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

КУРСОВАЯ РАБОТА

по дисциплине «Алгоритмы и структуры данных»

Тема: «Исследование абстрактных структур данных. Хеш-таблицы с открытой адресацией и методом цепочек.»

Студент гр. 0304 _____ Гурьянов С.О.

Преподаватель _____ Берленко Т.А.

Санкт-Петербург
2021

АННОТАЦИЯ

В данной курсовой работе выполняется сравнение вставки, поиска и удаления в AVL-дереве и Хэш-таблице с двойным хэшированием. Программа написана на языке Python. Содержится интерфейс, тестирование и обработка результатов.

ЗАДАНИЕ
НА КУРСОВУЮ РАБОТУ

Студент: Гурьянов С.О.
Группа: 0304

Тема работы: «Исследование абстрактных структур данных. АВЛ-дерево vs Хеш-таблица (двойное хеширование).»

Исходные данные:
АВЛ-дерево) vs Хеш-таблица (двойное хеширование). **Исследование.** "Исследование" — реализация требуемых структур данных/алгоритмов; генерация входных данных (вид входных данных определяется студентом); использование входных данных для измерения количественных характеристик структур данных, алгоритмов, действий; сравнение экспериментальных результатов с теоретическими. Вывод промежуточных данных не является строго обязательным, но должна быть возможность убедиться в корректности алгоритмов.
Содержание пояснительной записки:
«Содержание», «Введение», «Реализации структур данных», «Тестирование корректности реализаций структур данных», «Исследование структур данных», «Заключение», «Список использованных источников».
Предполагаемый объем пояснительной записки:
Не менее 13 страниц.

Дата выдачи задания: 26.10.21

Дата сдачи реферата:
Дата защиты реферата:

Студент _____ Гурьянов С.О.

Преподаватель _____ Берленко Т.А.

СОДЕРЖАНИЕ

	Содержание	3
1.	Введение	4
1.1.	Цель курсовой работы и исходные условия	4
1.2.	Задачи	4
2.	Реализации структур данных	4
2.1.	Хеш-таблица с разрешением коллизий методом двойного хеширования	4
2.2.	AVL-дерево	7
3.	Тестирование корректности реализаций структур данных	9
4.	Исследование структур данных	10
4.1.	Хеш-таблица с разрешением коллизий методом двойного хеширования	13
4.2.	AVL-дерево	18
4.3.	Сравнение структур данных	21
5.	Вывод	22

1. ВВЕДЕНИЕ

1.1. Цель курсовой работы и исходные условия

Исходные условия таковы: Хеш-таблица (двойное хеширование) vs AVL-дерево.

Исследование.

"*Исследование*" - реализация требуемых структур данных/алгоритмов; генерация входных данных (вид входных данных определяется студентом); использование входных данных для измерения количественных характеристик структур данных, алгоритмов, действий; сравнение экспериментальных результатов с теоретическими. Вывод промежуточных данных не является строго обязательным, но должна быть возможность убедиться в корректности алгоритмов.

Целью данной работы является реализация данных структур данных, тестирование их характеристик производительности по времени, а также сравнение операций вставки и поиска

1.2. Задачи

Для достижения поставленной цели требуется решить следующие задачи:

- Реализовать интерфейс и саму логику требуемых структур данных (хеш-таблиц с разрешением коллизий на основе двойного хеширования и АВЛ-дерева).

- Реализовать набор тестов для проверки корректности работы реализации структур данных.
- Реализовать программу для исследования характеристик производительности данных структур.
- Построить графики зависимости времени выполнения операций от количества входных данных.

2. РЕАЛИЗАЦИИ СТРУКТУР ДАННЫХ

Хеш-табли́ца — это структура данных, реализующая интерфейс ассоциативного массива, а именно, она позволяет хранить пары и выполнять три операции: операцию добавления новой пары, операцию поиска и операцию удаления пары по ключу.

АВЛ-дерево — сбалансированное по высоте двоичное дерево поиска: для каждой его вершины высота её двух поддеревьев различается не более чем на 1. Каждая структура данных реализована в виде отдельного класса.

2.1. ХЕШ-ТАБЛИЦА С РАЗРЕШЕНИЕМ КОЛЛИЗИЙ МЕТОДОМ ДВОЙНОГО ХЭШИРОВАНИЯ

Хеш-таблица реализована следующим образом:

```
class Hash_table:
    def __init__(self, size, hash1, hash2, max_fill=0.3):
        self.size = 2 ** size
        self.pairs = [[None, None] for i in range(self.size)]
        self.hash1 = hash1
        self.hash2 = hash2
        self.max_fill = max_fill
        self.current_fill = 0

    def extension(self):
        self.current_fill /= 2
        self.size *= 2
        old_pairs = self.pairs
        self.pairs = [[None, None] for i in range(self.size)]
        for pair in old_pairs:
            if pair[0] != None:
                self.add(pair)
```

```
def find_hash(self, object):
    i = -1
    while 1:
        i += 1
        yield (self.hash1(object) + i * self.hash2(object)) % self.size

def add(self, pair):
    get_hash = self.find_hash(pair[0])
    hash = next(get_hash)
    while self.pairs[hash][0] != None and self.pairs[hash][0] != pair[0]:
        hash = next(get_hash)
    self.pairs[hash] = pair
    self.current_fill += 1 / self.size
    if self.current_fill >= self.max_fill:
        self.extension()

def delete(self, key):
    get_hash = self.find_hash(key)
    hash = next(get_hash)
    while self.pairs[hash][0] != key:
        hash = next(get_hash)
    self.pairs[hash] = [None, None]

def find(self, key):
    get_hash = self.find_hash(key)
    hash = next(get_hash)
    while self.pairs[hash][0] != key:
        if self.pairs[hash][0] == None:
            raise IndexError(key)
        hash = next(get_hash)
    return self.pairs[hash][1]
```

Конструктор данного класса принимает на вход размер таблицы (точнее, степень двойки, соответствующую размеру таблицы), 2 хэш функции, которые будут использоваться, а так же максимальный коэффициент заполнения. Создаётся список пар [None, None] длины, соответствующей длине хэш-таблицы. Текущий уровень заполнения таблицы выставляется на ноль.

Метод *add(self, pair)* позволяет добавить пару *pair* в хэш-таблицу. Алгоритм добавления следующий: сперва вычисляется значение первой хэш-функции от ключа pair[0]. Далее происходит обращение к хэш-таблицы по индексу, которую вернула хэш-функция. Если при этом на данном месте пара [None, None], то она заменится парой pair. Изначально счётчик i = 0. В случае, когда возникла коллизия, алгоритм её обработки следующий: счётчик i инкрементируется, считается значение (hash1(pair[0]) + i * hash2(pair[0])) % self.size, где self.size –размер таблицы, происходит попытка обращения к данному индексу хэш-таблицы. Процедура повторяется до тех пор, пока коллизия не возникнет. Для того, чтобы была гарантия того, что в случае, когда имеется свободная ячейка в хэш-таблицы, она будет достигнута, достаточно, чтобы hash2(object) возвращала значения, взаимно простые с размером таблицы. Это достигается за счёт того, что размер таблицы всегда является степенью 2, а вторая хэш-функция возвращает нечётные значения.

