369 и 390 = 21

Ответ: 21.

Ограничения

2 <= N <= 10000

-10000 <= ai <=10000

Сигнатура функции diff на python:

def diff(root: Node) -> int:

класса:

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

```
        self.left = left
        self.right = right
```

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
```

Пример использования функции insert

```
root = None
root = insert(5, root)
root = inesrt(9, root)
```

Выполнение работы.

В начале программы была написана функция *def check(root: Union[Node, None]) -> bool*, которая проверяет, является ли поданное дерево авл-деревом. Внутри оно рекурсивно проходит по каждому *Node'y*, и сравнивает высоты левого и правого поддеревьев. Если разность высот не превышает 1 для всех поддеревьев от всех вершин, то возвращается *True*, если же дерево поданное дерево не сбалансированно, возвращается *false*.

Далее была написана функция *def diff(root: Union[Node, None]) -> int*, которая ищет в дереве и возвращает минимальную абсолютную разницу между узлами поддерева. Внутри рекурсивно вызывается эта же функция для левого и

правого элемента-ребенка от текущего элемента, и вычисляется значение между текущим элементом и каждым ребенком, после чего сравниваются все 4 полученных значения и возвращается минимальное из них.

Далее были реализованы вспомогательные функции для функции *insert()*.

Функция *def height(root: Union[Node, None]) -> int* возвращает высоту поддерева с поданной вершиной, если вершина пустая, то возвращается 0.

Функция *def right_rotate(root: Node) -> Node* выполняет правый поворот АВЛ-дерева с поданной вершиной, после чего изменяет высоты новых поддеревьев и возвращает новую вершину.

Функция *def left_rotate(root: Node) -> Node* выполняет левый поворот АВЛ-дерева с поданной вершиной, после чего изменяет высоты новых поддеревьев и возвращает новую вершину.

Функция *def balance(root: Union[Node, None]) -> int* возвращает разницу высот между левым и правым поддеревом поданной вершины.

Функция *def insert(val, root: Union[Node, None]) -> Node* вставляет новый элемент со значением *val* в АВЛ-дерево. Внутри идет поиск места для вставки как в бинарном дереве: если новое значение меньше текущего, переход к левой ветке, в противном случае, переход к правой. После поднимаясь рекурсивно обратно вверх у вершин изменяются значения *height*, и проверяется *flag = balance()*, если *flag == 2*, нужно выполнить правый поворот, если высота правой ветки левого поддерева меньше либо равна высоте левой ветки левого поддерева, выполняется малый правый поворот, в противном случае выполняется большой правый поворот (вначале малый левый, потом малый правый). Если *flag == -2*, то выполняется левый поворот, если высота левой ветки правого поддерева меньше либо равна высоте правой ветки правого поддерева, то выполняется малый левый поворот, в противном случае выполняется большой левый поворот (вначале малый правый, потом малый левый). В конце возвращается голова дерева.

Далее были созданы функции удаления элемента из АВЛ-дерева по значению, и удаление минимального и максимального элемента.

