**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 |  | Козьмин Н.В. |
| Преподаватель |  | Шестопалов Р.П. |

Санкт-Петербург

2024

## Цель работы.

Изучить и реализовать на Python структуру АВЛ-деревьев через использование узлов, связанных друг с другом. Выполнить проверку на соответствие АВЛ-дереву, визуализацию, балансировку изменённого узла, добавление и удаление элементов. Сравнить время и количество операций, необходимых для реализованных операций, с теоретическими оценками.

## Задание.

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.

## Основные теоретические положения.

АВЛ-дерево — это бинарное дерево поиска, сбалансированное по высоте так, что разница между высотами левого и правого поддерева каждого узла не более 1.

АВЛ-деревья (как и другие деревья поиска с балансировкой) используются так как все операции над деревьями поиска зависят от высоты дерева.

Выставляя дополнительные ограничения на высоту, мы тем самым сокращаем время каждой операции поиска, жертвуя дополнительными затратами на балансировку при вставке и удалении.

## Выполнение работы.

Описание структуры кода.

Класс Node уже был дан в лабораторной работе. Его поля очевидны (значение, левый узел, правый узел, высота, зависящая от дочерних элементов)

Реализованные объекты:

* Функция find\_height\_and\_balance подсчитывает высоту и проверяет сбалансированность, начиная от каждого узла дерева. Функционал не использует уже имеющуюся длину узлов, что позволяет нам независимо проверить правильность выполнения алгоритмов по работе с деревьями.
* Функция check\_avl\_tree, используя второе значение кортежа, возвращаемого find\_height\_and\_balance, даёт только ответ на то, является ли root узлом avl-дерева.
* Функция diff является дополнением в курсе для лабораторной работы, поэтому она тоже приведена здесь. Она вычисляет минимальную разницу значений между связанными узлами, используя обход в длину.
* Функция height является вспомогательной и просто, либо берёт значение узла при его наличии, либо возвращает 0.
* Функции small\_left\_turn и small\_right\_turn делают повороты ветвей узла при их вызове согласно описанию из вики.
* Сама функция балансировки balancing использует функции поворота при соблюдениях определённых условий, которые также согласуются с вики. Также используются, так называемые, большие повороты, как комбинация двух стандартных (маленьких).
* Функция вставки также через обход в длину находит место и вставляет значение в него (очевидно, с созданием узла), а затем поднимаясь вверх по узлам, балансирует их. Так как внутри балансировки производится пересчёт высоты корня, после неё пересчёт высоты не нужен.
* Функция delete\_max проходит до крайнего правого узла и ставит на его место текущий левый. При этом во время обратного подъёма балансирует каждый узел. Также возвращается удалённое значение.
* Функция delete\_min выполняет те же действия, что и delete\_max с заменой сторон на противоположные.
* Функция delete доходит до значения, которое нужно удалить (если его нет, завершает работу), а затем берёт следующее значение по величине из правой ветки (через функцию delete\_min с попутной балансировкой) и заменяет на него значение, которое нужно удалить. Затем возвращаясь, также балансирует каждый узел
* Функция in\_order используется для визуализации дерева в одну строку, рекурсивно строя его из всех значений левых узлов, значения текущего и всех значений правых (возвращает значение).
* Функция visualize используется для визуализации дерева в ряд строк, показывая результат по аналогии с ответом от проверяющей системы в случае ошибки (выводит значения). Для обработки дерева также проходит в длину, так как эта реализация проще.
* Функция generate\_avl\_tree из строки, содержащей числа через пробел, делает дерево, используя вставку (возвращает корневой узел).

Описание пайплайна.

* По умолчанию данные не считываются, так как суть лабораторной работы в предоставлении функционала для разработки, то есть ряда функций. Значения принимаются в поочерёдной вставке элементов или через функцию generate\_avl\_tree. Самостоятельно создавать узлы и связывать их друг с другом нельзя, так как это приведёт к построению не бинарного дерева поиска.
* Предобработкой (как и постобработкой) является выполнение балансировки для каждого необходимого узла во время добавления и удаления элементов.
* Основная работа происходит при обновлении (добавлении и удалении элементов) и визуализации АВЛ-дерева.
* Вывод данных происходит в основной поток при вызове visualize или при выводе результата вызова in\_order.