Аналогичным образом работает метод *find(self, pair)*. Только в случае, если будет выполнено обращение к элементу [None, None], метод вернёт исключение о том, что ключ отсутствует в таблицы. В случае, когда была найдена пара, отличная от [None, None], будет выполнена проверка ключей. Если ключи будут совпадать, будет возвращено значение, соответствующее данном ключу, в противном случае счётчик инкрементируется и будет пересчитано значение (hash1(pair[0]) + i * hash2(pair[0])) % self.size, как в случае метода *add(self, pair)*.

Однако после каждой вставки в хэш-таблицу пересчитывает её коэффициент заполнения. В том случае, когда он достигнет максимально возможного (определённого при инициализации хэш-таблицы), будет выполнена перестройка хэш-таблицы *def extension(self)*. В этом случае её размер увеличивается вдвое, а также происходит вставка всех пар заново. Требуется это для того, чтобы сохранять скорость работы хэш-таблицы при поиске, вставке и удалении.

Метод *def delete(self, key)* позволяет удалить элемент хэш-таблицы. Алгоритм тут аналогичен алгоритму поиска, но при удалении элемента уменьшается коэффициент заполнения таблицы.

2.2. AVL-дерево

Узел AVL-дерево реализован следующим образом:

```
class node:
    def __init__(self, value=None):
        self.value = value
        self.left_child = None
        self.right_child = None
        self.parent = None  # pointer to parent node in tree
        self.height = 1  # height of node in tree (max dist. to leaf) NEW FOR AVL
```

Конструктор данного класса принимает значение value, которое выставляет в узел. При этом левый, правый наследники и родитель выставляются в None, а высота – в 1.

Само AVL-дерево представлено в виде отдельного класса. Его рекурсивный метод *_find(self, value, cur_node)* выполняет рекурсивный поиск элемента. Он сравнивает значение value со значением узла *cur_node*. Если они оказались равными, будет возвращен узел *cur_node*, иначе будет запущен рекурсивный поиск по его наследникам.

Метод *_left_rotate(self, z)* позволяет выполнить малое левое вращение, которое показано на рисунке 1.

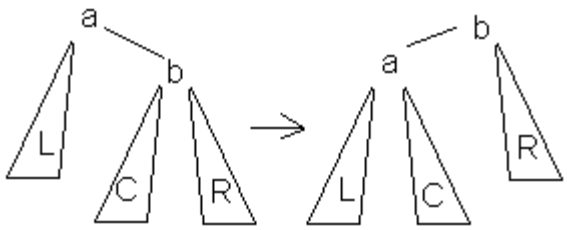

Рисунок 1 – малое левое вращение

Метод *_right_rotate(self, z)* позволяет выполнить малое правое вращение, которое показано на рисунке 2.

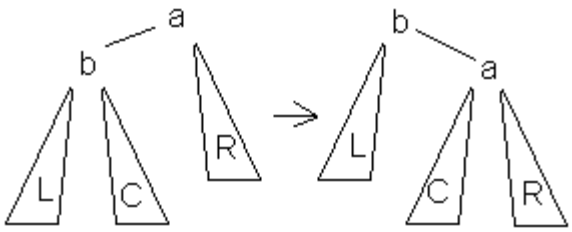

Рисунок 2 – малое правое вращение

Метод *_rebalance_node(self, z, y, x)* выполняет ребаланс дерева, используя левое и правое вращения.

Методы *_inspect_insertion(self, cur_node, path)* и *_inspect_deletion(self, cur_node)* используются для балансировки дерева при вставки или удалении элемента соответственно.

Само удаление вершины происходит по следующему алгоритму: если вершина — лист, то удалим её и вызовем балансировку всех её предков в порядке от родителя к корню. Иначе найдём самую близкую по значению вершину в поддереве наибольшей высоты (правом или левом) и переместим её на место удаляемой вершины, при этом вызвав процедуру её удаления.

Вставка осуществляется по следующему алгоритму: если ключ меньше того значения узла, который сейчас рассматривается, то происходит проверка наличия левого узла. Если его нет, то вставляем элемент в качестве левого узла, иначе вызываем вставку рекурсивно для левого узла. Если ключ больше того значения узла, который сейчас рассматривается, то происходит проверка наличия правого узла. Если его нет, то вставляем элемент в качестве правого узла, иначе вызываем вставку рекурсивно для правого узла.

3 ТЕСТИРОВАНИЕ КОРРЕКТНОСТИ РЕАЛИЗАЦИЙ СТРУКТУР ДАННЫХ

Тестирование корректности реализация структур данных содержится в файле *test.py*.

Созданы 2 функции для тестирования хэш-таблицы. Первая из них проверяет корректность работы хэш-таблиц различного размера, а вторая – различной максимальной заполненности.

```
def test_hash_for_size():
    length_pows = [i for i in range(2, 7)]
    for length_pow in length_pows:
        ht = Hash_table(length_pow, hash1, hash2)
        right_values = []
        for i in range(2 ** length_pow):
            right_values.append(2 ** i - 3 * i + 5)
            ht.add([i, 2 ** i - 3 * i + 5])
        for i in range(2 ** length_pow):
            value = ht.find(i)
            assert value == right_values[i]
            ht.delete(i)
            assert ht.find(i) == None


def test_hash_for_max_fill():
    fills = [i for i in range(2, 10)]
    for fill in fills:
        ht = Hash_table(10, hash1, hash2, fill / 10)
        right_values = []
        for i in range(2 ** 10):
            right_values.append(2 ** i - 3 * i + 5)
            ht.add([i, 2 ** i - 3 * i + 5])
        for i in range(2 ** 10):
            value = ht.find(i)
            assert value == right_values[i]
```

```
        ht.delete(i)
        assert ht.find(i) == None
```

AVL-таблица тестируется тремя функциями:

```
def test_for_AVL1():
    tree = AVLTree()
    for i in range(10):
        tree.insert(i)
    right_Traverse = [3, 1, 0, 2, 7, 5, 4, 6, 8, 9]
    preverse = []
    preOrdertestTraverse(tree, tree.root, preverse)
    assert preverse == right_Travers

def test_for_AVL2():
    tree = AVLTree()
    for i in range(20):
        tree.insert(i)
    tree.delete_value(19)
    tree.delete_value(7)
    right_Traverse = [8, 3, 1, 0, 2, 5, 4, 6, 15, 11, 9, 10, 13, 12, 14, 17, 16, 18]
    preverse = []
    preOrdertestTraverse(tree, tree.root, preverse)
    assert preverse == right_Traverse

def test_for_AVL3():
    tree = AVLTree()
    for i in range(0):
        tree.insert(i)
    right_Traverse = []
    preverse = []
    preOrdertestTraverse(tree, tree.root, preverse)
    assert preverse == right_Traverse
```

4. ИССЛЕДОВАНИЕ СТРУКТУР ДАННЫХ

Для измерения времени, затраченного на выполнение операций вставки, поиска и удаления использовался метод time.perf_counter() библиотеки time. Для

исследования хэш-таблиц написан модуль research_Hash.py. В нём объявлены функции best_hash_functions(), average_hash_functions(), worse_hash_functions(). Каждая из них возвращает 2 функции hash1 и hash2.

