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: B-дерево и Хеш-таблица (двойное хеширование)

Студент гр. 1304 _____ Байков Е.С.

Преподаватель _____ Иванов Д.В.

Санкт-Петербург

2022

ЗАДАНИЕ
НА КУРСОВУЮ РАБОТУ

Студент Байков Е.С.

Группа 1304

Тема работы : Исследование структур данных: B-дерево и Хеш-таблица (двойное хеширование).

Исходные данные:
Реализовать B-дерево и Хеш-таблицу с открытой адресацией.
Провести исследование. Входные данные генерируются линейно.

Содержание пояснительной записки:
«Содержание», «Введение», «Основные теоретические сведения», «Реализация структур», «Теоретическая оценка сложности базовых операций», «Сравнение теоретических значений с экспериментальными значениями», «Заключение», «Список использованных источников»

Дата выдачи задания: 25.10.2022

Дата сдачи реферата: 24.12.2022

Дата защиты реферата: 24.12.2022

Студент _____ Байков Е.С.

Преподаватель _____ Иванов Д.В.

АННОТАЦИЯ

В курсовой работе реализованы такие структуры данных, как B-дерево и хеш-таблица, использующая метод двойного хеширования для предотвращения коллизий. Сгенерированы входные данные для исследования лучшего, среднего и худшего случаев работы операций вставки, удаления и поиска элемента. Сравнение экспериментальных значений с теоретическими. Построены графики зависимости времени работы базовых операций от количества элементов в структуре.

SUMMARY

The course work implements data structures such as a B-tree and a hash table using the double hashing method to prevent collisions. Input data has been generated to investigate the best, average and worst cases of insertion, deletion and element search operations. Comparison of experimental values with theoretical ones. Graphs of the dependence of the operating time of basic operations on the number of elements in the structure are constructed.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Целью данной работы является исследование двух структур данных: хеш-таблицы с двойным хешированием и B-дерева.

Исследование заключается в реализации требуемых структур данных, базовых алгоритмов для каждой из структур, а также сравнении теоретических и экспериментальных значений работы данных алгоритмов.

На основе выше изложенной цели, были сформулированы следующие задачи:

1. Изучение таких структур данных, как хеш-таблица с двойным хешированием и B-дерево.
2. Изучение особенностей работы с данными структурами данных.
3. Сравнение теоретических и экспериментальных значений алгоритмов.

1. ОСНОВНЫЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

1.1. B-деререво.

B-дерево – структура данных, дерево поиска. С точки зрения внешнего логического представления – сбалансированное, сильно ветвистое дерево. Особый тип самобалансирующегося дерева поиска, в котором каждый узел может содержать более одного ключа и может иметь более двух дочерних элементов (см. рисунок 1).

Использование B-деревьев впервые было предложено Р. Бэйером (англ. R. Bayer) и Э. МакКрейтом (англ. E. McCreight) в 1970 году.

Сбалансированность означает, что длины любых двух путей от корня до листьев различаются не более, чем на единицу.

Ветвистость дерева — это свойство каждого узла дерева ссылаться на большое число узлов-потомков.

С точки зрения физической организации B-дерево представляется как мультисписочная структура страниц памяти, то есть каждому узлу дерева соответствует блок памяти (страница). Внутренние и листовые страницы обычно имеют разную структуру.

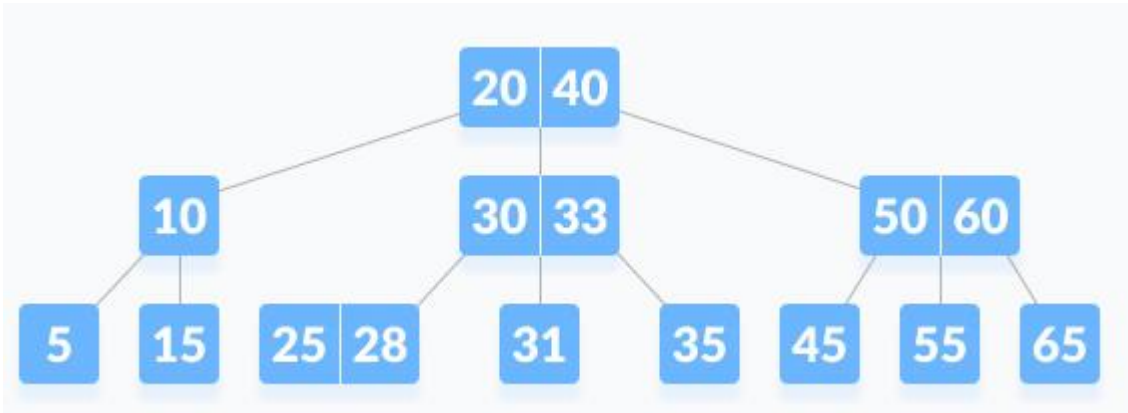

Рисунок 1 – B-дерево

Структура и принципы построения:

1) Ключи в каждом узле обычно упорядочены для быстрого доступа к ним. Корень содержит от 1 до 2t-1 ключей. Любой другой узел содержит от t-1 до 2t-1 ключей. Листья не являются исключением из этого правила. Здесь t —

параметр дерева, не меньший 2 (и обычно принимающий значения от 50 до 2000).

2) У листьев нет потомков. Любой другой его узел, содержащий ключи k_i ($i = 1, \dots, n$), содержит $n+1$ потомков, при этом:

- Первый потомок и все его потомки содержат ключи из интервала $(-\infty; k_1)$.
- Для $2 \le i \le n$, i-й потомок и все его потомки содержат ключи из интервала $(k_{i-1}; k_i)$
- $(n+1)$-й потомок и все его потомки содержат ключи из интервала $(k_n; +\infty)$.

3) Глубина всех листьев одинакова.

Свойство 2 можно сформулировать иначе: каждый узел B-дерева, кроме листьев, можно рассматривать как упорядоченный список, в котором чередуются ключи и указатели на потомков.

