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 структур данных: АВЛ-дерево и хеш-таблица (открытая адресация)

Студент гр. 1384 _____ Прошичев А.В.

Преподаватель _____ Иванов Д.В.

Санкт-Петербург

2022

ЗАДАНИЕ
НА КУРСОВУЮ РАБОТУ

Студент Прошичев А.В.

Группа 1384

Тема работы: Исследование структур данных: АВЛ-дерева и хеш-таблица (открытая адресация).

Исходные данные:

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», «Основные теоретические сведения», «Оценка эффективности структур данных по времени», «Заключение», «Список используемой литературы», «Приложение А»

Предполагаемый объем пояснительной записки:
Не менее 25 страниц.

Дата выдачи задания: 25.10.2022

Дата сдачи реферата: 26.12.2022

Дата защиты реферата: 27.12.2022

Студент _____ Прошичев А.В.

Преподаватель _____ Иванов Д.В.

АННОТАЦИЯ

Курсовая работа требует реализацию таких структур данных, как АВЛ-дерево и хеш-таблица с открытой адресацией. Для обоих структур создаются операции вставки, удаления и поиска. Затем генерируются входные данные и происходит сравнение теоретических и практических оценок эти структур между собой и по отдельности. На основе результатов составляются графики, по которым видна зависимость времени работы программы от количества элементов в структуре.

SUMMARY

Coursework requires the implementation of data structures such as an AVL tree and a hash table with open addressing. Insert, delete, and search operations are created for both structures. Then the input data is generated and the theoretical and practical estimates of these structures are compared with each other and separately. Based on the results, graphs are compiled, according to which the dependence of the program's operation time on the number of elements in the structure is visible.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Целью данной работы является получение навыков построение таких структур данных, как Хеш-таблица с открытой адресацией и АВЛ-дерево, а именно реализация операция вставки, поиска и удаления, также генерация исходных данных и исследование на этих данных времени различных операций данных структур и сравнение полученных результатов с теоретической оценкой.

1. ОСНОВНЫЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

1.1. Хеш-таблица с открытой адресацией

Хеш-таблица - структура данных, которая позволяет хранить пары (ключ, значение) и осуществлять доступ к элементу по ключу. (см. рис. 1)

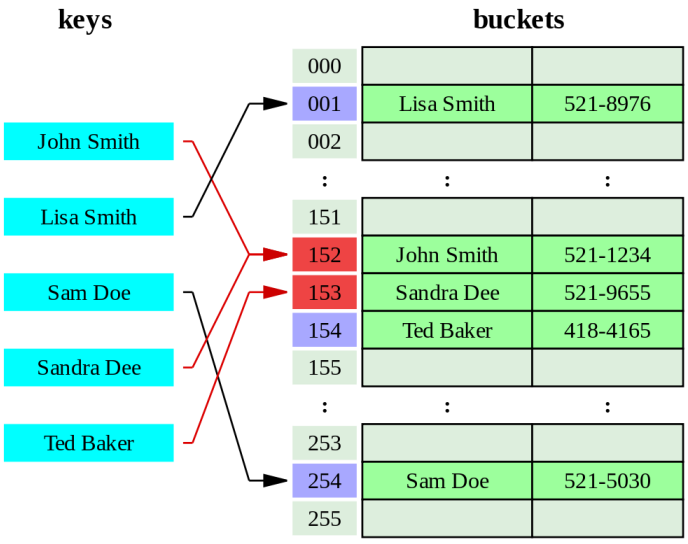

Рисунок 1 – Хеш-таблица с открытой адресацией

Такая структура данных имеет минимум 3 вида операций: вставка, поиск и удаление. Каждая хеш-таблица имеет собственную хеш-функцию. Данная хеш-функция сопоставляет ключу определённое числовое значение и по этому значению вписывает заданную пару из ключа и значения с хеш-таблицу. Проблема хеш-таблицы заключается в коллизиях. Ситуация, когда для различных ключей получается одно и то же хеш-значение, называется коллизией. Существуют 2 способа решить данную проблему: метод цепочек и открытая адресация.

В рамках данной курсовой работы рассматривается открытая адресация. В массиве хранятся сами пары ключ-значение. Алгоритм вставки элемента проверяет ячейки массива в некотором порядке до тех пор, пока не будет найдена первая свободная ячейка, в которую и будет записан новый элемент. Последовательность, в которой просматриваются ячейки хеш-таблицы, называется последовательностью проб. В общем случае, она зависит только от

ключа элемента. Операция удаления в хеш-таблице с данным способом решения коллизий должна оставлять информацию о удалённой ячейки, чтобы операция вставки смогла сохранить в эту ячейку элемент, а операция поиска смогла пройти через эту ячейку дальше к другим элементам, которые были добавлены после удаляемого элемента. Существуют 3 способа выбрать последовательность проб:

- Линейное пробирование – пробирование, при котором ячейки просматриваются через определённый шаг. Формула имеет вид:

$$(hash(x) + i*k) \bmod N$$

где x – ключ, hash() – хеш-функция, i – номер шага пробирования, k – константа, N – максимальное количество элементов в таблице.

- Квадратичное пробирование – пробирование, при котором ячейки просматриваются через определённый шаг с квадратичной зависимость. Формула имеет вид:

$$(hash(x) + c_1*i + c_2*i^2) \bmod N$$

где x – ключ, hash() – хеш-функция, i – номер шага пробирования, c1 и c2 – константы, N – максимальное количество элементов в таблице.

- Двойное хеширование – пробирование, при котором ячейки просматриваются через определённый шаг с зависимость от определённой новой хеш-фукнции. Формула имеет вид:

$$(hash(x) + i*new\_hash(x)) \bmod N$$

где x – ключ, hash() и new_hash() – хеш-функции, i – номер шага пробирования, N – максимальное количество элементов в таблице.

1.2. АВЛ-дерево

АВЛ-дерево — сбалансированное по высоте двоичное дерево поиска: для каждой его вершины высота её двух поддеревьев различается не более чем на 1. Данная структура хранит данные в виде пары из ключа и значения имеет минимум 4 операции: вставка по ключу, удаление по ключу, поиск по ключу, балансировка дерева.