В функции лучшего случая hash1(object) будет всегда возвращать 0, а функция hash2(object) вернёт либо наибольшее нечётное число, не превышающее заданное, если в качестве object передаётся целое число, либо ближайшее нечётное число к длине строкового представления объекта. Основной смысл данной хэш-функции в том, чтобы в качестве объектов передавать целые числа. Лучший случай хэш-функции будет реализован тогда, когда передаются либо только чётные, либо только нечётные числа, тогда хэш-функция будет возвращать уникальные значения для каждого объекта. Коллизий при этом не возникает.

В функции среднего случая hash1(object) и hash2(object) возвращают число, которое зависит от строкового представления объекта. Для некоторых значений возвращаемые хэш-функциями значения будут совпадать, что приведёт к возникновению коллизий.

В функции худшего случая hash1(object) будет всегда возвращать 1, а hash2(object) – всегда 3. Таким образом, для 2 любых объектов хэши будут совпадать.

В исследовании хэш-таблицы происходят замеры времени вставки, поиска и удаления элемента для разных длин хэш-таблиц (диапазон от 2**10 до 2**15), разных коэффициентов заполнения (50%, 75% и 99%) и разных случаев (лучший, средний и худший).

Исследование AVL-дерева происходит в модуле research_AVL. Изначально формируется список из 2**15 случайных чисел в диапазоне от 0 до 2**20. Далее формируются 2 списка: lengths и times. В lengths помещается число узлов AVL-дерева, а в times – время, которое при этом понадобилось для совершения операции при данном числе узлов. Всего происходит 3 исследования: вставка, поиск и удаление.

4.1. ХЕШ-ТАБЛИЦА С РАЗРЕШЕНИЕМ КОЛЛИЗИЙ МЕТОДОМ ДВОЙНОГО ХЭШИРОВАНИЯ

Условные обозначения диаграмм:

K = 0.5

K = 0.75

K = 0.99

Лучший случай:

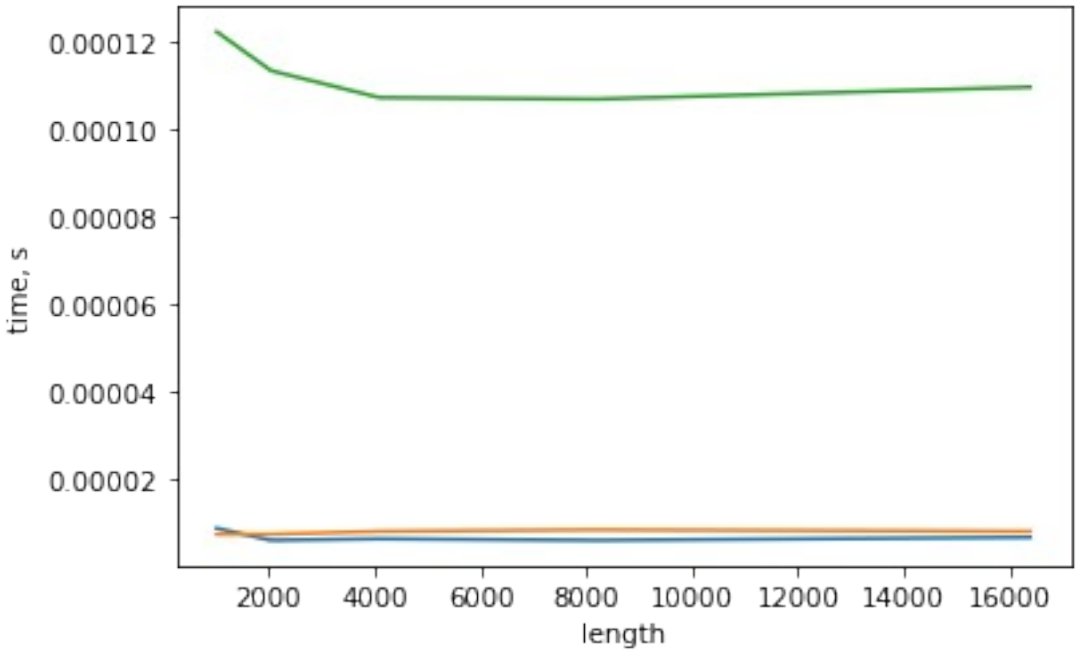

Рисунок 1 - вставка, лучший случай

Как видно из графика, в этом случае время практически константное при любой длине хэш-таблицы. Скорость выполнения операции в лучшем случае составляет O(1) и совпадает с теоретическим.