Базовые операции:

- Поиск по ключу
- Добавление ключа
- Удаление ключа

Основные достоинства:

- Во всех случаях полезное использование пространства вторичной памяти составляет свыше 50%. С ростом степени полезного использования памяти не происходит снижения качества обслуживания.
- Произвольный доступ к записи реализуется посредством малого количества подопераций (обращения к физическим блокам).
- В среднем достаточно эффективно реализуются операции включения и удаления записей; при этом сохраняется естественный порядок ключей с целью последовательной обработки, а также соответствующий баланс дерева для обеспечения быстрой произвольной выборки.

- Неизменная упорядоченность по ключу обеспечивает возможность эффективной пакетной обработки.

Основной недостаток B-деревьев состоит в отсутствии для них эффективных средств выборки данных (то есть метода обхода дерева), упорядоченных по свойству, отличному от выбранного ключа.

1.2. Хеш-таблица (открытая адресация).

Хеш-таблица — это структура данных, реализующая интерфейс ассоциативного массива, а именно, она позволяет хранить пары (ключ, значение) (см. рисунок 2).

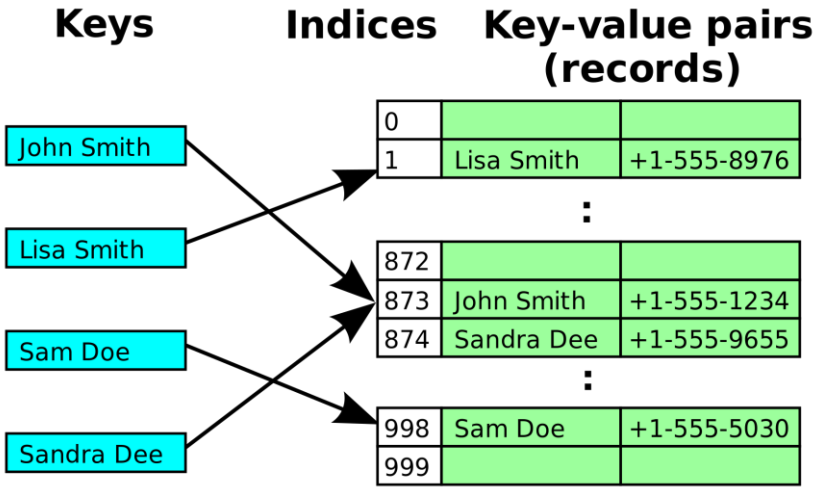

Рисунок 2 – Хеш-таблица

Выполнение операции в хеш-таблице начинается с вычисления *хеш-функции* от ключа. Хеш-значение - индекс в таблице. Принято считать, что хорошей, с точки зрения практического применения, является хеш-функция, которая удовлетворяет следующим условиям:

- Функция должна быть простой с вычислительной точки зрения;
- Функция должна распределять ключи в хеш-таблице наиболее равномерно;
- Функция не должна отображать какую-либо связь между значениями ключей в связь между значениями адресов;
- Функция должна минимизировать число коллизий;

Коллизии – совпадение хеш-значения при разных ключах.

Есть несколько способов разрешения коллизий:

- Метод цепочек;
- Метод открытой адресации;

В данной работе будет рассмотрен только метод открытой адресации с пробированием при помощи двойного хеширования.

В массиве H хранятся сами пары ключ-значение. Алгоритм вставки элемента проверяет ячейки массива H в некотором порядке до тех пор, пока не будет найдена первая свободная ячейка, в которую и будет записан новый элемент. Этот порядок вычисляется на лету, что позволяет сэкономить на памяти для указателей, требующихся в хеш-таблицах с цепочками.

Последовательность, в которой просматриваются ячейки хеш-таблицы, называется последовательностью проб. В общем случае, она зависит только от ключа элемента, то есть это последовательность $h_0(x)$, $h_1(x)$, …, $h_{n-1}(x)$, где x — ключ элемента, а $h_i(x)$ — произвольные функции, сопоставляющие каждому ключу ячейку в хеш-таблице. Первый элемент в последовательности, как правило, равен значению некоторой хеш-функции от ключа, а остальные считаются от него одним из приведённых ниже способов. Для успешной работы алгоритмов поиска последовательность проб должна быть такой, чтобы все ячейки хеш-таблицы оказались просмотренными ровно по одному разу.

Алгоритм поиска просматривает ячейки хеш-таблицы в том же самом порядке, что и при вставке, до тех пор, пока не найдется либо элемент с искомым ключом, либо свободная ячейка (что означает отсутствие элемента в хеш-таблице).

Удаление элементов в такой схеме несколько затруднено. Обычно поступают так: заводят булевый флаг для каждой ячейки, помечающий, удален элемент в ней или нет. Тогда удаление элемента состоит в установке этого флага для соответствующей ячейки хеш-таблицы, но при этом необходимо модифицировать процедуру поиска существующего элемента так, чтобы она считала удалённые ячейки занятыми, а процедуру добавления — чтобы она их считала свободными и сбрасывала значение флага при добавлении.

Двойное хеширование: интервал между ячейками фиксирован, как при линейном пробировании, но, в отличие от него, размер интервала вычисляется второй, вспомогательной хеш-функцией, а значит, может быть различным для

разных ключей. Значения этой хеш-функции должны быть ненулевыми и взаимно-простыми с размером хеш-таблицы, что проще всего достичь, взяв простое число в качестве размера, и потребовав, чтобы вспомогательная хеш-функция принимала значения от 1 до N — 1.

Базовые операции, поддерживающиеся данной структурой данных:

- Операция добавления новой пары (ключ, значение);
- Операция поиска;
- Операция удаления пары по ключу;

2. РЕАЛИЗАЦИЯ СТРУКТУР ДАННЫХ

2.1. B-дерево

Для реализации B-дерева создан классы *BTree* – класс дерева, и *BTreeNode* – класс узла.