Разработанный программный код см. в приложении А.

Результаты тестирования см. в приложении Б.

## Анализ полученных значений.

Результаты работы структуры представлены на рисунках 1-5.

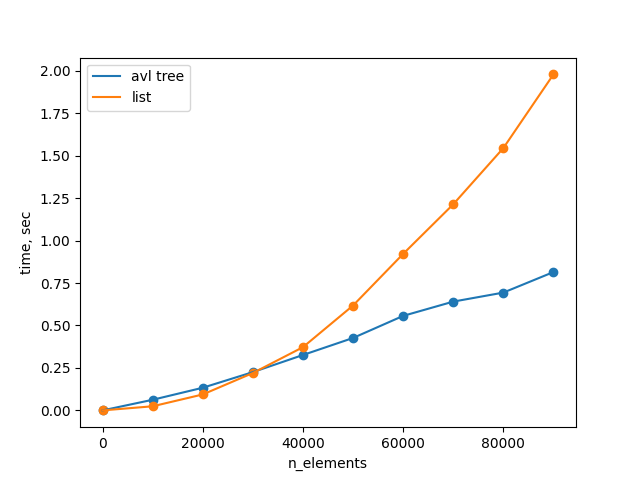


Рисунок 1 – Вставка в худшем случае

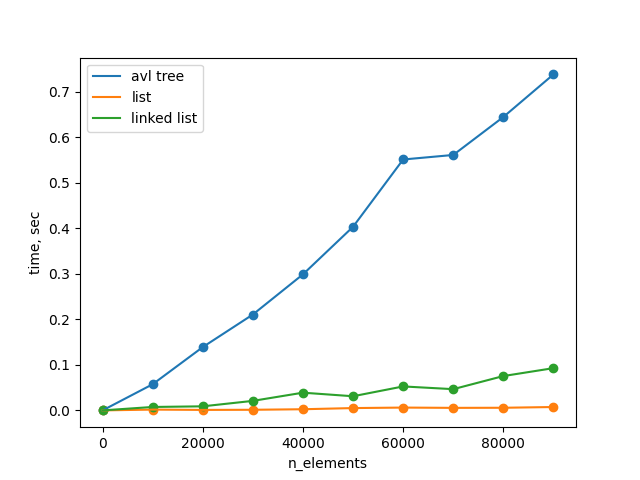


Рисунок 2 – Вставка в лучшем случае

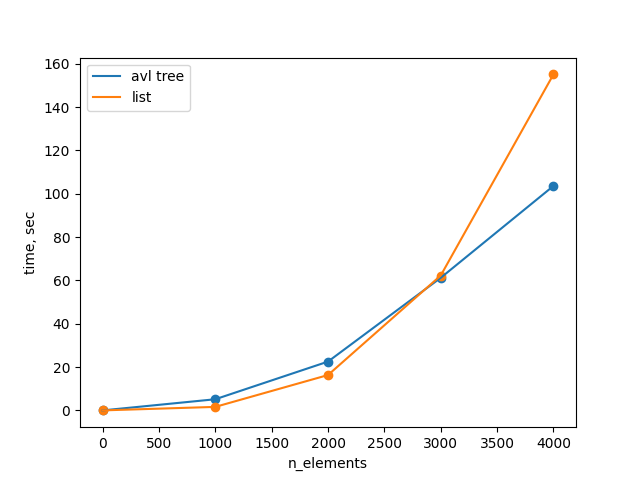


Рисунок 3 – Пересоздание структуры и сортировка в список

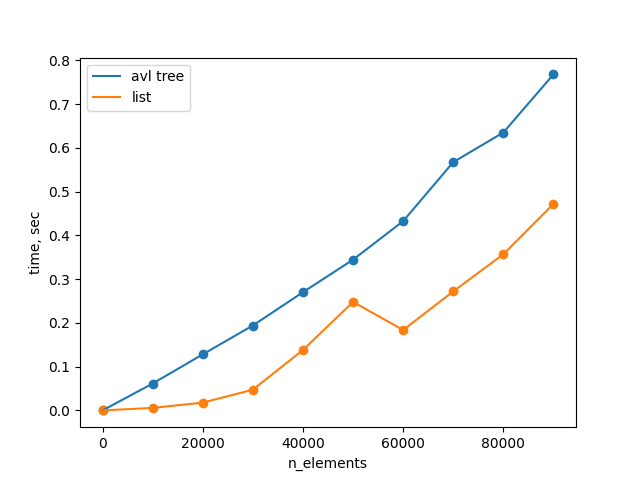


Рисунок 4 – Удаление в худшем случае

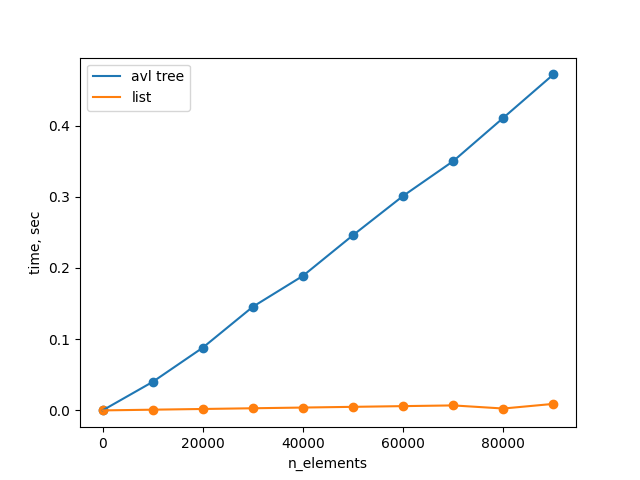


Рисунок 5 – Удаление в лучшем случае

Важно учесть, что n\_elements характеризует не только количество элементов в структуре, но и количество применения самих операций для одной и той же структуры вместо принятого одного раза для упрощения и наглядности. Полученные данные можно описать следующим образом: АВЛ-дерево работает стабильнее по времени при вставках и выигрывает при определенных обстоятельствах при потребности в сортировки. Также можно предполагать, что поиск тоже будет быстрее, нежели в других указанных структурах при анализе.

## Выводы.

В проделанной работе были изучены АВЛ-деревья. Для них были реализованы добавление, удаление и визуализация элементов на Python вместе с сопутствующими функциями. Дополнительно была измерена скорость работы реализованной структуры на выборках разного объемам и разброса значений, произведён анализ полученных данных.

# Приложение А Исходный код программы

Название файла: src/main.py

class Node:

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

self.val = val