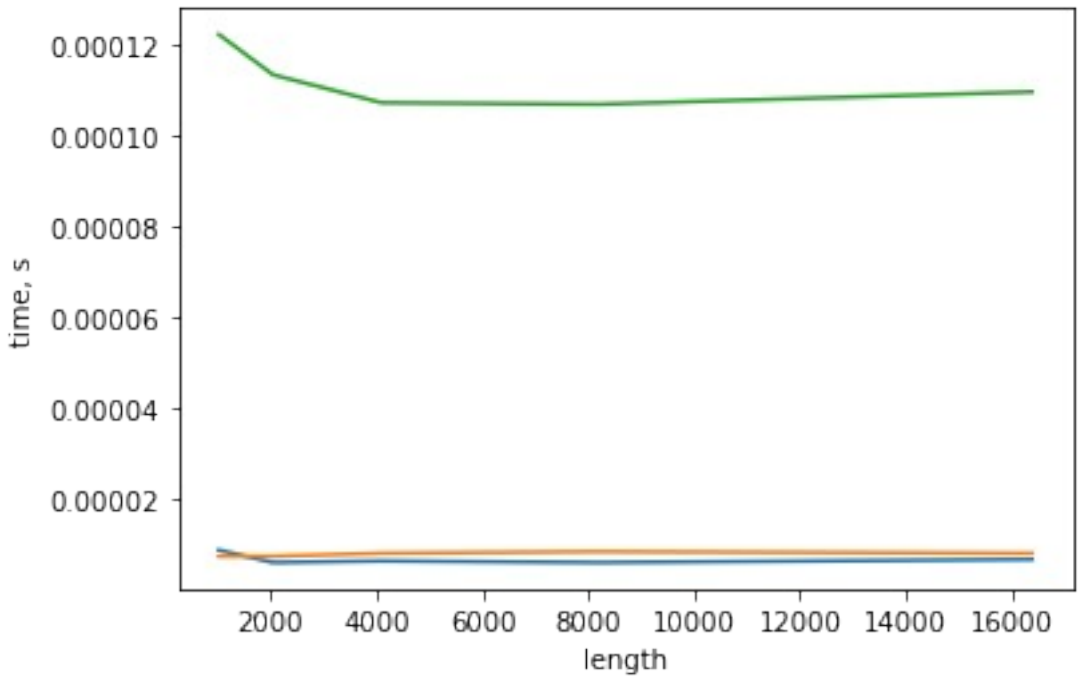

Рисунок 2 - поиск, лучший случай

В лучшем случае поиск выполняется за время O(1), о чём свидетельствует данный график. Скорость поиска в лучшем случае совпадает с теоретической.

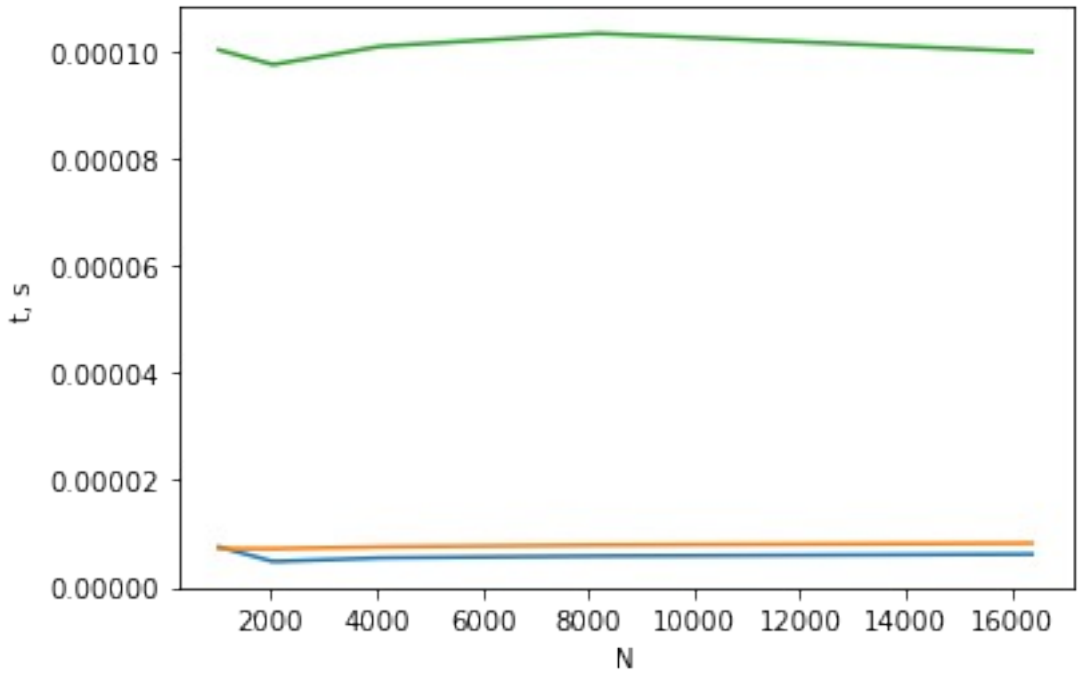

Рисунок 3 - удаление, лучший случай

Удаление выполняется с такой же скоростью O(1).

Средний случай:

Зелёный цвет — K = 0.99, оранжевый — K = 0.75, синий — K = 0.50.

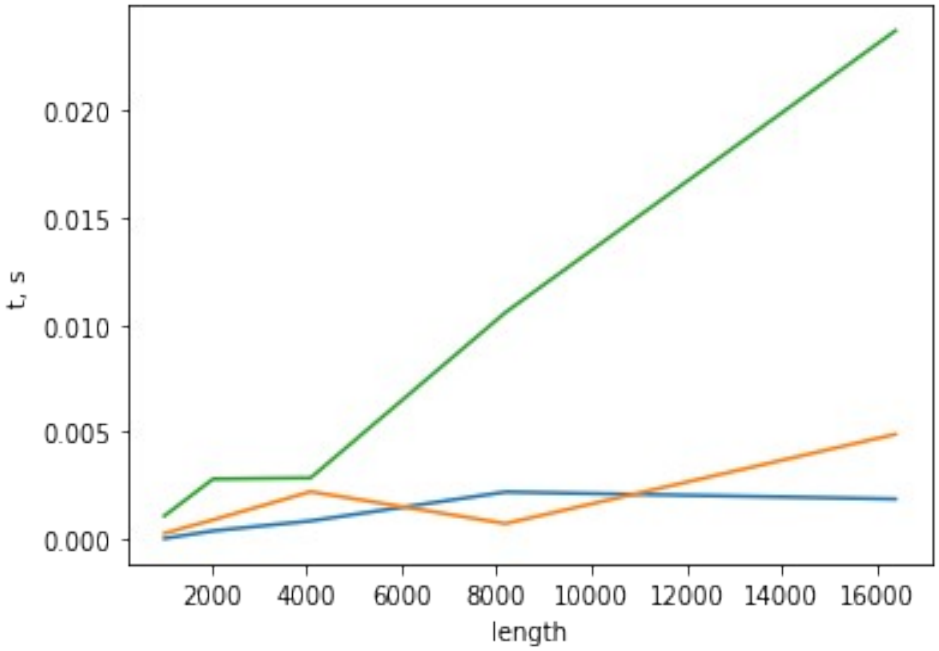

Рисунок 4 - вставка, средний случай

Как видно, в среднем случае время вставки зависит от коэффициента заполнения таблицы и определяется как O(1 + KN).