BTreeNode:

1. *__init__*: инициализирует узел, принимая на вход булевое значение равное по умолчанию *False*. Также создаются списки *keys* и *child*, в которых будут хранится ключи и дети соответственно.
2. *__len__*: переопределенный метод для возврата длины, который возвращает количество ключей в узле.
3. *__search_index*: метод, который возвращает либо индекс ключа внутри узла либо индекс указывающий на ребенка, в котором может хранится искомый ключ.
4. *find*: возвращает значение найденного индекса.

BTree:

1. *__init__*: инициализирует дерево, с заданным пользователем параметром *t*, а также создает пустой узел со значением *True*.
2. *insert*: осуществляет вставку значения в дерево. Если на данный момент дерево является пустым, то вставка будет производится в корень дерева. Если же размер корневого элемента равен максимальному допустимому значению, то будет происходить разделение корня на левый и правый узел, а центральное значение станет новым корнем.
3. *split_child*: вспомогательный метод, производящий разделение на правого и левого ребенка, а также создающий нового родителя правого и левого ребенка, который изначально был центральным элементом, поданного на вход узла. Также в случае, если узел не являлся листом, то его дети также делятся.
4. *insert_non_full*: осуществляет добавление в би-дерево согласно его свойствам.
5. *find*: реализует поиск элемента, вызывается рекурсивно.

6. *delete*: осуществляет удаление элемента, используя вспомогательные методы *__remove, merge_children, move_key_left, move_key_right*.
7. *print_tree*: метод для вывода дерева.

2.2. Хеш-таблица (двойное хэширование)

Для реализации хеш-таблицы создан класс *HashTable*.

HashTable:

1. *__init__*: создает таблицу, с заданным пользователем размером, по умолчанию равен 128, а также определяет некоторые поля класса.
2. *first_hash_func*: хеширование при помощи умножения в кольце вычетов по размеру таблицы.
3. *second_hash_func*: метод для хеширования. Умножает значение ключа на константу и возвращает взаимно простое с размером таблицы число.
4. *find*: осуществляет поиск данных по ключу в таблице. Вычисляется первый хеш и сравниваются сами ключи, если они не сходятся то применяется двойное хеширование для поиска необходимого элемента или обнаружения, что его нет в таблице.
5. *_find*: метод аналогичный предыдущему, но вместо возврата данных возвращает хеш, по которому записаны искомые данные.
6. *_insert*: метод для вставки при рехеширование таблицы.
7. *insert*: метод для вставки элемента в таблицу. Высчитывается хеш затем проверяется свободно ли значение в таблице по этому адресу, если да то записывается в таблицу, иначе происходит двойное хеширование, с помощью которого и находится свободная ячейка.
8. *delete*: с помощью функции *_find* находится место, где сохранено нужное значение и удаляется.
9. *resize*: высчитывается α – коэффициент заполнения равный отношению количества элементов в таблице к размеру. Если коэффициент больше 70%, то происходит расширение и рехеширование таблицы. Если коэффициент меньше 30% и размер таблицы не меньше 128, то происходит рехеширование.

3. ТЕОРЕТИЧЕСКАЯ ОЦЕНКА СЛОЖНОСТИ БЫЗОВЫХ ОПЕРАЦИЙ

3.1. B-дерево

Оценка сложности базовых операций B-дерева см. таблицу 1.

Таблица 1 – оценка сложности базовых операций B-дерева

	Лучший случай	Средний случай	Худший случай
Вставка	O(1) или $O(t\log_t n)$	$O(t\log_t n)$	$O(t\log_t n)$
Поиск	O(1) или $O(t\log_t n)$	$O(t\log_t n)$	$O(t\log_t n)$
Удаление	O(1) или $O(t\log_t n)$	$O(t\log_t n)$	$O(t\log_t n)$

Высота дерева логарифмически зависит от числа его узлов. Базовые операции над двоичными деревьями поиска (поиск, вставка и удаление узлов) напрямую зависят от его высоты, поэтому данные операции имеют логарифмическую зависимость времени работы от числа ключей в дереве. График см. Рисунок 3.

