**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: АВЛ-деревья.

|  |  |  |
| --- | --- | --- |
| Студент гр. 3382 |  | Копасова К. А. |
| Преподаватель |  | Иванов Д. В. |

Санкт-Петербург

2024

**Цель работы**

Изучить АВЛ-деревья, различные балансировки и создать программу на языке программирования Python, где будут реализовано АВЛ-дерево с методами вставки узлов и удаления узлов (максимального, минимального и любого). Также нужно сравнить время и количество операция, необходимых для реализованных методов, с теоретическими оценками.

**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:

1. реализовать функции удаления узлов: любого, максимального и минимального
2. 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**

Для начала разделим нашу задачу на два файла: в первом файле main.py будет реализовано АВЛ-дерево.

Во втором файле test.py будут находится все тесты и вычисления для анализа работы методов .

Начнем с файла main.py – опишу все функции и классы данного файла.

Main.py:

Программа включает классы Node и AVLTree, а также функции для различных операций над деревом, включая вставку и удаление узлов с автоматической балансировкой через вращения.

1. Класс Node - данный класс представляет собой узел АВЛ-дерева. Конструктор \_\_init\_\_ принимает значение узла и необязательные параметры для его левого и правого потомков. Каждый узел хранит свое значение, ссылки на левого и правого потомков, а также текущую высоту узла, которая используется для вычисления баланса и определения необходимости балансировки.
2. Класс AVLTree - данный класс представляет само АВЛ-дерево. В конструкторе \_\_init\_\_ создается дерево с корнем, равным None, т. е. дерево изначально пусто. Класс включает методы для вставки и удаления узлов, поиска минимального и максимального значения, а также вспомогательные функции для балансировки дерева.
   1. Метод get\_min\_val\_node - данный метод принимает узел дерева и возвращает узел с минимальным значением, проходя по левым потомкам.
   2. Метод get\_max\_val\_node - данный метод находит узел с максимальным значением, проходя по правым потомкам.
   3. Метод insert - данный метод добавляет элемент в дерево. Если корень дерева отсутствует, новый элемент становится корнем. Иначе вызывается рекурсивный метод \_insert.
   4. Метод \_insert - данный метод принимает значение и узел дерева, рекурсивно вставляет значение в дерево и балансирует его. Если новое значение меньше текущего, оно добавляется слева, иначе — справа. После вставки обновляется высота узла, затем вычисляется его баланс. В зависимости от баланса и значений выполняются малые (левое или правое) или большие (левое или правое) вращения для поддержания сбалансированности.
   5. Метод remove - данный метод удаляет узел с указанным значением из дерева. Если дерево непусто, вызывается метод \_remove для рекурсивного удаления.
   6. Метод \_remove - данный метод осуществляет рекурсивное удаление узла, сохраняя балансировку. Если удаляемый узел не имеет потомков или имеет одного потомка, его место занимает потомок. Если узел имеет двух потомков, находит узел с минимальным значением в правом поддереве, присваивает его значение удаляемому узлу и удаляет его из правого поддерева. Затем обновляет высоту узла и выполняет балансировку с учётом новых значений.
   7. Методы remove\_min и remove\_max удаляют узлы с минимальным и максимальным значениями соответственно. Каждый метод вызывает соответствующую рекурсивную функцию (\_remove\_min или \_remove\_max), которая удаляет крайний левый или правый узел. После удаления высота и баланс узлов обновляются с помощью аналогичных методов.
3. Функция mini\_right\_rotate выполняет малое правое вращение. Берется левый потомок текущего узла, который становится новым корнем для его поддерева. Обновляются ссылки и высоты узлов, обеспечивая восстановление баланса.
4. Функция mini\_left\_rotate выполняет малое левое вращение. Берется правый потомок текущего узла, который становится новым корнем для его поддерева. Аналогично, обновляются ссылки и высоты узлов, восстанавливая баланс.
5. Функция max\_left\_rotate выполняет большое левое вращение. Оно состоит из двух шагов: сначала малое левое вращение для левого поддерева, затем малое правое вращение для текущего узла, что позволяет корректировать баланс в более сложных ситуациях.
6. Функция max\_right\_rotate выполняет большое правое вращение, аналогично состоящее из малого правого вращения для правого поддерева, а затем малого левого вращения для узла.
7. Функция get\_height возвращает высоту узла, которая равна 0 для пустых узлов, используется для вычисления баланса.
8. Функция update\_height обновляет высоту узла после выполнения балансировки. Она вычисляет максимальную высоту между левым и правым поддеревом, прибавляя единицу для текущего узла.
9. Функция get\_balance определяет разницу высот между левым и правым поддеревом узла. Если баланс положительный, то левое поддерево выше правого, и наоборот. Значение баланса используется для определения необходимости выполнения вращений.
10. Функция breadth\_first\_search используется для визуализации дерева через библиотеку graphviz. Принимает корень дерева, объект dot для визуализации и шаг текущей итерации. Функция создает граф, обходя дерево в ширину. Для каждого узла создается графический элемент с указанием левого и правого потомков, а также направления к ним, позволяя просматривать структуру дерева в процессе выполнения программы.
11. Функция main — основная функция, управляющая вставкой и удалением узлов. Создается экземпляр дерева, затем через объект graphviz.Digraph создаются графы для операций вставки и удаления узлов. После этого программа ожидает ввода узлов для вставки, добавляет их в дерево и визуализирует результат на каждом шаге. Аналогично происходит удаление узлов. Визуализация сохраняется в PDF-файлах, демонстрируя процесс добавления и удаления элементов.