Балансировка дерева является самой важной операцией, позволяющей дереву быть стабильно эффективным. Существуют 4 способа сбалансировать дерева:

- малое левое вращение (см. рис. 2)

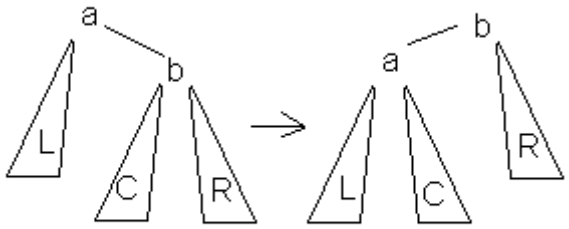

Рисунок 2 – Малое левое вращение

Осуществляется, если разница высоты поддерева *A* и поддерева *B* равна 2, а высота поддерева *C* не больше высоты поддерева *R.*

- малое правое вращение (см. рис. 3)

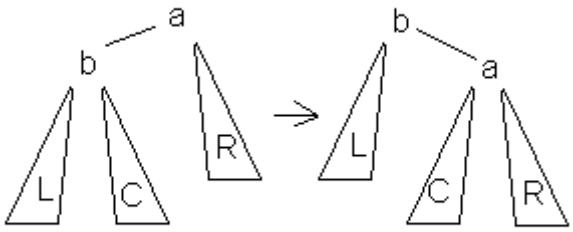

Рисунок 3 – Малое правое вращение

Осуществляется, если разница высоты поддерева *A* и поддерева *B* равна 2, а высота поддерева *C* не больше высоты поддерева *L.*

- большое левое вращение (см. рис. 4)

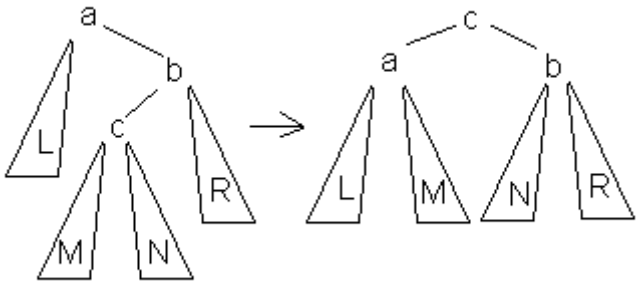

Рисунок 4 – Большое левое вращение

Осуществляется, если разница высоты поддерева *A* и поддерева *B* равна 2, а высота поддерева *C* больше высоты поддерева *R*.

- большое правое вращение (см. рис. 5)

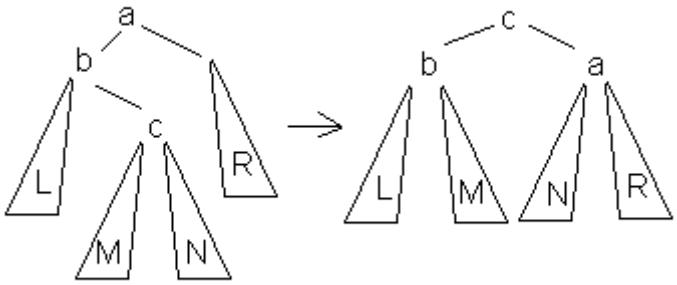

Рисунок 5 – Большое правое вращение

Осуществляется, если разница высоты поддерева *A* и поддерева *B* равна 2, а высота поддерева *C* больше высоты поддерева *L*.

Вставка в дерево происходит, как и в обычном двоичном дереве поиска, если вставляется ключ с меньшим значением, чем значение ключа текущей вершины, то осуществляет переход в левое поддерево, иначе – в правое. Подвешивается вставляемое значение, как лист. Затем происходит балансировка дерева.

Поиск элемента полностью совпадает с поиском в двоичном дереве. Алгоритм аналогичен алгоритму поиска места для элемента во время вставки.

Удаление элемента является самой сложной операцией в АВЛ-дереве. Для начала происходит поиск удаляемой вершины. Если такая найдена, то в

правом поддереве происходит поиск вершины с минимальным ключом, после чего вершина удаляется, на её место устанавливается вершина с минимальным ключом в правом поддереве. Затем происходит балансировка дерева.

2. ОЦЕНКА ЭФФЕКТИВНОСТИ СТРУКТУР ДАННЫХ ПО ВРЕМЕНИ

2.1. Оценка хеш-таблицы с открытой адресацией

Оценка сложности операций вставки, поиска и удаления в хеш-таблице приведена в таблице 1.

	Лучший случай	Среднее случай	Худший случай
Вставка	O(1)	O(1)	O(n)
Поиск	O(1)	O(1)	O(n)
Удаление	O(1)	O(1)	O(n)