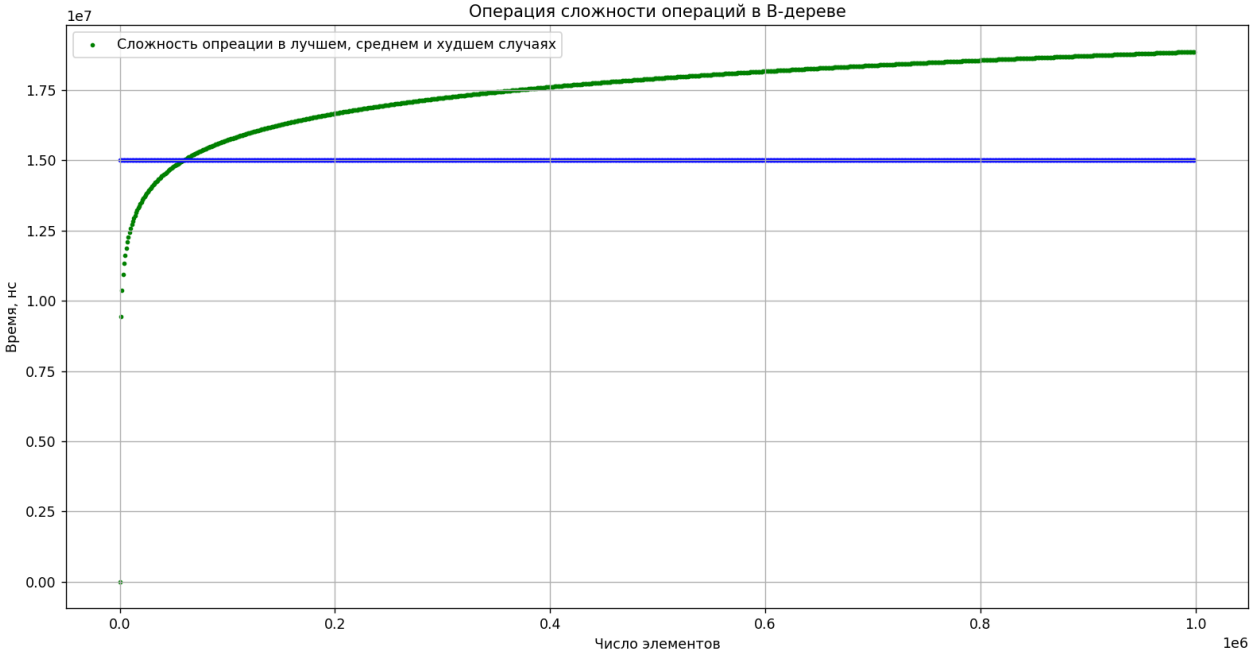

Рисунок 3 – График зависимости времени работы базовых операций B-дерева

3.2. Хеш-таблица (открытая адресация)

Оценка сложности базовых операций хеш-таблицы см. таблицу 2.

Таблица 2 – оценка сложности базовых операций хеш-таблицы

	Лучший случай	Средний случай	Худший случай
Вставка	O(1)	O(1)	O(1) или O(n)
Поиск	O(1)	O(1)	O(n)
Удаление	O(1)	O(1)	O(n)

В худшем случае в исследовании время работы операции вставки нельзя сказать однозначно, какое будет затрачено время, так как может возникнуть ситуация, когда большая часть таблицы будет заполнена и ее придется расширять, а при расширении придется производить рехеширование, что занимает время O(n).

График, демонстрирующий лучший случай см. рисунок 3

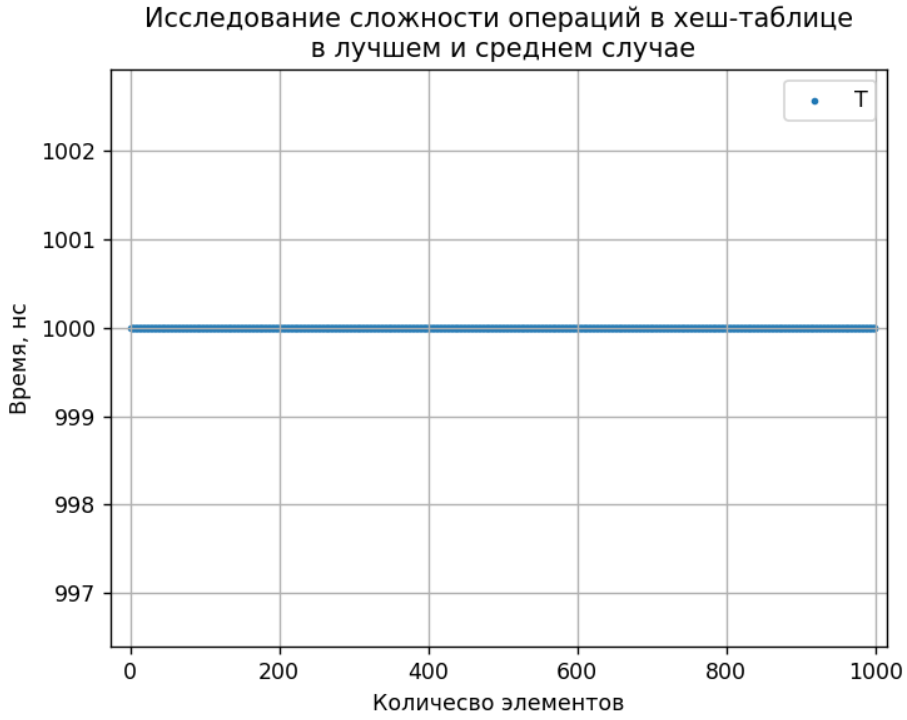

Рисунок 4 - График зависимости времени работы базовых операций хеш-таблицы в лучшем случае и среднем случае.

График, демонстрирующий худший случай см. рисунок 5.

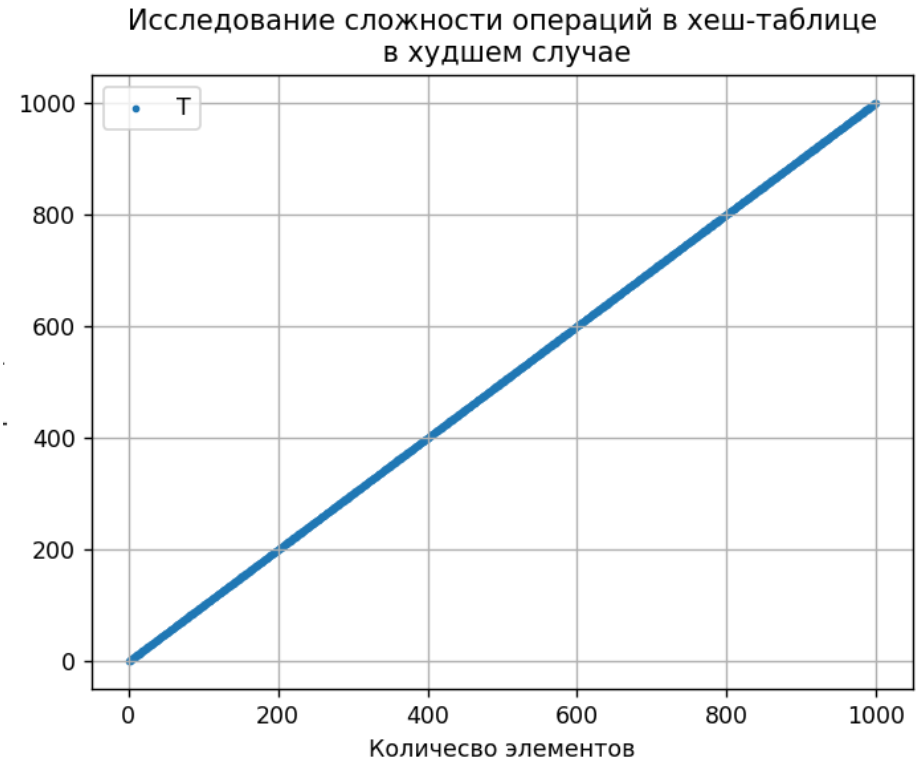

Рисунок 5 - График зависимости времени работы базовых операций хеш-таблицы в худшем случае

4. СРАВНЕНИЕ ТЕОРЕТИЧЕСКИХ ЗНАЧЕНИЙ С ЭКСПЕРИМЕНТАЛЬНЫМИ ЗНАЧЕНИЯМИ

4.1. B-дерево

Для тестирования в худшем и среднем случае взято малое значение t = 3 и количество значений равное 1000000. Результат тестирования см. рисунок 6.