Функция *def remove(val, root: Union[Node, None]) -> Node* удаляет элемент по поданному значению *val*. Внутри вызывается рекурсивный алгоритм поиска элемента как в бинарном дереве, если элемент найден, то, если правое поддерево найденного элемента пусто, найденный элемент удаляется, вместо него записывается его левое поддерево, в противном случае вызывается поиск минимального элемента в правом поддереве найденного элемента при помощи функции *def find_min(min_val, root: Union[Node, None]) -> (Node, int)*, в которой ищется минимальный элемент, который удаляется из дерева, и функция возвращает вершину этого поддерева и значение минимального элемента. После этого значение удаляемого элемента заменяется на найденное минимальное значение. После этого при рекурсивном подъеме вверх выполняется корректировка высот поддеревьев и балансировка (так же как в insert()).

Функция *def remove_max(root: Union[Node, None]) -> Node* удаляет максимальный элемент из АВЛ-дерева, при помощи функции *def find_max_value(root: Union[Node, None], max_val = 0) -> (Node, int)* находится максимальное значение, которое передается в функцию remove(), и удаляется.

Функция *def remove_min(root: Union[Node, None]) -> Node* удаляет минимальный элемент из АВЛ-дерева, при помощи функции *def find_min_value(root: Union[Node, None], max_val = 0) -> (Node, int)* находится минимальное значение, которое передается в функцию remove(), и удаляется.

Функция *def pre_order(root: Union[Node, None])* создана для вывода дерева в формате: *[left, root, right]*.

Описание пайплайна.

Для создания и заполнения АВЛ-дерева используется функция *insert()*, которая вставляет элементы в дерево и сразу балансирует его при помощи функций *height(), balance(), left_rotate() и right_rotate()*. Для удаления элемента из дерева по значению используется *remove()*, которая так же использует функцию *find_min()* для поиска элемента для замены удаленного. Так же есть

функции *remove_max()* и *remove_min(),* удаляющие максимальный и минимальный элементы соответственно. Внутри они используют функции *find_max_value() и find_min_value()* соответственно а так же функцию *remove()*. Кроме того есть функция *check()* для проверки дерева на сбалансированность и функция *diff()* для поиска минимальной абсолютной разницы между значениями дерева.

Анализ полученных значений.

Также было проведено исследование. Замерено время работы функций поиска, удаления элемента и добавления элемента в начале, середине и конце развернутого связного списка, массива и односвязного списка.

Единицы измерения 10^{-6} с.

Таблица 1 – Результаты исследования

Операция	Набор данных	Время выполнения
Вставка N элементов	10	28,8
	10000	87083,3
	100000	1028877,4 (1с)
Удаление элемента в центре	10	8,1
	10000	17,1
	100000	19,0
Удаление максимального элемента	10	4,7
	10000	15,9
	100000	22,1
Удаление минимального элемента	10	3,81
	10000	13,8
	100000	16,9

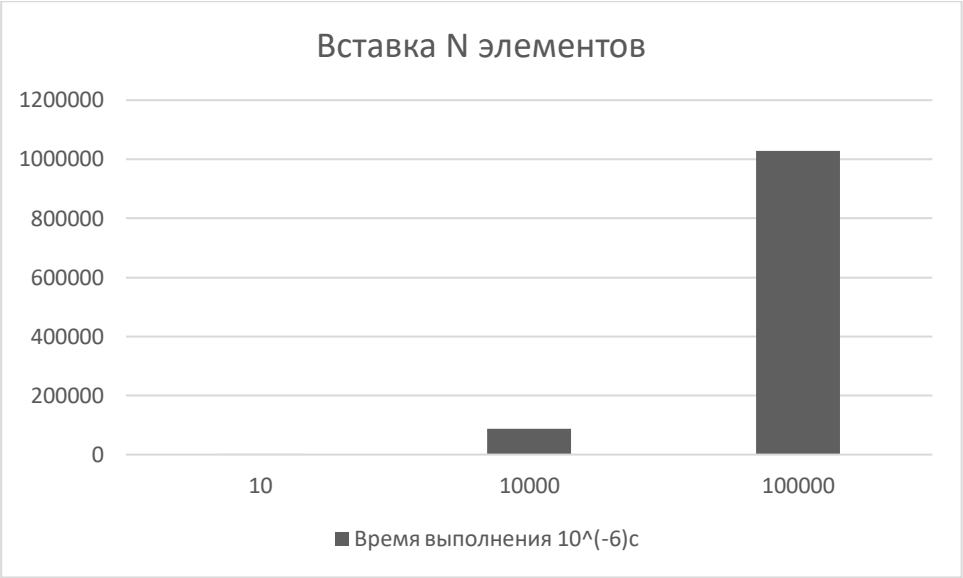


Рисунок 1 – Вставка N элементов

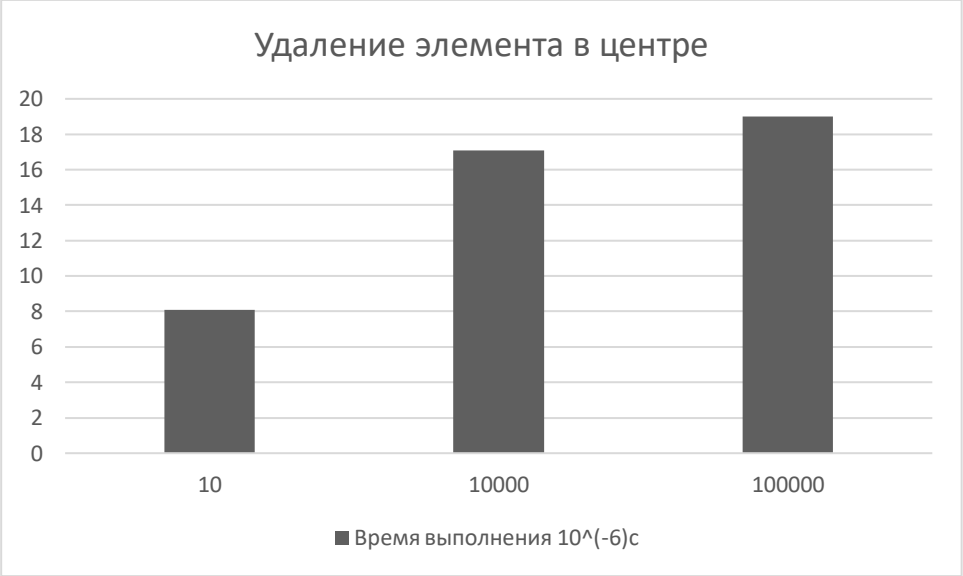


Рисунок 2 – Удаление элемента в центре

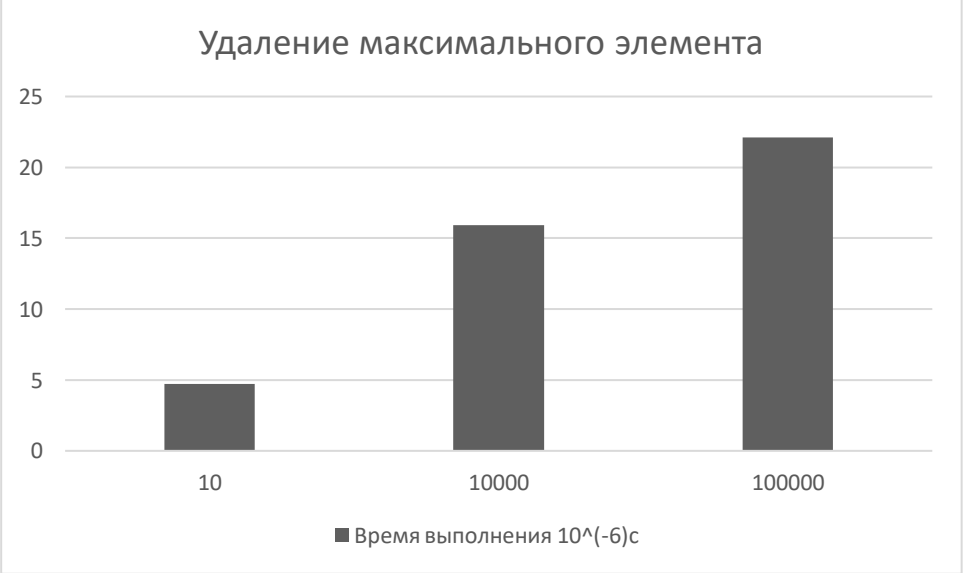


Рисунок 3 – Удаление максимального элемента

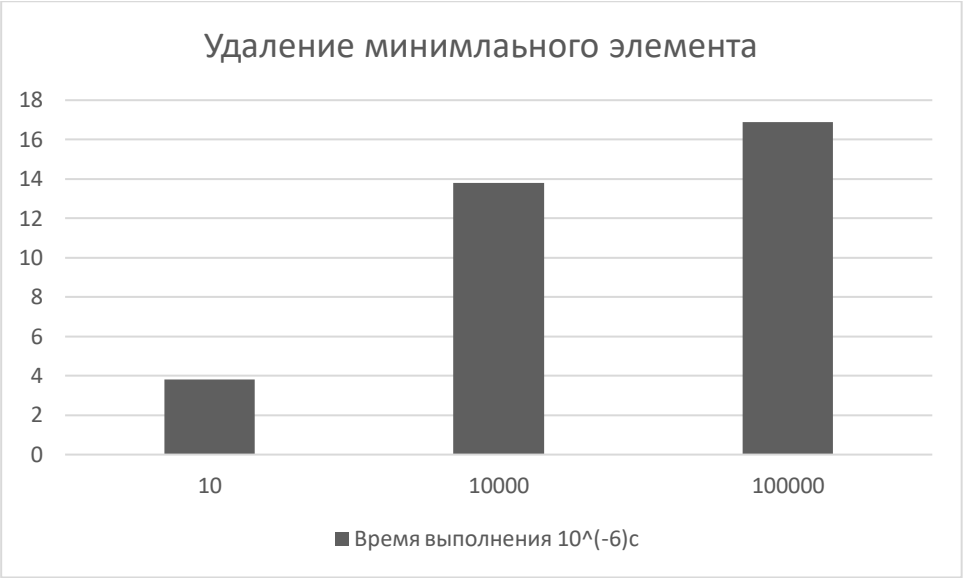


Рисунок 4 – Удаление минимального элемента