Таблица 1 – оценка сложности операций в хеш-таблице

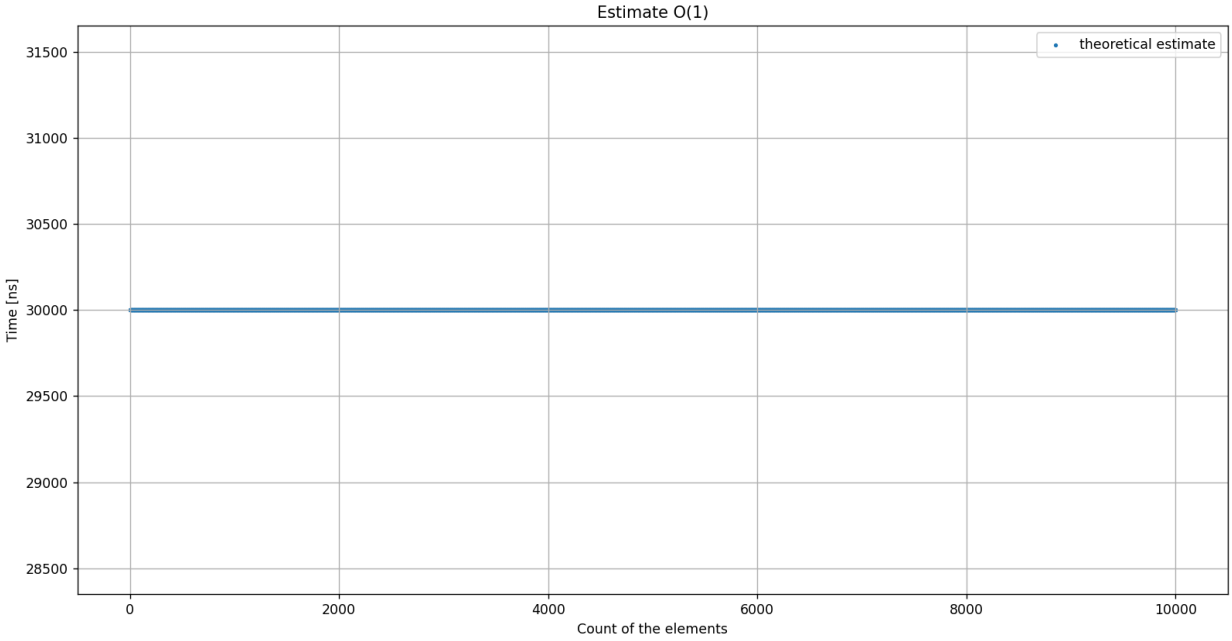

Рисунок 6 – лучший/средний случай операций хеш-таблицы

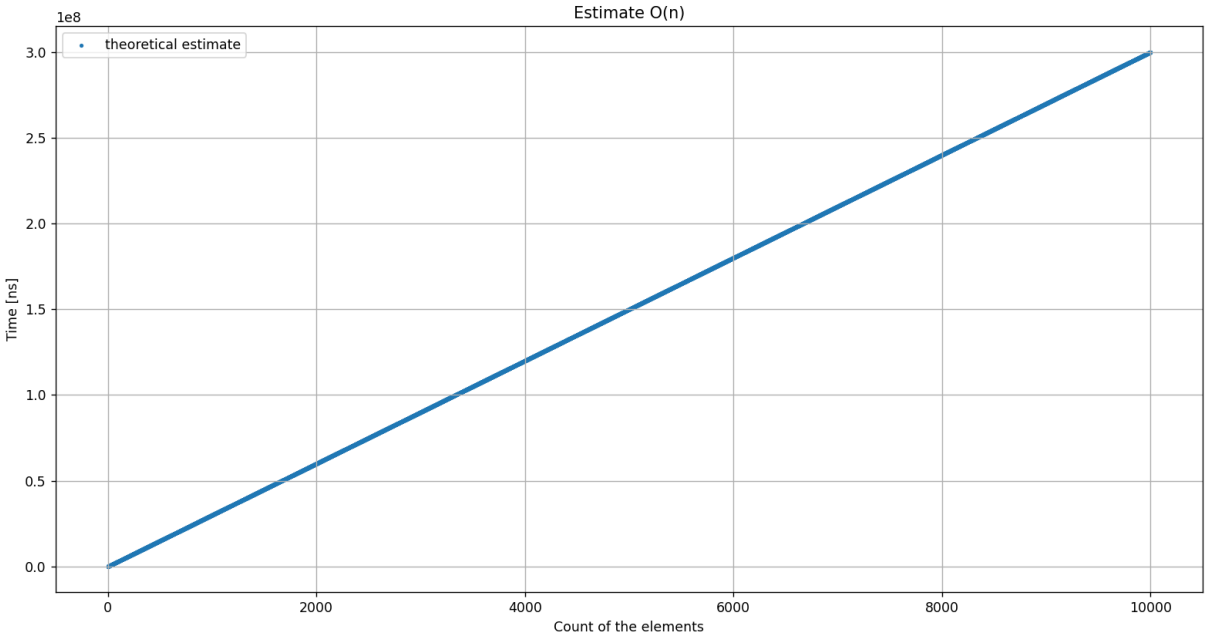

Рисунок 7 – худший случай операций хеш-таблицы

Были сгенерированы случайные данные и построен график зависимости времени выполнения от количества элементов в хеш-таблице во время вставки (см. рисунок 8, 9, 10), поиска (см. рисунок 11, 12, 13) и удаления (см. рисунок 14, 15, 16). Хеш-таблица используют открытую адресацию, в которой присутствует 3 вида пробирований. Таким образом, для каждой операции представлены теоретические и экспериментальные временные оценки для всех пробирований.

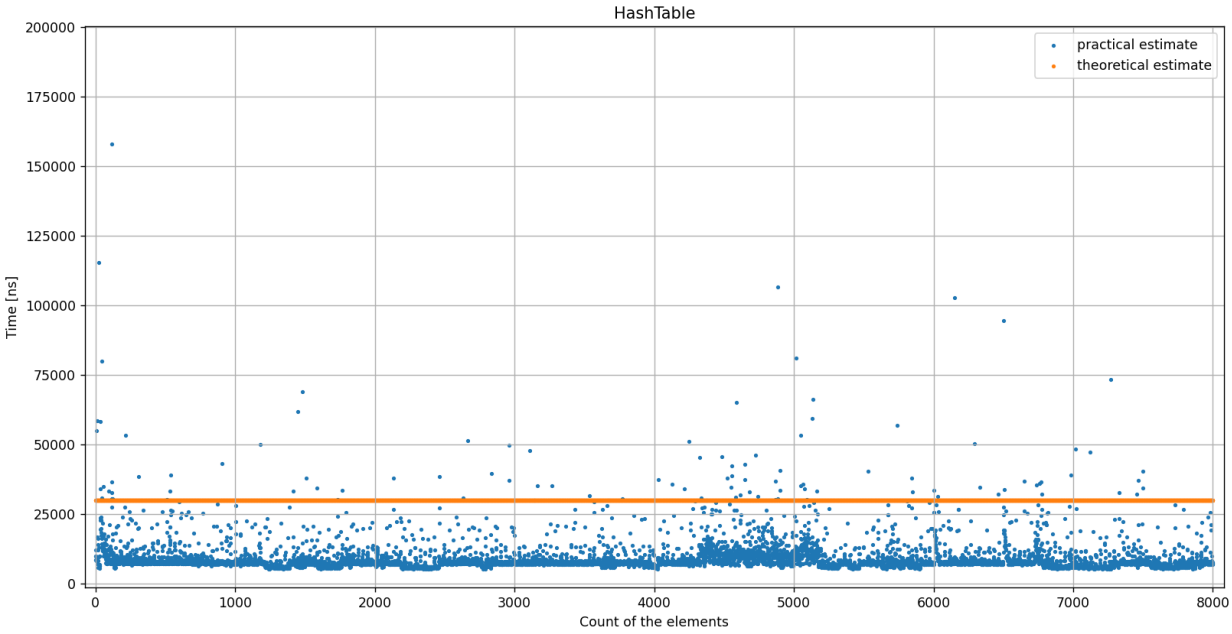

Рисунок 8 – теоретическая и практическая оценка операции вставки хеш-таблицы с двойным хеширование

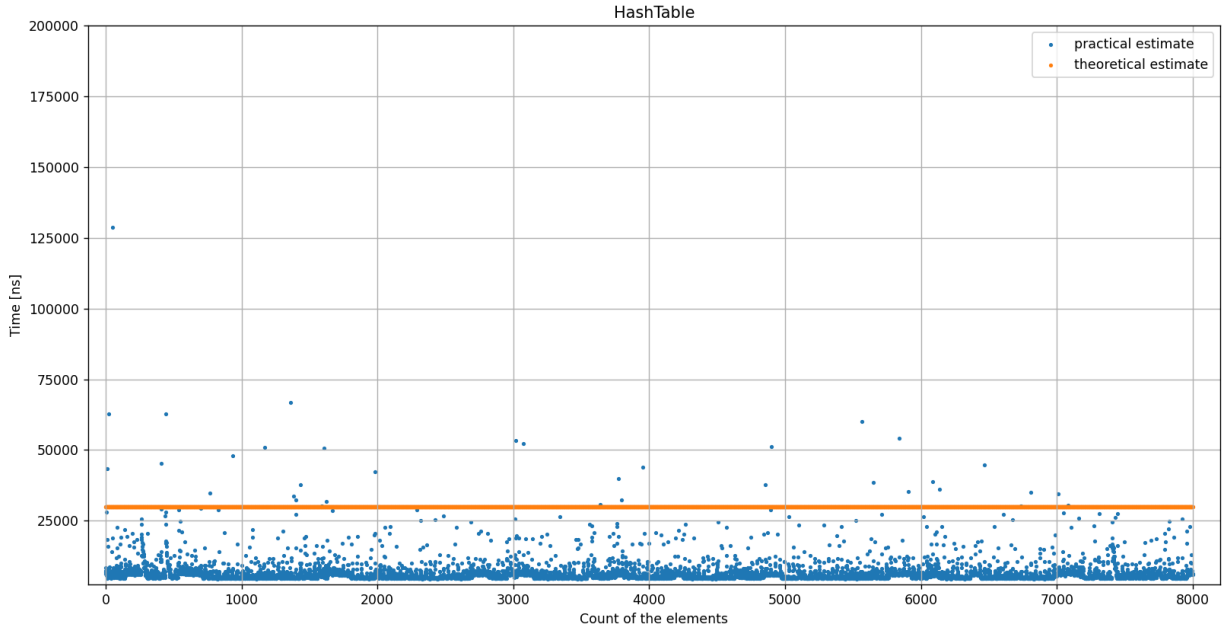

Рисунок 9 – теоретическая и практическая оценка операции вставки хеш-таблицы с квадратичным пробированием