В файле test.py находятся дополнительные функции, помогающие построить графики для анализа производительности вставки и удаления в АВЛ-деревьях в зависимости от количества элементов в массиве и «сложности» подаваемых данных (отсортированные значения, любые значения и значения, отсортированные в обратном порядке) . В данном файле была создана функция, которая генерирует данные в зависимости от их количества и сложности. После этого программа пропускает все значения (всего 9) через вставку и удаление и считает время выполения для каждого метода соответственно. С помощью библиотек matplotlib и pandas был построен график зависимости.

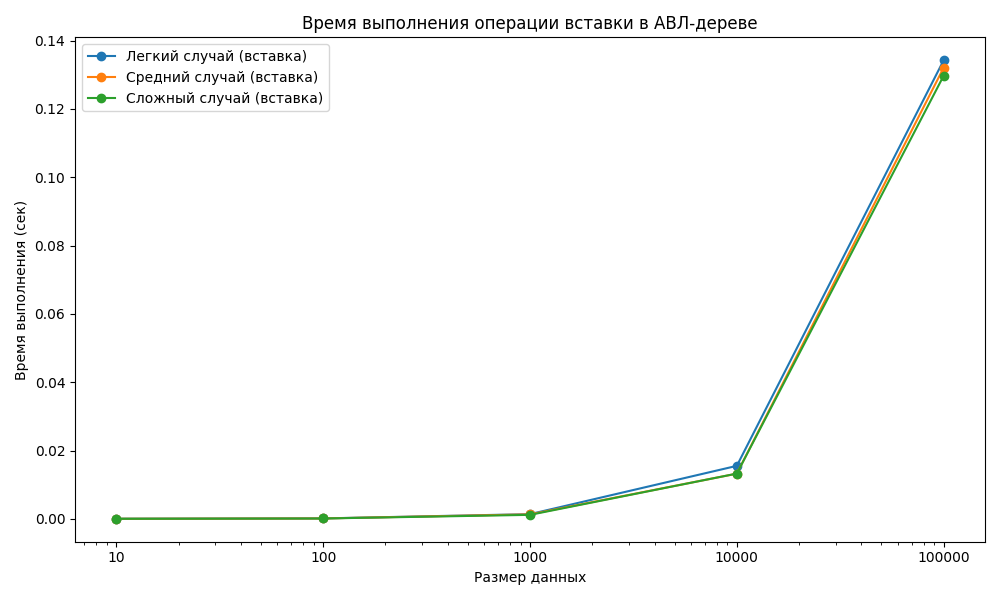
Код программ находится в Приложении А.