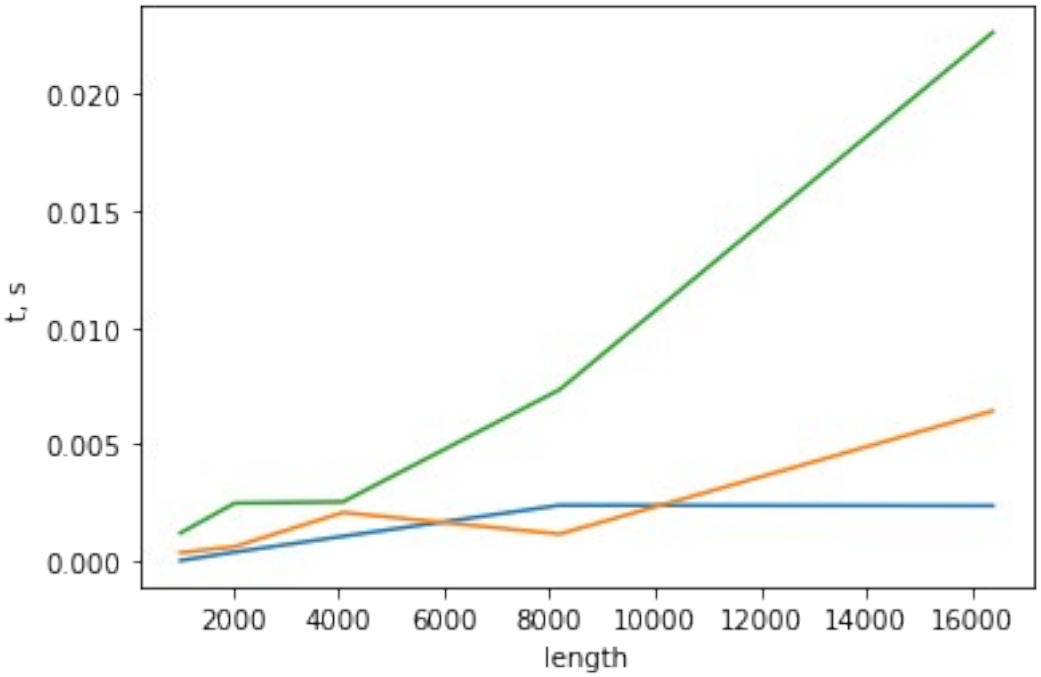

Рисунок 5 - поиск, средний случай

Аналогичная ситуация и с поиском. Зелёная линия идёт почти как прямая, что свидетельствует о том, что время поиска зависит от коэффициента заполнения таблицы. То есть, оно составляет O(1 + KN).

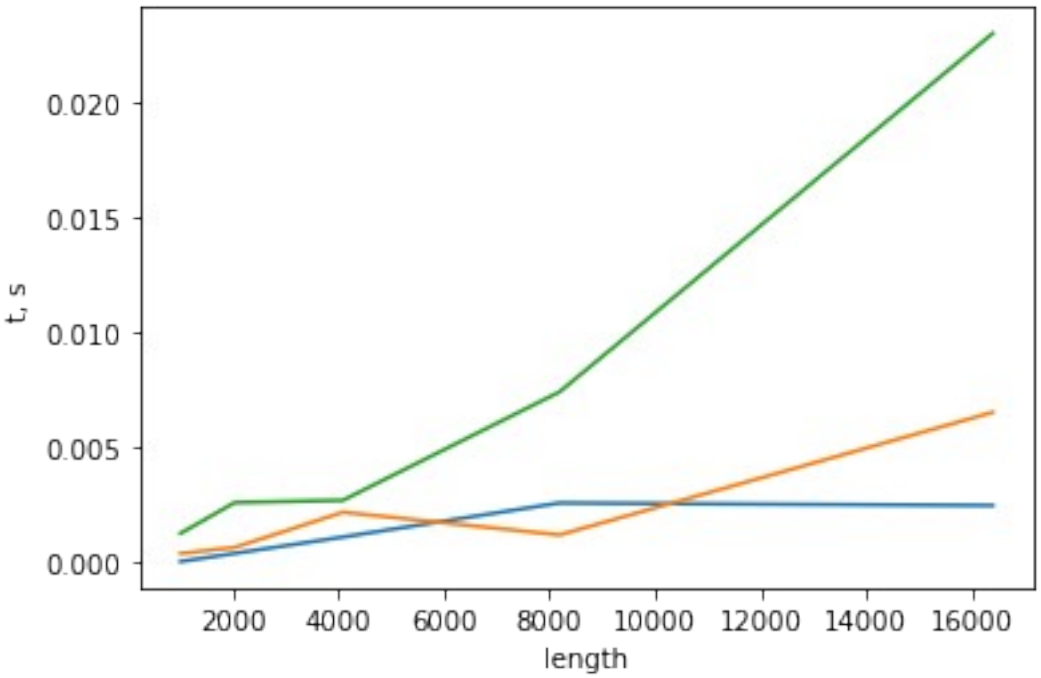

Рисунок 6 - удаление, средний случай

Как при вставке и поиске, при удалении скорость работы составляет O(1 + KN).

Худший случай:

Зелёный цвет — K = 0.99, оранжевый — K = 0.75, синий — K = 0.50.