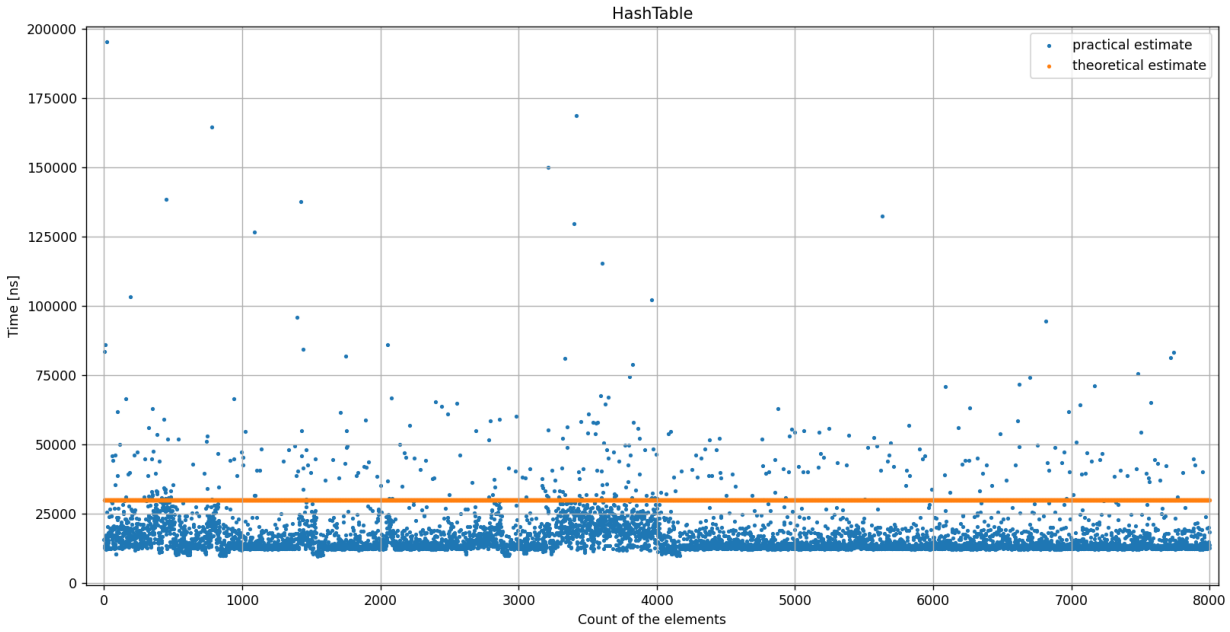

Рисунок 10 – теоретическая и практическая оценка операции вставки хеш-таблицы с линейным пробированием

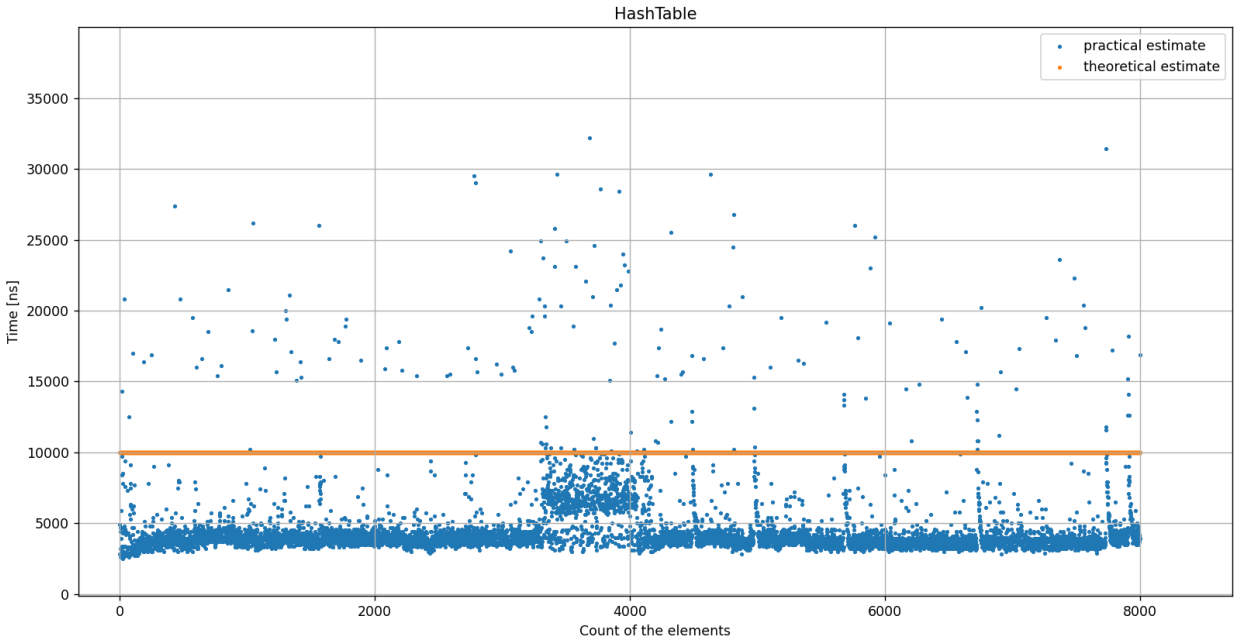

Рисунок 11 – теоретическая и практическая оценка операции поиска хеш-таблицы с двойным хеширование