**Тестирование**

Таблица 1 — Результаты тестирования основного кода программы:

|  |  |  |  |
| --- | --- | --- | --- |
| №п/п | Входные данные | Выходные данные | Комментарии |
| 1 | 5 6 4 2 8 5 56  5 6 4 | OK | OK |
| 2 | 17 58 465 1213 5 545 6515 4  5 4 862 64 1 89 | OK | OK |
| 3 | 1 2 5 7 -4 -8 -9 -8 7 4 51 654 684 124 548 45 7 4 -4 8 -54 -565 -10 12 13 18 5 95 87 5 -4 -56  1 2 4 8 6 15 26 48 67 845 2 458 65 8 12 35 | OK | OK |

**Анализ полученных данных**

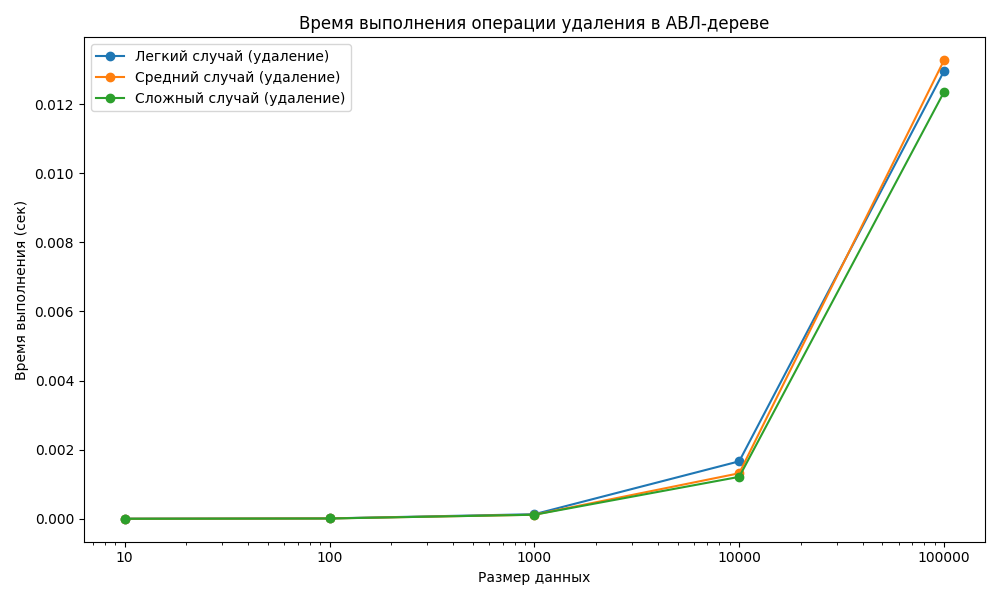
После анализа работы метода вставки в АВЛ-дерево получим следующий график:

Из графика видно, что, несмотря на различное количество данных, используемых в методе, прослеживается одинаковая тенденция - чем больше элементов, тем больше времени требуется на вставку элементов, что логично.

Теоретически: сложность операции вставки в АВЛ-дерево равна O(log n) благодаря поддержанию баланса на каждом уровне. В любых случаях вставка требует поиска по высоте дерева и балансировки.

Практически: график показывает кривую, которая асимптотически схожа с графиком log n, но, т. к. исследование происходило с данными, которые имеют большой разрыв между собой (10000 и 100000), график имеет резкий скачок.

После анализа работы метода удаления в АВЛ-дереве получим следующий график:



Аналогично случаю вставки, время выполнение метода удаления увеличивается с ростом числа данных.

Теоретически: сложность операции удаления в АВЛ-дереве также равна O(log n) из-за необходимости поиска элемента, а затем балансировки дерева. Практически: график удаления показывает более выраженные различия между сложными и простыми случаями по сравнению с вставкой. Это объясняется тем, что удаление узла с двумя потомками требует дополнительной операции поиска наименьшего элемента в правом поддереве. Также, как и при вставке, виден рост времени выполнения, который асимптотически схож с графиком log n, но из-за сильной разницы между количеством данных (10000, 100000) имеет сильный скачок.