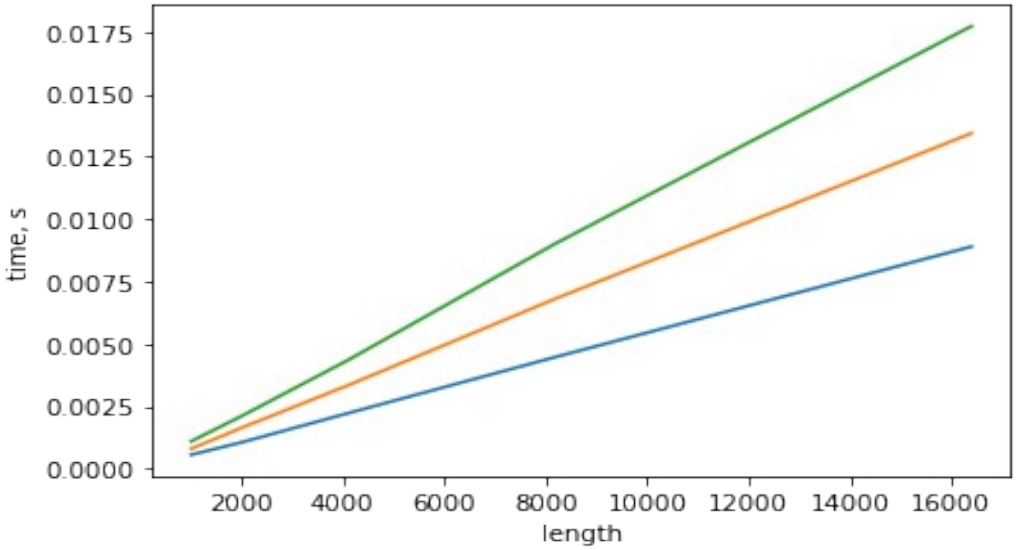

Рисунок 7 - вставка, худший случай

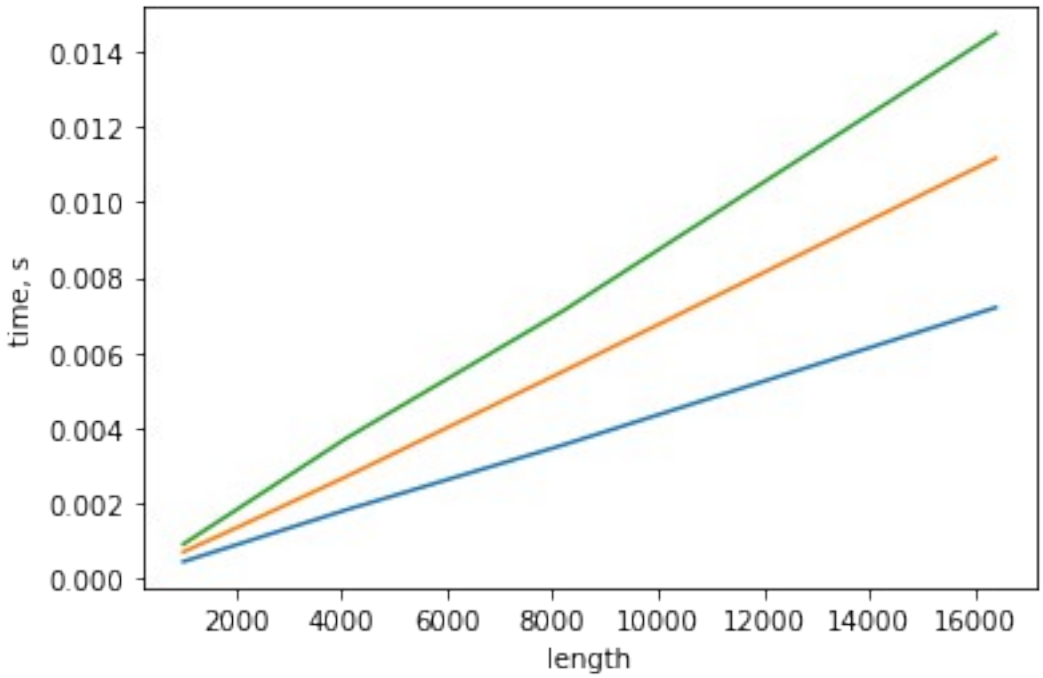

Рисунок 8 - поиск, худший случай

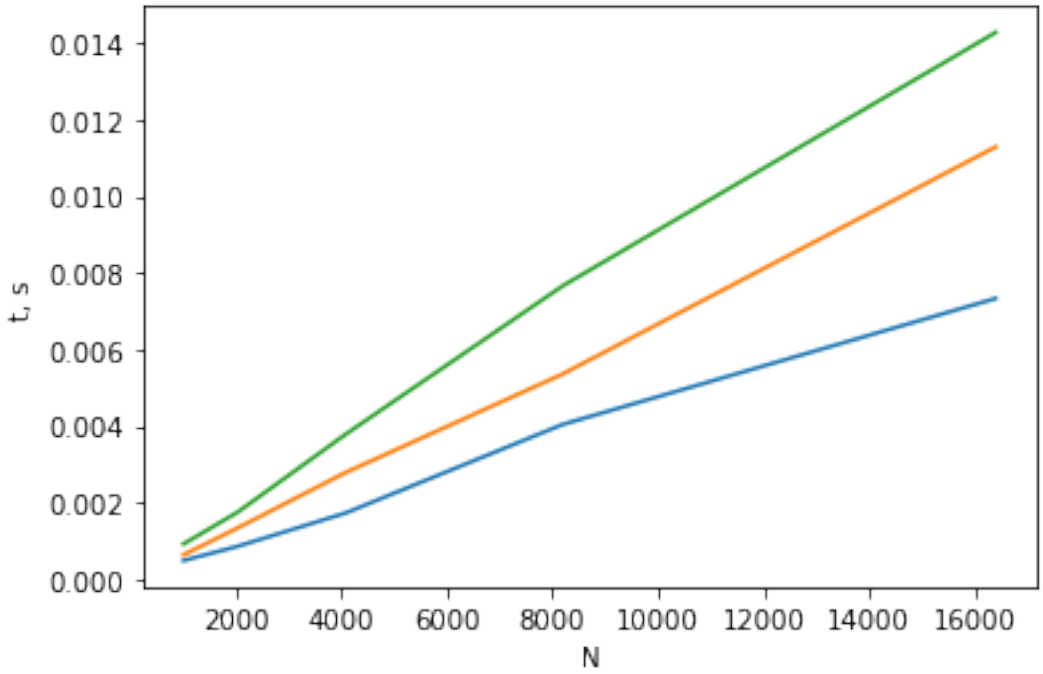

Рисунок 9 - удаление, худший случай

Понятно, что в худшем случае время работы вставки, поиска и удаления составляет O(n). Объясняется это тем, что при каждой операции возникает коллизия, что из-за чего приходится обходить почти всю хэш-таблицу.

Таким образом, в лучшем случае каждая из операций работает за время O(1), в среднем случае – за O(1 + KN), в худшем – за время O(N). Объясняется это выбором хэш-функции. Если хэш-функция удачно подобрана, то коллизий возникать не будет вообще, из-за чего любая операция будет выполняться почти мгновенно, при среднем выборе хэш-функции могут возникать коллизии, тогда время работы уже будет зависеть от коэффициента заполнения хэш-таблицы, ведь чем больше заполнена таблица, тем больше шанс попасть на коллизию при обращении к хэш-таблице. В худшем случае при выполнении операции придётся обойти всю хэш-таблицу.