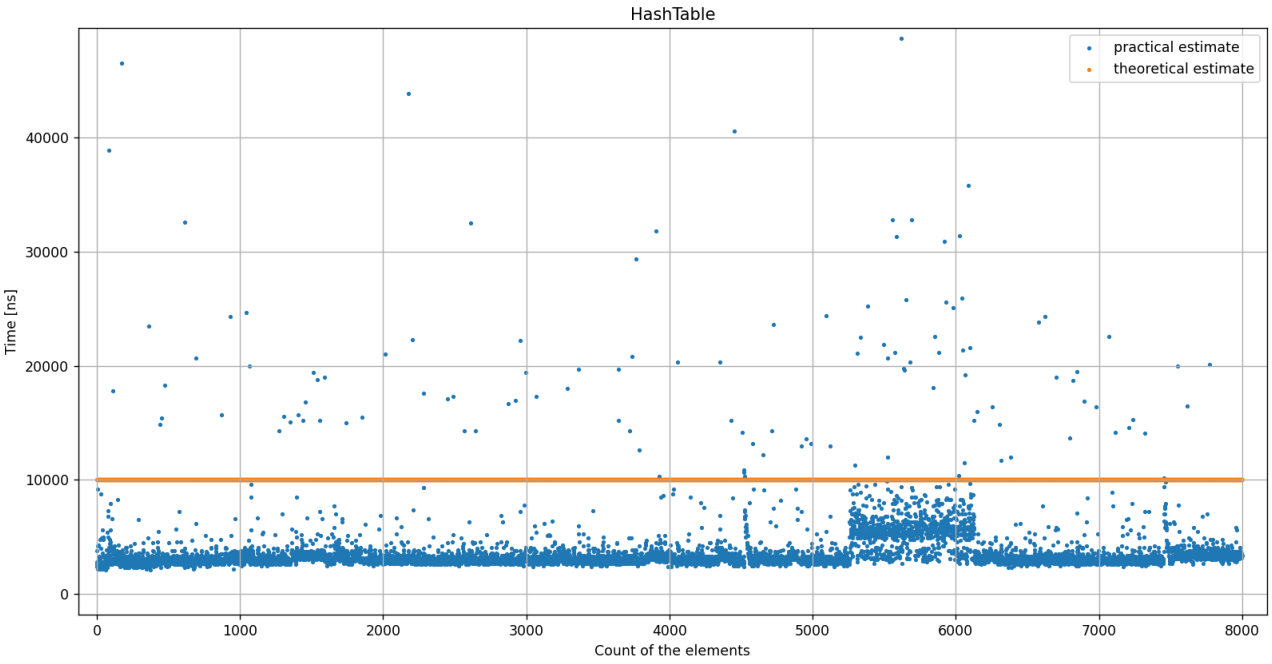

Рисунок 12 – теоретическая и практическая оценка операции поиска хеш-таблицы с квадратичным пробированием

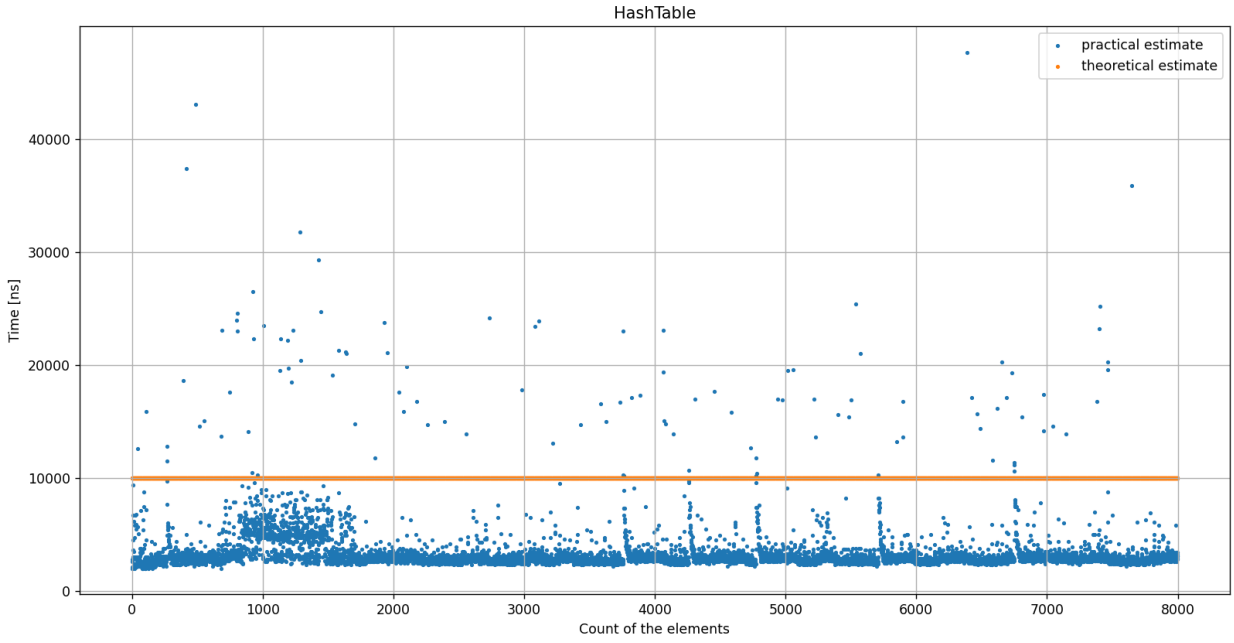

Рисунок 13 – теоретическая и практическая оценка операции поиска хеш-таблицы с линейным пробированием

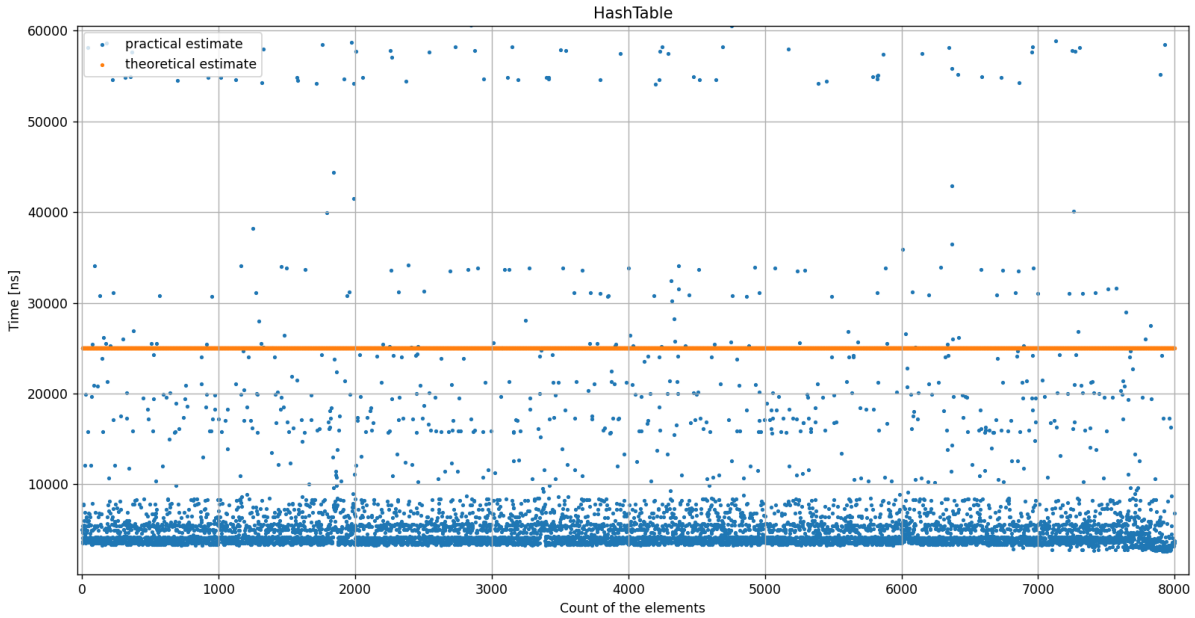

Рисунок 14 – теоретическая и практическая оценка операции удаления хеш-таблицы с двойным хеширование