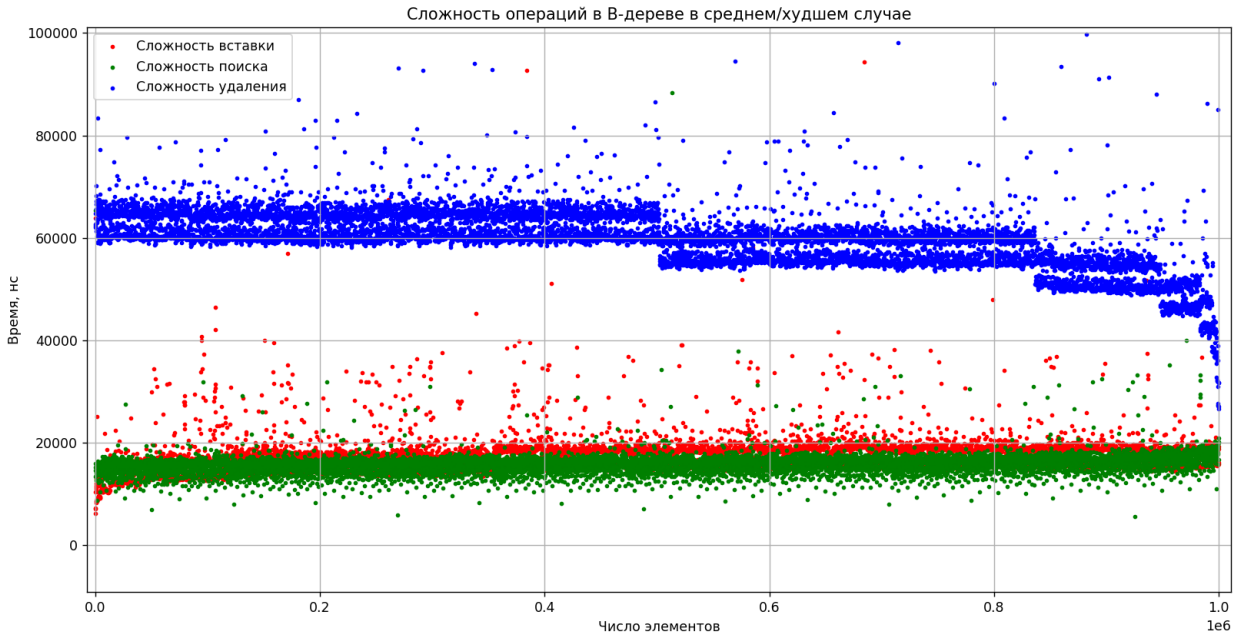

Рисунок 6 – демонстрация сложности операций в B-дереве в среднем/худшем случае.

Для тестирования лучшего случая взято число t = 50. Результат см. рисунок 7.

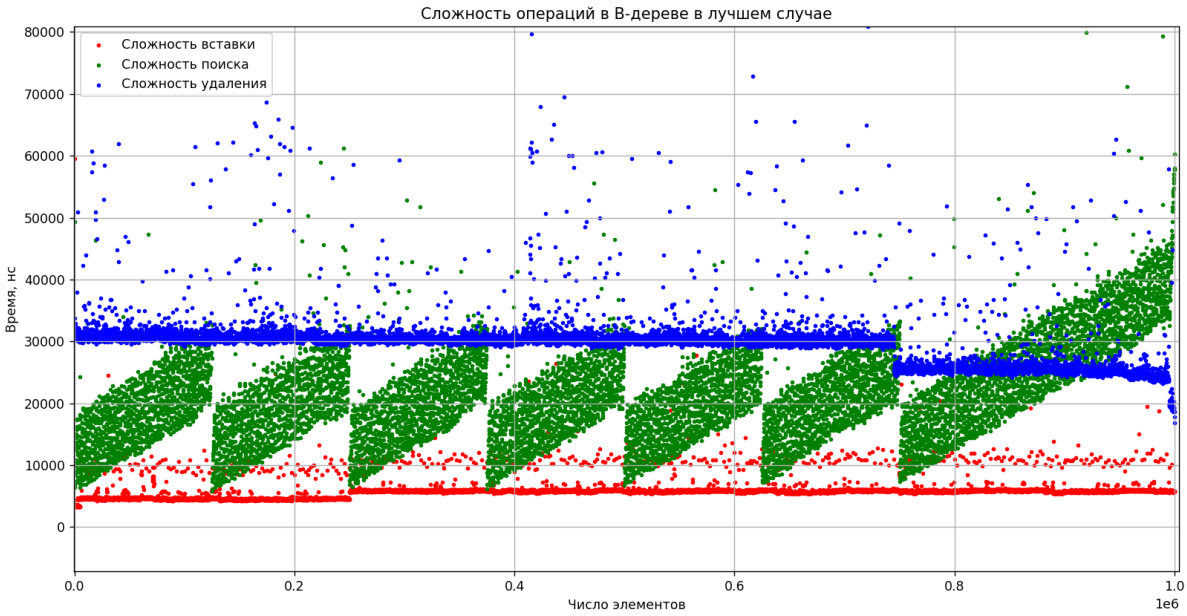

Рисунок 7 – тестирование в лучшем случае.