4.2. AVL-дерево

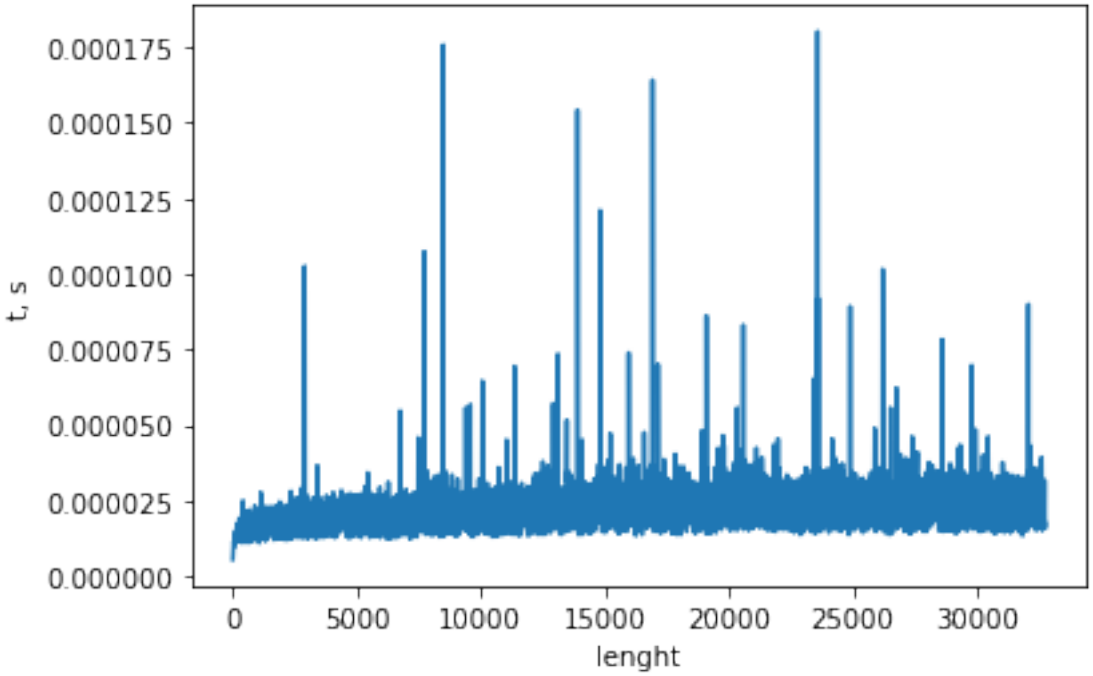

Рисунок 10 - вставка в AVL-дерево

Для того чтобы оценить сложность алгоритма вставки, нужно дорисовать верхнюю и нижнюю воображаемые границы графика.

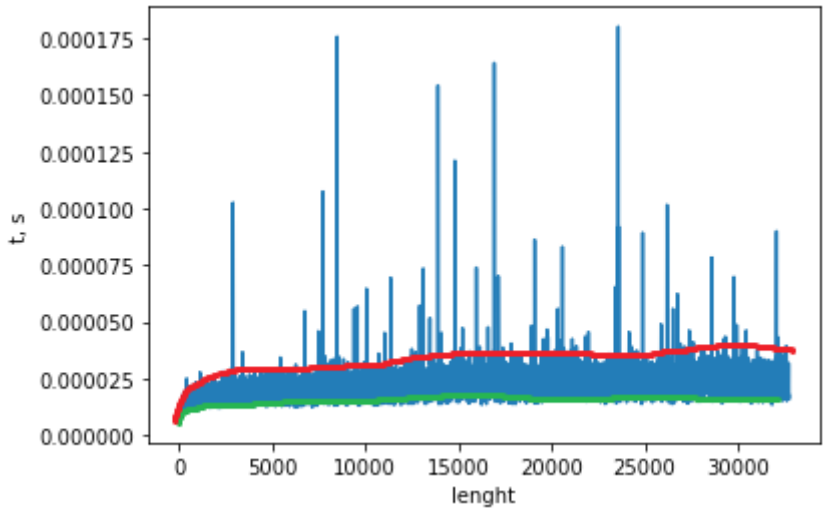

Рисунок 11 - вставка в AVL-дерево с дорисованными границами

Теперь можно заметить, что в худшем случае (соответствует красная линия) вставка в дерево занимает время O(log N) точно так же, как и в лучшем (зелёная

линия). Поэтому всё, что находится между красной и зелёной линией, тоже подчиняется закону O(log N). Поэтому вставка в AVL-дерево выполняется во всех случаях за время O(log N). Отклонения объясняются случайной генерацией дерева, поскольку вставка иногда осуществляется в начало дерево.Это полностью соответствует теоретическим положениям.

Повторим ситуацию, но уже с поиском.

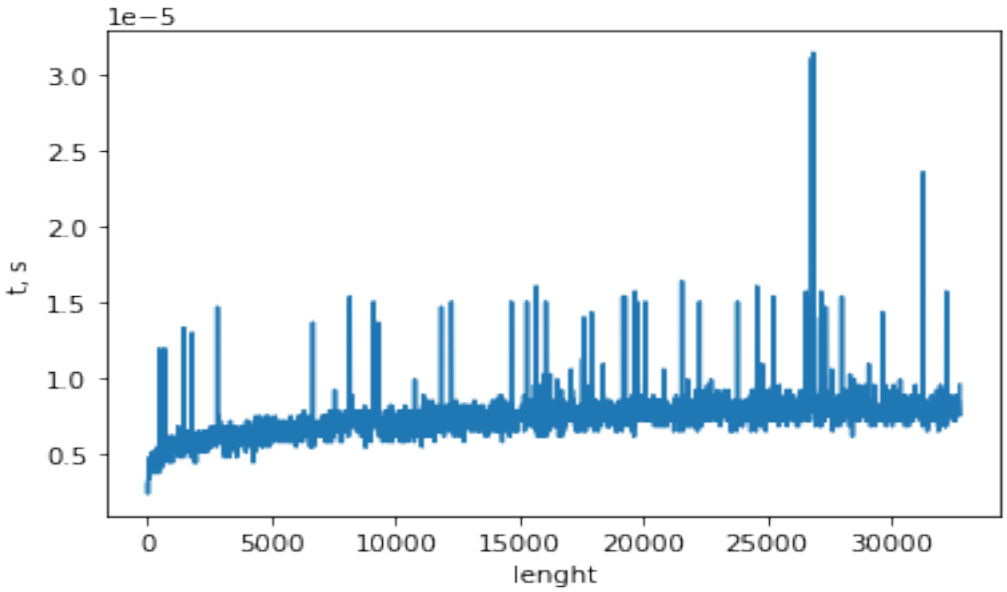

Рисунок 12 - поиск в AVL-дереве

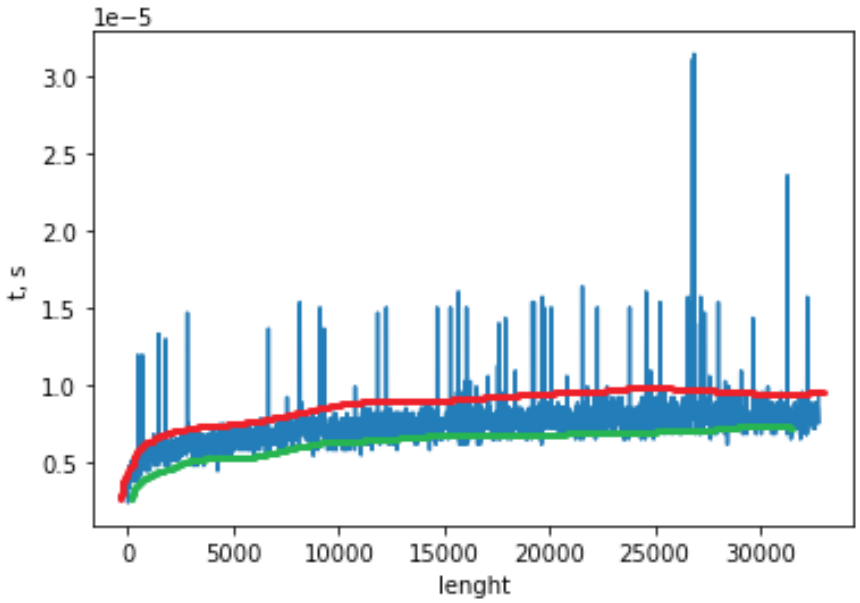

Рисунок 13 - поиск в AVL-дереве (изображены границы)

Как видно, поиск в AVL-дереве точно так же во всех случаях работает за время O(log N). Отклонения же происходят тогда, когда мы осуществляем поиск нижних элементов. В нашем случае это происходит случайным образом, так как исходное дерево генерируется тоже случайно.