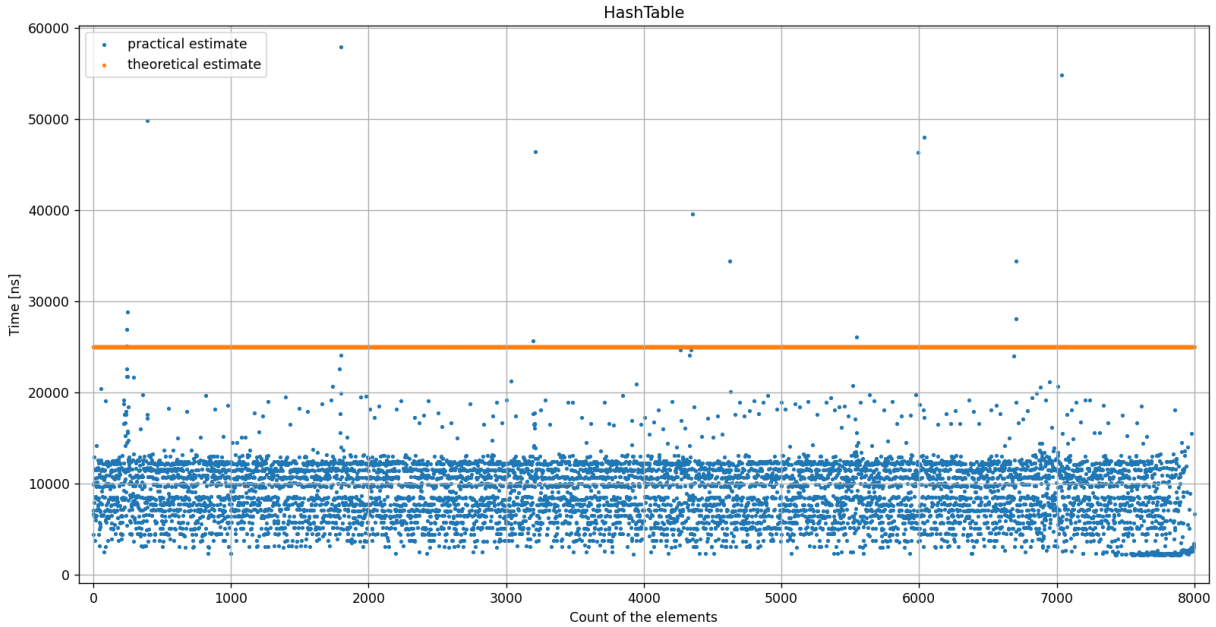

Рисунок 15 – теоретическая и практическая оценка операции удаления хеш-таблицы с квадратичным пробированием

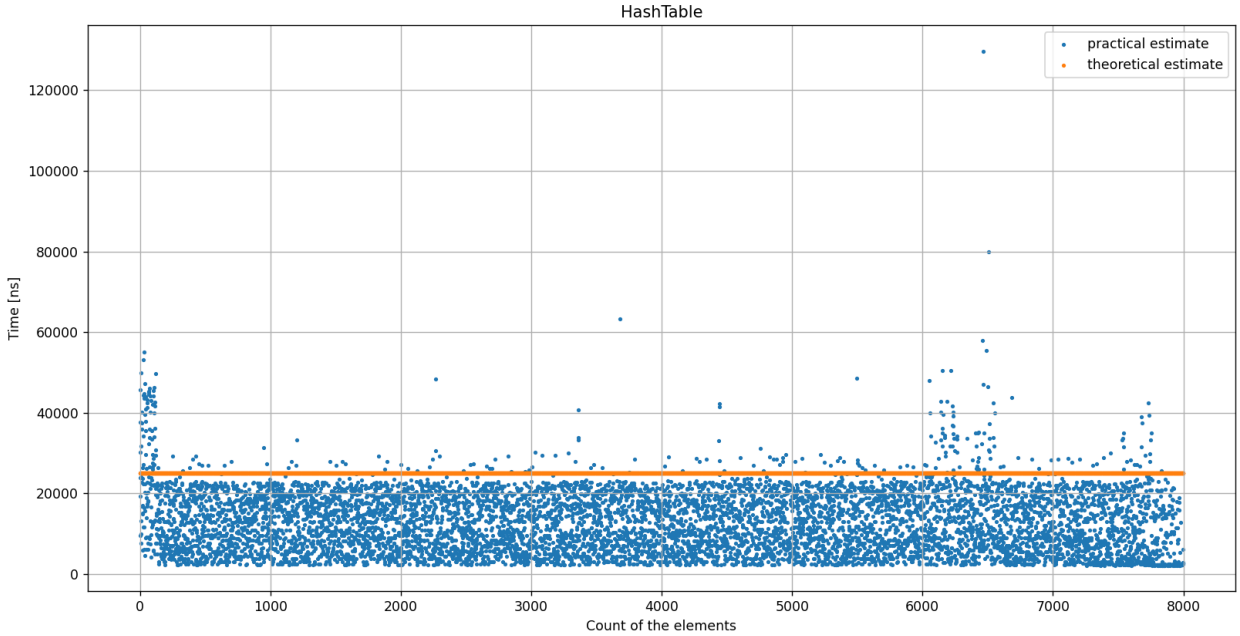

Рисунок 16 – теоретическая и практическая оценка операции удаления хеш-таблицы с линейным пробированием

По графикам можно сделать вывод, что квадратичное пробирование и двойное хеширование оказались наиболее эффективными, чем линейное пробирование.

2.2. Оценка АВЛ-дерева

Оценка сложности операции вставки, поиска и удаления для АВЛ-дерева представлен в таблице 2.

	Лучший случай	Среднее случай	Худший случай
Вставка	O(1)	O(log n)	O(log n)
Поиск	O(1)	O(log n)	O(log n)
Удаление	O(1)	O(log n)	O(log n)

Таблица 2 – оценка сложности операций в АВЛ-дереве

Были сгенерированы случайные тестовые данные в размере 10.000 элементов. на основе которых построен график зависимости времени выполнения от количества элементов в АВЛ-дереве во время вставки (см. рисунок 17), поиска (см. рисунок 18) и удаления (см. рисунок 19).