4.2. Хеш-таблица (двойное хеширование)

Для тестирования хеш-таблицы при лучшем/среднем случае взяты числа отсортированные в порядке возрастания, а также использована «хорошая» хеш-функция. Результаты тестирования см. рисунки 8, 9, 10.

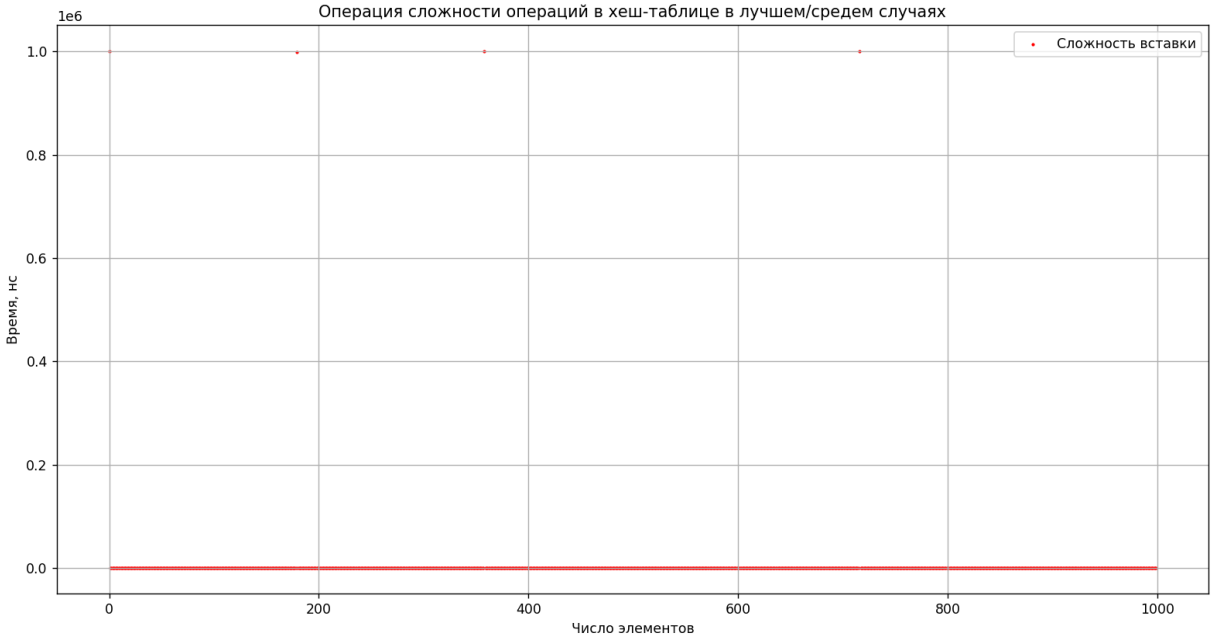

Рисунок 8 – тестирование вставки в хеш-таблицу в лучшем/среднем случаях.

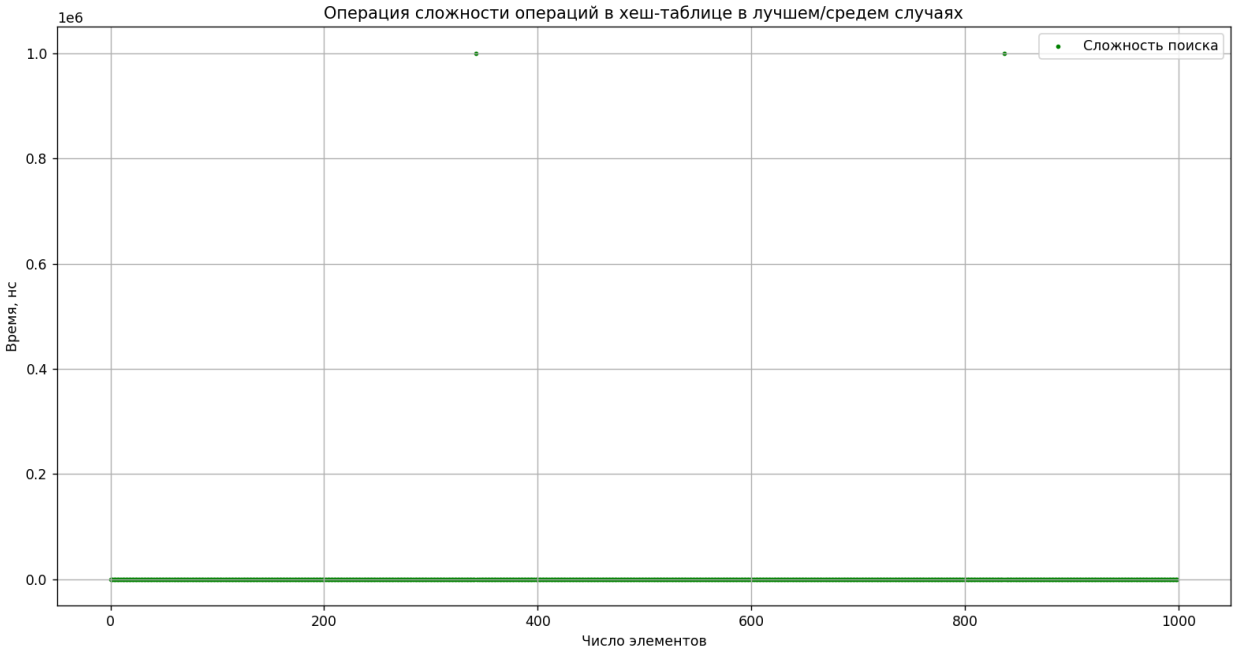

Рисунок 9 – тестирование поиска в лучшем/среднем случаях.