## Выводы

В ходе лабораторной работы удалось изучить АВЛ-деревья, различные балансировки и создать программу на языке программирования Python, где реализовано АВЛ-дерево с методами вставки узлов и удаления узлов (максимального, минимального и любого). Также удалось провести анализ сравнения времени выполнения программы от количества операций, необходимых для реализованных методов, с теоретическими оценками.

# ПРИЛОЖЕНИЕ А **ИСХОДНЫЙ КОД ПРОГРАММЫ**

Файл main.py:

"""Реализация АВЛ-дерева"""

import graphviz

class Node:

    """Класс для узла АВЛ-дерева"""

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

        """Конструктор узла АВЛ-дерева."""

        self.val = val

        self.left: Node | None = left

        self.right: Node | None = right

        self.height: int = 1

class AVLTree:

    """Класс для АВЛ-дерева."""

    def \_\_init\_\_(self) -> None:

        """Конструктор АВЛ-дерева."""

        self.root = None

    def get\_min\_val\_node(self, node: Node):

        """Поиск узла с минимальным значением."""

        while node.left:

            node = node.left

        return node

    def get\_max\_val\_node(self, node: Node):

        """Поиск узла с максимальным значением."""

        while node.right:

            node = node.right

        return node

    def insert(self, val):

        """Вставка элемента в дерево."""

        if not self.root:

            self.root = Node(val)

        else:

            self.root = self.\_insert(val, self.root)

    def \_insert(self, val, node: Node | None):

        """Рекурсивная вставка и балансировка."""

        if not node:

            return Node(val)

        if val < node.val:

            if node.left:

                node.left = self.\_insert(val, node.left)

        else:

            if node.right:

                node.right = self.\_insert(val, node.right)

        update\_height(node)

        balance = get\_balance(node)

        if balance > 1 and val < node.left.val:

            return mini\_right\_rotate(node)

        if balance < -1 and val > node.right.val:

            return mini\_left\_rotate(node)

        if balance > 1 and val > node.left.val:

            return max\_left\_rotate(node)

        if balance < -1 and val < node.right.val:

            return max\_right\_rotate(node)

        return node

    def remove(self, val):

        """Удаление элемента из дерева."""

        if self.root:

            self.root = self.\_remove(val, self.root)

    def \_remove(self, val, node: Node | None):

        """Рекурсивное удаление и балансировка дерева."""

        if not node:

            return node

        if val < node.val:

            if node.left:

                node.left = self.\_remove(val, node.left)

        elif val > node.val:

            if node.right:

                node.right = self.\_remove(val, node.right)

        else:

            # узел без детей или с одним ребенком

            if not node.left:

                return node.right

            elif not node.right:

                return node.left

            # узел с двумя детьми

            temp = self.get\_min\_val\_node(node.right)

            node.val = temp.val

            node.right = self.\_remove(temp.val, node.right)

        update\_height(node)

        balance = get\_balance(node)

        if balance > 1 and get\_balance(node.left) >= 0:

            return mini\_right\_rotate(node)

        if balance < -1 and get\_balance(node.right) <= 0:

            return mini\_left\_rotate(node)

        if balance > 1 and get\_balance(node.left) < 0:

            return max\_left\_rotate(node)

        if balance < -1 and get\_balance(node.right) > 0:

            return max\_right\_rotate(node)

        return node

    def remove\_min(self):

        """Удаление узла с минимальным значением."""

        if self.root:

            self.root = self.\_remove\_min(self.root)

    def \_remove\_min(self, node: Node):

        """Рекурсивное удаление узла с минимальным значением."""

        if not node.left:

            return node.right  # узел с минимальным значением

        node.left = self.\_remove\_min(node.left)

        update\_height(node)

        balance = get\_balance(node)

        if balance > 1 and get\_balance(node.left) >= 0:

            return mini\_right\_rotate(node)

        if balance < -1 and get\_balance(node.right) <= 0:

            return mini\_left\_rotate(node)

        if balance > 1 and get\_balance(node.left) < 0:

            return max\_left\_rotate(node)

        if balance < -1 and get\_balance(node.right) > 0:

            return max\_right\_rotate(node)

        return node

    def remove\_max(self):