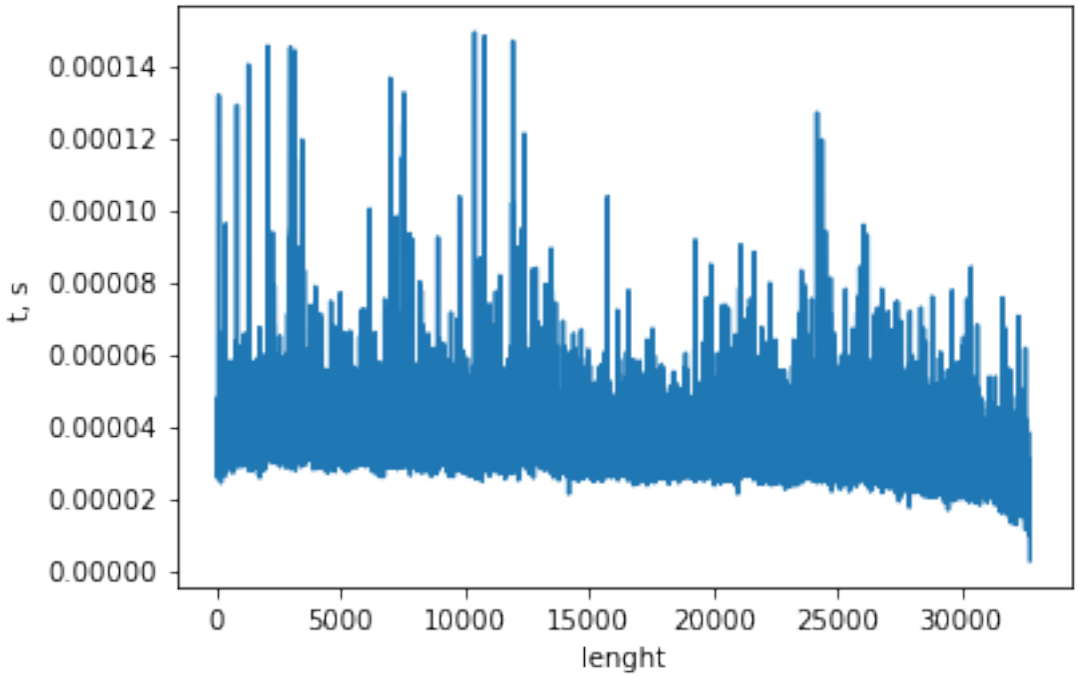

Рисунок 14 - удаление из AVL-дерева

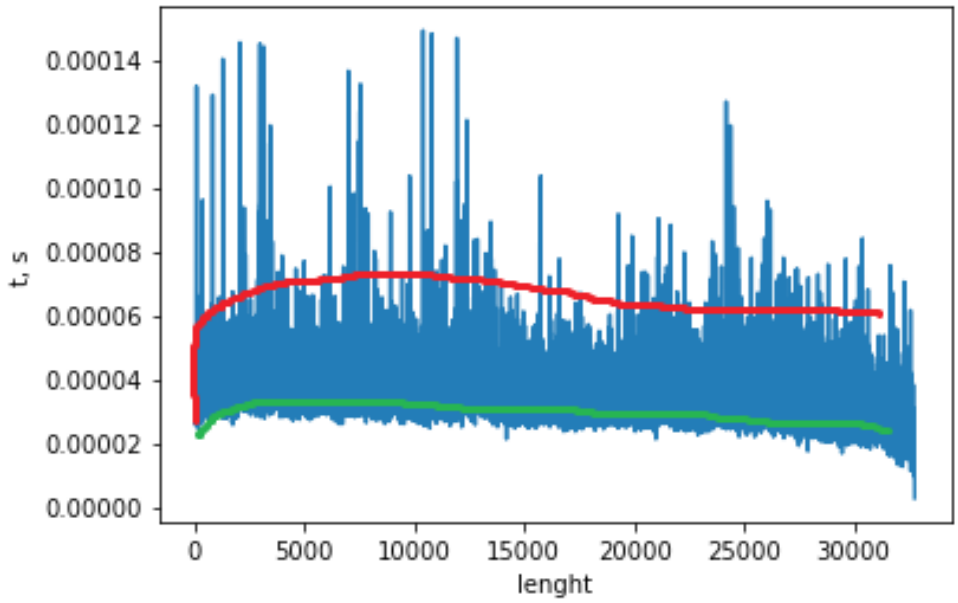

Рисунок 15 - удаление из AVL-дерева (границы дорисованы)

С удалением ситуация та же: во всех случаях выполняется за время O(log N). Опять же, есть соответствие теории. Однако отклонения от границ объясняются тем, что каждый раз происходит удаление рандомного элемента, и иногда этот элемент оказывается корнем дерева.

Таким образом, все три операции в AVL-дереве всегда выполняются за время O(log N). Это объясняется его структурой: высота дерева составляет порядка log N, поэтому при выполнении какой-либо операции требуется порядка log N сравнений.

4.3. СРАВНЕНИЕ СТРУКТУР ДАННЫХ

Если же теперь говорить о сравнении двух этих структур, то AVL-дерево стабильно во всех случаях работает за время O(log N). Это его преимущество по сравнению с хэш-таблицей. В хэш-таблице требуется наличие правильной хэш-функции. В противном случае, её скорость работы будет очень мала. Однако при правильном подборе хэш-функции хэш-таблица будет работать почти мгновенно, опережая по скорости AVL-дерево.

5 ЗАКЛЮЧЕНИЕ

Были изучены, смоделированы такие структуры данных, как AVL-дерево и хэш-таблица с методом двойного хэширования. Было проведено подробное сравнение операций вставки, поиска и удаления элементов из этих структур. Было экспериментально доказано, что время выполнения операций действительно соответствует теории. Выяснено, что в среднем AVL-дерево работает быстрее и не зависит от каких-либо входных данных, в то время как хэш-таблица очень сильно зависит от хэш-функции. Однако при правильном подборе скорость её работы выше скорости работы AVL-дерева.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 https://ru.m.wikipedia.org/wiki/%D0%A5%D0%B5%D1%88-%D1%82%D0%B0%D0%B1%D0%BB%D0%B8%D1%86%D0%B0

2 https://youtu.be/4qJVFQ-LK7A

3 https://pythonworld.ru/samouchitel-python

4 https://matplotlib.org