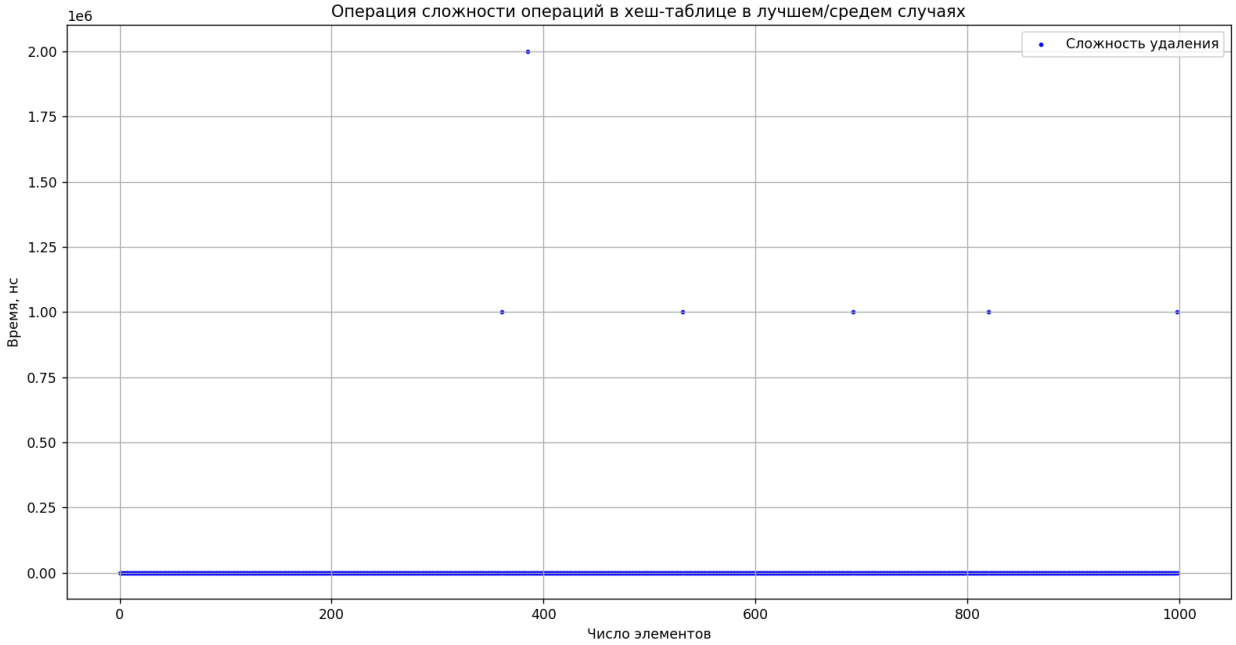

Рисунок 10 – сложность удаления в лучшем/среднем случаях.

В худшем случае заполним таблицу ключами, которые будут равны в кольце равном размеру таблицы и значениями, которые генерируются случайно. Далее произведем поиск элементов и их удаление. Результат тестирование см. рисунок 9.

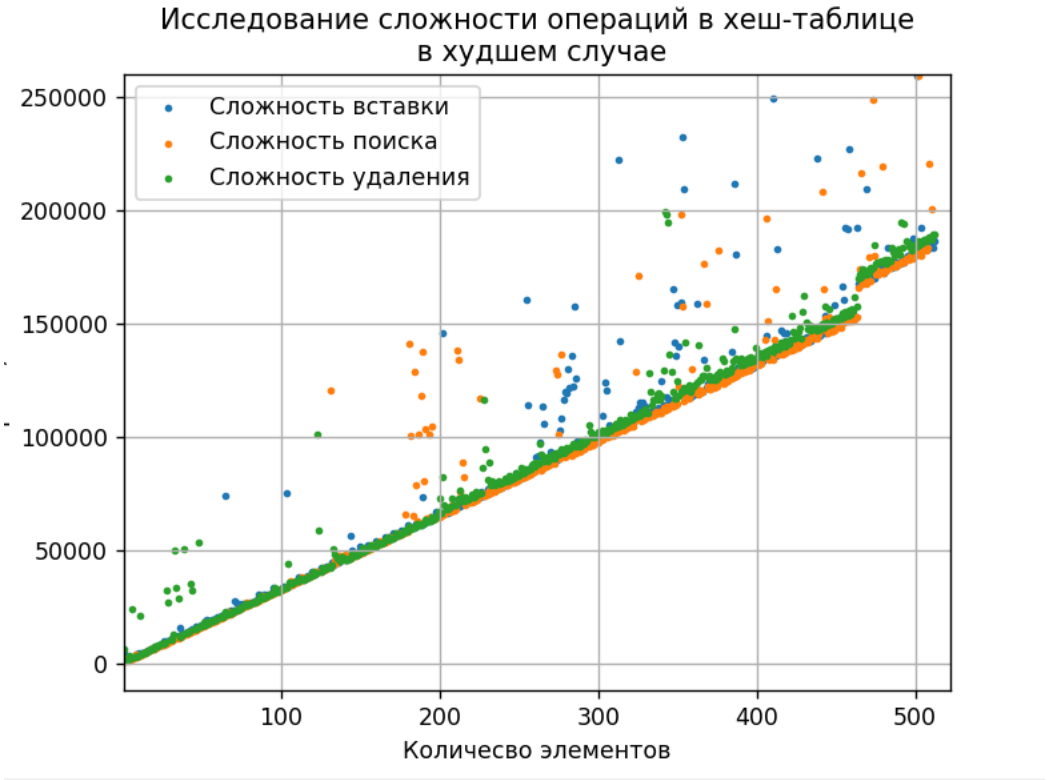

Рисунок 11 – График зависимости времени работы базовых операций хеш-таблицы в худшем случае

На рисунке 11 видно, что возникают коллизии, так как таблица состоит из одинаковых ключей в кольце. Наблюдается линейная зависимость O(n).

ЗАКЛЮЧЕНИЕ

В ходе курсовой работы реализованы такие структуры данных, как B-дерево и хеш-таблица с двойным хешированием, а также базовые операции над ними на языке Python.