Разработанный программный код см. в приложении А.

Выводы.

Была изучена и реализована новая структура данных — АВЛ-дерево, а также реализованы вспомогательные функции, для проверки того, является ли дерево АВЛ-деревом и нахождение минимальной абсолютной разницы между элементами АВЛ-дерева. Проведено исследование скорости его работы.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.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

def height(root: Union[Node, None]) -> int:
    if root == None:
        return 0
    return root.height

def right_rotate(root: Node) -> Node:
    out = root.left
    current = out.right
    out.right = root
    root.left = current

    root.height = 1 + max(height(root.left), height(root.right))
    out.height = 1 + max(height(out.left), height(out.right))

    return out

def left_rotate(root: Node) -> Node:
    out = root.right
    current = out.left
    out.left = root
    root.right = current

    root.height = 1 + max(height(root.left), height(root.right))
    out.height = 1 + max(height(out.left), height(out.right))

    return out

def balance(root: Union[Node, None]) -> int:
    if root == None:
        return 0
    left_height = height(root.left)
    right_height = height(root.right)
    return left_height - right_height

def insert(val, root: Union[Node, None]) -> Node:
    if root == None:
        return Node(val)
    if root.val > val:
        root.left = insert(val, root.left)
    else:
        root.right = insert(val, root.right)
```

```
        root.height = 1 + max(height(root.left), height(root.right))

        flag = balance(root)

        if flag == 2:
            if height(root.left.right) <= height(root.left.left):
                return right_rotate(root)
            else:
                root.left = left_rotate(root.left)
                return right_rotate(root)
        if flag == -2:
            if height(root.right.left) <= height(root.right.right):
                return left_rotate(root)
            else:
                root.right = right_rotate(root.right)
                return left_rotate(root)

        return root

    def pre_order(root: Union[Node, None]):
        if root.left != None:
            left = root.left.val
            pre_order(root.left)
        else:
            left = -1
        if root.right != None:
            right = root.right.val
            pre_order(root.right)
        else:
            right = -1
        print(f'{left}; {root.val}; {right}')

    def find_min(min_val, root: Union[Node, None]) -> (Node, int):
        if root.left != None:
            root.left, min_val = find_min(min_val, root.left)
        elif root.right != None:
            root.right, min_val = find_min(min_val, root.right)
        else:
            min_val = root.val
            root = None
        return root, min_val

    def find_max_value(root: Union[Node, None], max_val = 0) -> (Node,
int):
        if root.right != None:
            root.right, new_max = find_max_value(root.right, max_val)
            max_val = max(new_max, root.val)
        elif root.left != None:
            root.left, new_max = find_max_value(root.left, max_val)
            max_val = max(new_max, root.val)