self.left: Node | None = left

self.right: Node | None = right

self.height: int = 1

def find\_height\_and\_balance(root: Node | None) -> tuple:

if root == None:

return 0, True

left = find\_height\_and\_balance(root.left)

right = find\_height\_and\_balance(root.right)

height = max(left[0], right[0])+1

is\_balanced = abs(left[0] - right[0]) < 2

return height, left[1] and right[1] and is\_balanced

# Checking the whole tree

def check\_avl\_tree(root: Node | None) -> bool:

return find\_height\_and\_balance(root)[1]

def diff(root: Node | None) -> int:

if root == None:

return float("+inf")

diffs = [float("+inf")]

if root.left != None:

diffs.append(abs(root.val - root.left.val))

diffs.append(diff(root.left))

if root.right != None:

diffs.append(abs(root.right.val - root.val))

diffs.append(diff(root.right))

return min(diffs)

def height(root: Node | None) -> int:

return root.height if root != None else 0

def small\_left\_turn(root: Node) -> Node:

tmp\_root = root

root = root.right

tmp\_root.right = root.left

root.left = tmp\_root

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

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

return root

def small\_right\_turn(root: Node) -> Node:

tmp\_root = root

root = root.left

tmp\_root.left = root.right

root.right = tmp\_root

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

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

return root

def balancing(root: Node | None) -> Node:

if root == None:

return None

left\_height = height(root.left)

right\_height = height(root.right)