        """Удаление узла с максимальным значением."""

        if self.root:

            self.root = self.\_remove\_max(self.root)

    def \_remove\_max(self, node: Node):

        """Рекурсивное удаление узла с максимальным значением."""

        if not node.right:

            return node.left  # узел с максимальным значением

        node.right = self.\_remove\_max(node.right)

        update\_height(node)

        balance = get\_balance(node)

        if balance > 1 and get\_balance(node.left) >= 0:

            return mini\_right\_rotate(node)

        if balance < -1 and get\_balance(node.right) <= 0:

            return mini\_left\_rotate(node)

        if balance > 1 and get\_balance(node.left) < 0:

            return max\_left\_rotate(node)

        if balance < -1 and get\_balance(node.right) > 0:

            return max\_right\_rotate(node)

        return node

def mini\_right\_rotate(b: Node) -> Node:

    """Малый правый поворот."""

    a = b.left

    temp = a.right

    a.right = b

    b.left = temp

    update\_height(b)

    update\_height(a)

    return a

def mini\_left\_rotate(a: Node) -> Node:

    """Малый левый поворот."""

    b = a.right

    temp = b.left

    b.left = a

    a.right = temp

    update\_height(a)

    update\_height(b)

    return b

def max\_left\_rotate(node: Node) -> Node:

    """Большой левый поворот."""

    node.left = mini\_left\_rotate(node.left)

    return mini\_right\_rotate(node)

def max\_right\_rotate(node: Node) -> Node:

    """Большой правый поворот."""

    node.right = mini\_right\_rotate(node.right)

    return mini\_left\_rotate(node)

def get\_height(node: Node | None) -> int:

    """Определение высоты узла."""

    return node.height if node else 0

def update\_height(node: Node):

    """Обновление высоты после балансировки."""

    node.height = max(get\_height(node.left), get\_height(node.right)) + 1

def get\_balance(node: Node) -> int:

    """Определение разницы высот для балансировки."""

    return get\_height(node.left) - get\_height(node.right) if node else 0

def breadth\_first\_search(root, dot, step):

    """Визуализация дерева через Graphviz."""

    queue = [(root, "root")]

    dot.node(f"{step}\_{root.val}", label=f"{root.val} ")

    while queue:

        tmp\_queue = []

        for element in queue:

            if element[0].left:

                dot.node(f"{step}\_{element[0].left.val}", label=str(element[0].left.val))

                dot.edge(f"{step}\_{element[0].val}", f"{step}\_{element[0].left.val}", label="left")

                tmp\_queue.append((element[0].left, "left"))

            if element[0].right:

                dot.node(f"{step}\_{element[0].right.val}", label=str(element[0].right.val))

                dot.edge(f"{step}\_{element[0].val}", f"{step}\_{element[0].right.val}", label="right")

                tmp\_queue.append((element[0].right, "right"))

        queue = tmp\_queue

def main():

    """Основная функция для вставки и удаления узлов из АВЛ-дерева."""

    avl\_tree = AVLTree()

    dot\_insert = graphviz.Digraph(comment="AVLTree - Insert")

    dot\_remove = graphviz.Digraph(comment="AVLTree - Remove")

    # Вставка узлов

    nodes\_add = list(map(int, input().split()))

    for step, node in enumerate(nodes\_add, 1):

        avl\_tree.insert(node)

        breadth\_first\_search(avl\_tree.root, dot\_insert, step)

    dot\_insert.render("AddElemToAVLTree", format="pdf", cleanup=True)

    # Удаление узлов

    breadth\_first\_search(avl\_tree.root, dot\_remove, "initial")

    nodes\_remove = list(map(int, input().split()))

    for step, node in enumerate(nodes\_remove, 1):

        avl\_tree.remove(node)

        breadth\_first\_search(avl\_tree.root, dot\_remove, f"remove\_any\_{node}")

    dot\_remove.render("RemoveElemToAVLTree", format="pdf", cleanup=True)

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

    main()

Файл test.py:

import time

import random

import pandas as pd

import matplotlib.pyplot as plt

from main import AVLTree

def data\_generation\_avl():