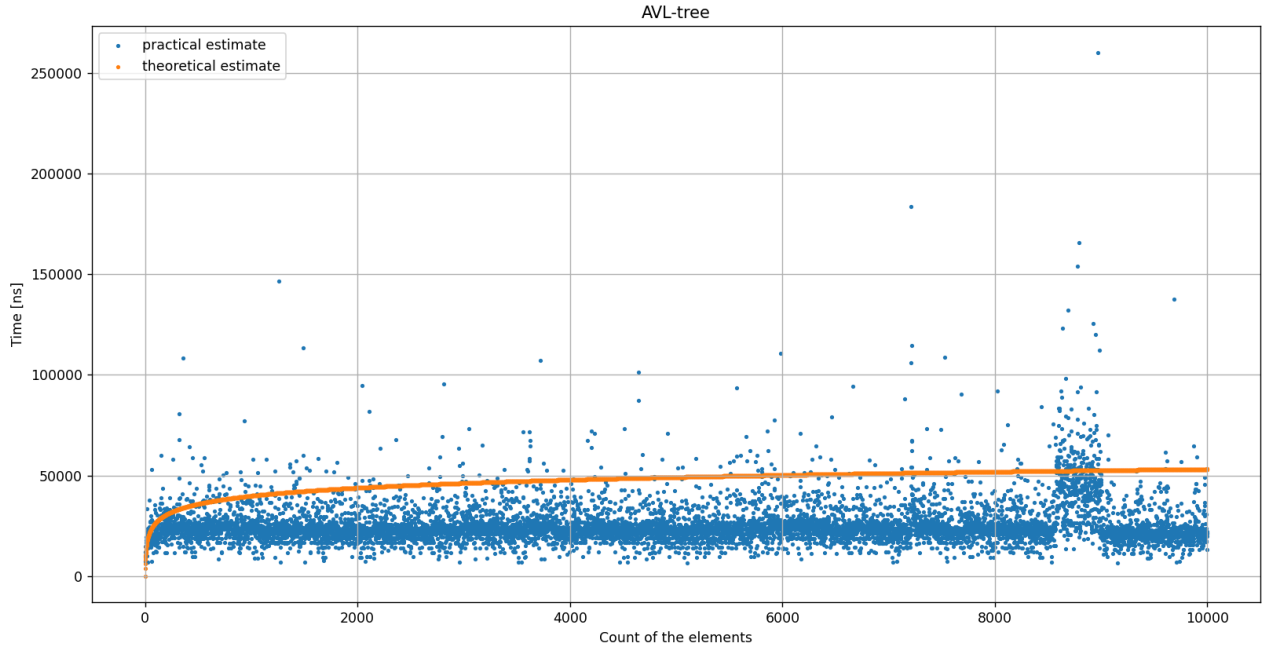

Рисунок 17 – теоретическая и практическая оценка операции вставки

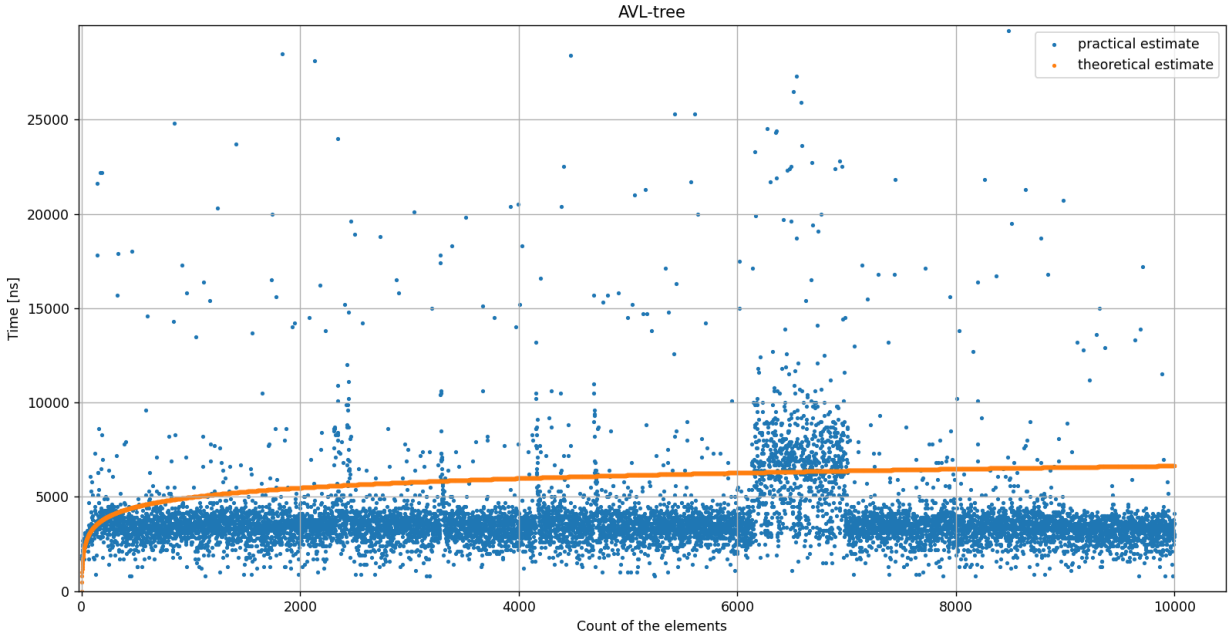

Рисунок 18 – теоретическая и практическая оценка операции поиска

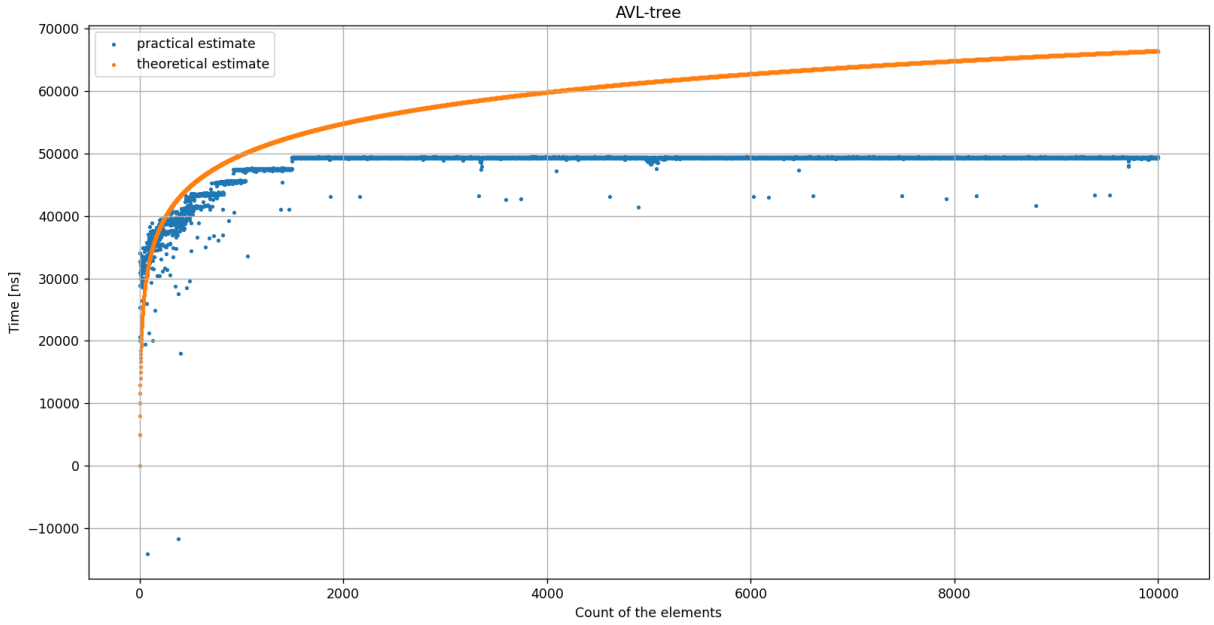

Рисунок 19 – теоретическая и практическая оценка операции удаления

Из графиков видно, что соблюдается логарифмическая сложность. Тем самым, АВЛ-дерево является стабильно эффективной структурой данных по времени.