# Left turn

if left\_height - right\_height == -2:

# Big turn

if height(root.right.left) - height(root.right.right) > 0:

root.right = small\_right\_turn(root.right)

return small\_left\_turn(root)

# Right turn

elif left\_height - right\_height == 2:

# Big turn

if height(root.left.left) - height(root.left.right) < 0:

root.left = small\_left\_turn(root.left)

return small\_right\_turn(root)

else:

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

return root

def insert(val, root: Node | None) -> Node:

if root == None:

return Node(val)

if val <= root.val:

if root.left == None:

root.left = Node(val)

else:

root.left = insert(val, root.left)

else:

if root.right == None:

root.right = Node(val)

else:

root.right = insert(val, root.right)

root = balancing(root)

return root

def delete\_max(root: Node | None) -> Node | None:

if root == None:

return None, None

if root.right != None:

root.right, del\_val = delete\_max(root.right)

root = balancing(root)

else:

del\_val = root.val

root = root.left

return root, del\_val

def delete\_min(root: Node | None) -> Node | None:

if root == None:

return None, None

if root.left != None:

root.left, del\_val = delete\_min(root.left)

root = balancing(root)

else:

del\_val = root.val

root = root.right

return root, del\_val

def delete(val, root: Node | None) -> Node | None:

if root == None:

return None

if val < root.val:

root.left = delete(val, root.left)

elif val > root.val:

root.right = delete(val, root.right)

else:

if root.right == None:

root = root.left

elif root.left == None:

root = root.right

else:

root.right, root.val = delete\_min(root.right)

root = balancing(root)

return root

def in\_order(root: Node | None) -> str:

if root == None:

return ""

rslt = []

left = in\_order(root.left)

if left != "":

rslt.append(left)

right = in\_order(root.right)

rslt.append(str(root.val))

if right != "":

rslt.append(right)

return " ".join(rslt)

def visualize(root: Node) -> None:

if root == None:

print("x x x, height: 0")

return

if root.left != None:

left = root.left.val

else:

left = "x"

if root.right != None:

right = root.right.val

else:

right = "x"

print(f"{left} {root.val} {right}, height: {root.height}")

if root.left != None:

visualize(root.left)

if root.right != None:

visualize(root.right)

def generate\_avl\_tree(line: str) -> Node:

root = None

arr = list(map(int, line.split()))

for i in arr:

root = insert(i, root)

return root

Название файла: tests/efficiency\_test.py

"""This file should be placed along with the executable file."""

import matplotlib.pyplot as plt

from random import randint

from main import \*

import time

# Node for LinkedList

class Node\_ll:

def \_\_init\_\_(self, value = None, next = None):

self.value = value

self.next = next

class LinkedList:

def \_\_init\_\_(self):

self.first = None

self.last = None

self.length = 0

def \_\_str\_\_(self):

if self.first != None:

current = self.first

out = 'LinkedList [\n' +str(current.value) +'\n'

while current.next != None:

current = current.next

out += str(current.value) + '\n'

return out + ']'

return 'LinkedList []'

def clear(self):

self.\_\_init\_\_()

def \_\_getitem\_\_(self, index):

cur = self.first

while cur != None:

if index == 0:

return cur.value

index -= 1

cur = cur.next

raise IndexError("linked list index out of range")

# add to end of LinkedList

def add(self, x):

self.length+=1

if self.first == None:

# self.first and self.last will point to the same memory location

self.last = self.first = Node\_ll(x, None)

else:

# here, already to different ones, because the assignment occurred

self.last.next = self.last = Node\_ll(x, None)

def InsertNth(self,i,x):

if self.first == None:

self.last = self.first = Node\_ll(x, None)

self.length += 1

return

if i == 0:

self.first = Node\_ll(x,self.first)

self.length += 1

return

curr=self.first

count = 0

while curr != None:

count+=1

if count == i:

curr.next = Node\_ll(x,curr.next)

self.length += 1

if curr.next.next == None:

self.last = curr.next

break

curr = curr.next

def Del(self,i):

if (self.first == None):

return

curr = self.first

count = 0

if i == 0:

self.first = self.first.next

self.length -= 1

return

while curr != None:

if count == i:

if curr.next == None:

self.last = curr

old.next = curr.next

self.length -= 1

break

old = curr

curr = curr.next

count += 1

if \_\_name\_\_ == "\_\_main\_\_":

set\_n\_elements = []

avl\_tree\_times = []

list\_times = []

ll\_times = []

for n\_elements in range(0, 100000, 10000):

print(f"Check {n\_elements} elements...")

set\_n\_elements.append(n\_elements)

a = [randint(-1000000, 1000000) for \_ in range(n\_elements)]

count = 2

avl\_tree\_local\_times = []

list\_local\_times = []

ll\_local\_times = []

for \_ in range(count):

root = generate\_avl\_tree(' '.join(map(str, range(n\_elements))))

start = time.time()

for i in range(n\_elements):

root = delete(i, root)

end = time.time() - start

avl\_tree\_local\_times.append(end)

lst = list(a)

start = time.time()

for i in range(n\_elements):

del lst[len(lst)-1] #len(lst)

end = time.time() - start

del lst

list\_local\_times.append(end)

# ll = LinkedList()

# for i in range(n\_elements):

# ll.add(a[i])

# start = time.time()

# for i in range(n\_elements):

# ll.Del(i) #ll.length

# end = time.time() - start

# del ll

# ll\_local\_times.append(end)

end = sum(avl\_tree\_local\_times) / count

avl\_tree\_times.append(end)

end = sum(list\_local\_times) / count

list\_times.append(end)

# end = sum(ll\_local\_times) / count

# ll\_times.append(end)

plt.scatter(set\_n\_elements, avl\_tree\_times)

plt.plot(set\_n\_elements, avl\_tree\_times, label='avl tree')

plt.scatter(set\_n\_elements, list\_times)

plt.plot(set\_n\_elements, list\_times, label='list')

# plt.scatter(set\_n\_elements, ll\_times)

# plt.plot(set\_n\_elements, ll\_times, label='linked list')

plt.xlabel("n\_elements")

plt.ylabel("time, sec")

plt.legend()

plt.show()

Название файла: tests/ functionality\_test.py

"""This file should be placed along with the executable file."""

import pytest

from main import \*

@pytest.mark.parametrize('root',

[

(None),

(Node(15)),

(generate\_avl\_tree("-34 45 66")),

(generate\_avl\_tree("758 -98 457 189 434 802 5 5 -561 92")),

])

def test\_generate\_avl\_tree(root):

assert check\_avl\_tree(root) == True

@pytest.mark.parametrize('root, del\_vals, expected\_inorder',

[

(None, [5], ""),

(Node(15), [15], ""),

(generate\_avl\_tree("-34 45 66"), [-34, 45], "66"),

(

generate\_avl\_tree("758 -98 457 189 434 802"),

[758, 457, 500, 802, 457],

"-98 189 434"

),

])

def test\_delete\_vals(root, del\_vals, expected\_inorder):

for i in del\_vals:

root = delete(i, root)

assert check\_avl\_tree(root) == True

assert in\_order(root) == expected\_inorder

# Приложение Б Тестирование

Таблица 1 - Примеры тестовых случаев

|  |  |  |  |
| --- | --- | --- | --- |
| № п/п | Входные данные | Выходные данные | Комментарии |
|  | None | True | ОК |
|  | Node(15) | True | ОК |
|  | generate\_avl\_tree("-34 45 66") | True | ОК |
|  | generate\_avl\_tree("758 -98 457 189 434 802 5 5 -561 92") | True | ОК |
|  | None, [5] | "" | ОК |
|  | Node(15), [15] | "" | ОК |
|  | generate\_avl\_tree("-34 45 66"), [-34, 45] | "66" |  |
|  | generate\_avl\_tree("758 -98 457 189 434 802"), [758, 457, 500, 802, 457] | "-98 189 434" | ОК |