# данные отсортированы

easy\_arrs = [

list(range(1, 11)),

list(range(1, 101)),

list(range(1, 1001)),

list(range(1, 10001)),

list(range(1, 100001))

]

# данные случайные

normal\_arrs = [

[random.randint(0, 10000) for \_ in range(10)],

[random.randint(0, 10000) for \_ in range(100)],

[random.randint(0, 10000) for \_ in range(1000)],

[random.randint(0, 10000) for \_ in range(10000)],

[random.randint(0, 10000) for \_ in range(100000)]

]

# данные отсортированы в обратном порядке

hard\_arrs = [

list(range(10, 0, -1)),

list(range(100, 0, -1)),

list(range(1000, 0, -1)),

list(range(10000, 0, -1)),

list(range(100000, 0, -1))

]

easy\_insert\_times, normal\_insert\_times, hard\_insert\_times = [], [], []

easy\_remove\_times, normal\_remove\_times, hard\_remove\_times = [], [], []

# измерение времени для вставки и удаления элементов (простой случай)

for arr in easy\_arrs:

avl\_tree = AVLTree()

# вставка

start\_time = time.time()

for value in arr:

avl\_tree.insert(value)

easy\_insert\_times.append(time.time() - start\_time)

# удаление

start\_time = time.time()

for value in arr:

avl\_tree.remove(value)

easy\_remove\_times.append(time.time() - start\_time)

# измерение времени для вставки и удаления элементов (средний случай)

for arr in normal\_arrs:

avl\_tree = AVLTree()

# вставка

start\_time = time.time()

for value in arr:

avl\_tree.insert(value)

normal\_insert\_times.append(time.time() - start\_time)

# удаление

start\_time = time.time()

for value in arr:

avl\_tree.remove(value)

normal\_remove\_times.append(time.time() - start\_time)

# измерение времени для вставки и удаления элементов (сложный случай)

for arr in hard\_arrs:

avl\_tree = AVLTree()

# вставка

start\_time = time.time()

for value in arr:

avl\_tree.insert(value)

hard\_insert\_times.append(time.time() - start\_time)

# удаление

start\_time = time.time()

for value in arr:

avl\_tree.remove(value)

hard\_remove\_times.append(time.time() - start\_time)

sizes = [10, 100, 1000, 10000, 100000]

data = {

'size': sizes \* 6,

'case': ['Легкий случай (вставка)'] \* 5 + ['Средний случай (вставка)'] \* 5 + ['Сложный случай (вставка)'] \* 5 +

['Легкий случай (удаление)'] \* 5 + ['Средний случай (удаление)'] \* 5 + ['Сложный случай (удаление)'] \* 5,

'time': easy\_insert\_times + normal\_insert\_times + hard\_insert\_times +

easy\_remove\_times + normal\_remove\_times + hard\_remove\_times

}

df = pd.DataFrame(data)

# построение графика для вставки

plt.figure(figsize=(10, 6))

insert\_df = df[df['case'].str.contains("вставка")]

for case\_label in insert\_df['case'].unique():

subset = insert\_df[insert\_df['case'] == case\_label]

plt.plot(subset['size'], subset['time'], marker='o', label=case\_label)

plt.xlabel('Размер данных')

plt.ylabel('Время выполнения (сек)')

plt.title('Время выполнения операции вставки в АВЛ-дереве')

plt.xscale('log')

plt.xticks(sizes, sizes)

plt.legend()

plt.tight\_layout()

plt.show()

# построение графика для удаления

plt.figure(figsize=(10, 6))

remove\_df = df[df['case'].str.contains("удаление")]

for case\_label in remove\_df['case'].unique():

subset = remove\_df[remove\_df['case'] == case\_label]

plt.plot(subset['size'], subset['time'], marker='o', label=case\_label)

plt.xlabel('Размер данных')

plt.ylabel('Время выполнения (сек)')

plt.title('Время выполнения операции удаления в АВЛ-дереве')

plt.xscale('log')

plt.xticks(sizes, sizes)

plt.legend()

plt.tight\_layout()

plt.show()

data\_generation\_avl()