        else:
            max_val = root.val
        return root, max_val

    def remove_max(root: Union[Node, None]) -> Node:
        root, max_val = find_max_value(root)
        root = remove(max_val, root)
```

```
        return root

    def find_min_value(root: Union[Node, None], min_val = 0) -> (Node,
int):
        if root.left != None:
            root.left, new_min = find_max_value(root.left, min_val)
            min_val = min(new_min, root.val)
        elif root.right != None:
            root.right, new_min = find_max_value(root.right, min_val)
            min_val = min(new_min, root.val)
        else:
            min_val = root.val
        return root, min_val

    def remove_min(root: Union[Node, None]) -> Node:
        root, min_val = find_min_value(root)
        root = remove(min_val, root)
        return root

    def remove(val, root: Union[Node, None]) -> Node:
        if root == None:
            return root
        if val > root.val:
            root.right = remove(val, root.right)
        elif val < root.val:
            root.left = remove(val, root.left)
        else:
            if root.right == None:
                root = root.left
                if root == None:
                    return root
            else:
                root.right,    min_val    =    find_min(root.right.val,
root.right)
                root.val = min_val

        root.height = 1 + max(height(root.left), height(root.right))

        flag = balance(root)

        if flag == 2:
            if height(root.left.right) <= height(root.left.left):
                return right_rotate(root)
            else:
                root.left = left_rotate(root.left)
                return right_rotate(root)
        if flag == -2:
            if height(root.right.left) <= height(root.right.right):
                return left_rotate(root)
            else:
                root.right = right_rotate(root.right)
                return left_rotate(root)

        return root

    def check(root: Union[Node, None]) -> bool:
        balanced = True
        left_height = height(root.left)
```

```
        right_height = height(root.right)
        if abs(left_height - right_height) > 1:
            balanced = False
        if root.left != None:
            flag = check(root.left)
            if flag == False:
                balanced = False
        if root.right != None:
            flag = check(root.right)
            if flag == False:
                balanced = False
        return balanced

    def diff(root: Union[Node, None], max_val = 100000000) -> int:
        min_count = max_val
        left_min_count = min_count
        add_left_count = min_count
        right_min_count = min_count
        add_right_count = min_count
        if root.left != None:
            left_min_count = diff(root.left)
            add_left_count = abs(root.val - root.left.val)
        if root.right != None:
            right_min_count = diff(root.right)
            add_right_count = abs(root.val - root.right.val)

        min_count = min(min_count, left_min_count, right_min_count,
add_left_count, add_right_count)
        return min_count
```