ЗАКЛЮЧЕНИЕ

По результатам исследования хеш-таблицы с открытой адресацией и АВЛ-дерева можно сделать такой вывод, АВЛ-дерево имеет более стабильную оценку, равную O(log n), когда хеш-таблица может быть как быстрой, так и медленной. Также хеш-таблица занимает больше места, за счёт этого она преуспевает в скорости. Обе структуры являются эффективными для использования. Каждая имеет свои преимущества и недостатки, который и определяют их сферы использования.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Теоретические сведения о хеш-таблице с открытой адресацией // ru.wikipedia.org URL: https://ru.wikipedia.org/wiki/Хеш-таблица
2. АВЛ-деревья // habr.com URL: https://habr.com/ru/post/150732/
3. АВД-дерево // ru.wikipedia.org URL: https://ru.wikipedia.org/wiki/АВЛ-дерево
4. Решение коллизий // neerc.ifmo.ru URL: https://neerc.ifmo.ru/wiki/index.php?title=Разрешение_коллизий

ПРИЛОЖЕНИЕ А

КОД ПРОГРАММЫ

Название файла: avl.py

```
class Node:
    def __init__(self, key, value):
        self.key = key
        self.value = value
        self.l_child = None
        self.r_child = None
        self.height = 1

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

    def get_height(self, node):
        if not node:
            return 0
        return node.height

    def max_height(self, node):
        left = self.get_height(node.l_child)
        right = self.get_height(node.r_child)
        return max(left, right)

    def sl_rotate(self, node):
        right_node = node.r_child
        right_left_node = right_node.l_child
        node.r_child = right_left_node
        right_node.l_child = node

        node.height = self.max_height(node) + 1
        right_node.height = self.max_height(right_node) + 1
        return right_node

    def sr_rotate(self, node):
        left_node = node.l_child
        left_right_node = left_node.r_child
        node.l_child = left_right_node
        left_node.r_child = node

        node.height = self.max_height(node) + 1
        left_node.height = self.max_height(left_node) + 1
        return left_node

    def br_rotate(self, node):
        node.l_child = self.sl_rotate(node.l_child)
        return self.sr_rotate(node)


    def bl_rotate(self, node):
        node.r_child = self.sr_rotate(node.r_child)
        return self.sl_rotate(node)


    def save_balance(self, local_root):
        if self.get_height(local_root.l_child) - self.get_height(local_root.r_child) == 2:
            left = local_root.l_child
            if self.get_height(left.r_child) <= self.get_height(left.l_child):
                return self.sr_rotate(local_root)
            else:
                return self.br_rotate(local_root)
```

```
    elif self.get_height(local_root.r_child) - self.get_height(local_root.l_child) == 2:
        right = local_root.r_child
        if self.get_height(right.l_child) <= self.get_height(right.r_child):
            return self.sl_rotate(local_root)
        else:
            return self.bl_rotate(local_root)
    return local_root

def insert(self, key, value):
    self.root = self.find_place(Node(key, value), self.root)

def find_place(self, node, root):
    if root == None:
        return node
    elif node.key > root.key:
        root.r_child = self.find_place(node, root.r_child)
    elif node.key == root.key:
        root.value = node.value
    else:
        root.l_child = self.find_place(node, root.l_child)

    root = self.save_balance(root)
    root.height = self.max_height(root) + 1
    return root

def find(self, key, node):
    if node == None:
        return None
    elif key == node.key:
        return node.value
    elif key < node.key:
        return self.find(key, node.l_child)
    else:
        return self.find(key, node.r_child)

def max(self, node):
    if node.r_child == None:
        return node
    return self.max(node.r_child)

def min(self, node):
    if node.l_child == None:
        return node
    return self.min(node.l_child)

def delete(self, key):
    self.root = self.remove(key, self.root)

def remove(self, key, node):
    if node == None:
        return None
    elif key < node.key:
        node.l_child = self.remove(key, node.l_child)
    elif key > node.key:
        node.r_child = self.remove(key, node.r_child)
    else:
        if node.r_child == None:
            node = node.l_child
        else:
            mn = self.min(node.r_child)
            mn.r_child = self.remove(mn.key, node.r_child)
            mn.l_child = node.l_child
            node = mn
    if node != None:
        node.height = self.max_height(node) + 1
        node = self.save_balance(node)
```

```
        return node


    def print_info(self, node, height = 0):
        if height == 0 and node == None:
            print("Tree is empty")
        if node:
            self.print_info(node.r_child, height + 1)
            print('    ' * height + str(node.key) + " " + str(node.value))
                    self.print_info(node.l_child, height + 1)
```

Название файла: hash.py

```
class Cell:
    def __init__(self, key, value, type_cell):
        self.key = key
        self.value = value
        self.type = type_cell

    def __str__(self):
        return str(self.key) + " " + str(self.value)

class HashTable:
    def __init__(self, default_size):
        self.table = [Cell(None, None, False)] * default_size
        self.max_size = default_size
        self.size = 0

    def resize(self):
        self.size = 0
        used_table = self.table
        self.table = [Cell(None, None, False)] * self.max_size * 2
        self.max_size *= 2
        for idx in range(self.max_size // 2):
            if used_table[idx].type and used_table[idx].key != None:
                self.insert(used_table[idx].key, used_table[idx].value)


    def hash(self, num):
        return (num + 1) % self.max_size

    def linear_research(self, idx, i):
        return (idx + 7*i) % self.max_size

    def square_research(self, idx, i):
        return (idx + i*i) % self.max_size

    def double_hash_research(self, idx, i):
        second_hash = (idx + 1) * 6577 % self.max_size
        return (self.hash(idx) % self.max_size + (i * second_hash) % self.max_size) % self.max_size

    def research(self, idx, i):
        return self.double_hash_research(idx, i)

    def insert(self, key, value):
        key_hash = self.hash(key)
        for i in range(self.size + 1):
            index = self.research(key_hash, i)
            if not self.table[index].type or self.table[index].key == None:
                self.table[index] = Cell(key, value, True)
                self.size += 1
```

```
                break
            elif self.table[index].key == key:
                self.table[index].value = value
                break
        if self.size / self.max_size >= 2 / 3:
            self.resize()

    def find(self, key):
        key_hash = self.hash(key)
        for i in range(self.size + 1):
            index = self.research(key_hash, i)
            if not self.table[index].type:
                return None
            if self.table[index].type and self.table[index].key == key:
                return self.table[index].value

    def remove(self, key):
        key_hash = self.hash(key)
        for i in range(self.max_size):
            index = self.research(key_hash, i)
            if not self.table[index].type:
                break
            if self.table[index].type and self.table[index].key == key:
                self.table[index].key = None
                self.size -= 1
                break

    def print_info(self):
        for i in range(self.max_size):
            print(self.table[i])
                    print()
```