Проведено исследование данных структур данных, время работы операций вставки, удаления и поиска в лучшем, среднем и худшем случаях. Сравнение теоретических значений с экспериментальными значениями, полученными в ходе работы.

В ходе исследования установлено, что экспериментальные значения соответствуют теоретическим значениям с небольшими погрешностями вызванными характеристиками использованного ПК.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Теоретические сведения о хеш-таблице с двойным хешированием // ru.wikipedia.org URL: https://ru.wikipedia.org/wiki/Хеш-таблица

2. Теоретические сведения о хеш-таблице с открытой адресацией // ru.wikipedia.org URL: https://ru.wikipedia.org/wiki/B-дерево

3. B-деревья // programiz.com URL: https://www.programiz.com/dsa/b-tree

4. Построение графиков. Библиотека matplotlib // matplotlib.org URL: https://matplotlib.org/stable/tutorials/introductory/pyplot.html

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: btree.py

```
import sys
sys.setrecursionlimit(100000)

class BTreeNode:
    def __init__(self, leaf=False) -> None:
        self.leaf = leaf
        self.keys = []
        self.child = []

    def __len__(self):
        return len(self.keys)

    def __search_index(self, key):
        left = -1
        right = len(self)
        while right - left > 1:
            middle = (left + right) // 2
            if self.keys[middle][0] < key:
                left = middle
            else:
                right = middle
        return right

    def find(self, key):
        return self.__search_index(key)


class BTree:
    def __init__(self, t=4) -> None:
        self.root = BTreeNode(True)
        self.t = t

    def insert(self, data):
```

```
        root = self.root
        if len(root.keys) == 2 * self.t - 1:
            temp = BTreeNode()
            self.root = temp
            temp.child.insert(0, root)
            self.split_child(temp, 0)
            self.insert_non_full(temp, data)
        else:
            self.insert_non_full(root, data)

    def insert_non_full(self, node: BTreeNode, data):
        i = len(node.keys) - 1
        if node.leaf:
            node.keys.append([None, None])
            while i >= 0 and data[0] < node.keys[i][0]: # Insert if key
less that current key
                node.keys[i + 1] = node.keys[i]
                i -= 1
            node.keys[i + 1] = data
        else:
            while i >= 0 and data[0] < node.keys[i][0]:
                i -= 1
            i += 1
            if len(node.child[i].keys) == 2 * self.t - 1:
                self.split_child(node, i)
                if data[0] > node.keys[i][0]:
                    i += 1
            self.insert_non_full(node.child[i], data)

    def split_child(self, node, i):
        t = self.t
        left_children = node.child[i]
        right_children = BTreeNode(left_children.leaf)
        node.child.insert(i + 1, right_children)
        node.keys.insert(i, left_children.keys[t - 1])
        right_children.keys = left_children.keys[t: (2 * t) - 1]
        left_children.keys = left_children.keys[0: t - 1]
        if not left_children.leaf:
```

```
            right_children.child = left_children.child[t: 2 * t]
            left_children.child = left_children.child[0: t]

    def print_tree(self, node, level=0):
        print("Level ", level, " ", len(node.keys), end=":")
        for i in node.keys:
            print(i, end=" ")
        print()
        level += 1
        if len(node.child) > 0:
            for i in node.child:
                self.print_tree(i, level)

    def find(self, key: int, node: None|BTreeNode=None):
        if node is not None:
            i = 0
            while i < len(node.keys) and key > node.keys[i][0]:
                i += 1
            if i < len(node.keys) and key == node.keys[i][0]:
                return node.keys[i]
            elif node.leaf:
                return None
            else:
                return self.find(key, node.child[i])
        else:
            return self.find(key, self.root)

    def __remove(self, node: BTreeNode, index: int):
        if node.leaf:
            node.keys.pop(index)
            return
        left = node.child[index]
        right = node.child[index + 1]
        if max(len(left), len(right)) == self.t - 1:
            new_index = len(left)
            new_node = self.merge_children(node, index)
        elif len(left) > len(right):
            new_node = self.move_key_right(node, index)
```

```
            new_index = 0
        else:
            new_node = self.move_key_left(node, index)
            new_index = len(new_node) - 1
        self.__remove(new_node, new_index)


    def delete(self, node: BTreeNode, key: int):
        if self.find(key) == None:
            return
        current_node = self.root
        while current_node:
            index = current_node.find(key)
            if index < len(current_node) and current_node.keys[index][0]
== key:
                self.__remove(current_node, index)
                break

            next_node = current_node.child[index]
            if len(next_node) == self.t - 1:
                if index != len(current_node) and \
                    (index == 0 or len(current_node.child[index + 1]) >
len(current_node.child[index - 1])):
                    if len(next_node) == len(current_node.child[index +
1]):
                        next_node = self.merge_children(current_node,
index)
                    else:
                        next_node = self.move_key_left(current_node,
index)
                else:
                    if len(next_node) == len(current_node.child[index -
1]):
                        next_node = self.merge_children(current_node,
index - 1)
                    else:
                        next_node = self.move_key_right(current_node,
index - 1)
```

```
            current_node = next_node

        if not len(self.root) and len(self.root.child[0]) != 0:
            self.root = self.root.child[0]

    def merge_children(self, node: BTreeNode, index: int):
        left = node.child[index]
        right = node.child[index + 1]
        new_node = BTreeNode(left.leaf)
        new_node.keys = left.keys + [node.keys[index]] + right.keys
        new_node.child = left.child + right.child
        node.keys.pop(index)
        node.child.pop(index)
        node.child[index] = new_node
        return new_node

    def move_key_left(self, node: BTreeNode, index: int) -> BTreeNode:
        left = node.child[index]
        right = node.child[index + 1]
        left.keys.append(node.keys[index])
        left.child.append(right.child[0])
        node.keys[index] = right.keys[0]
        right.keys.pop(0)
        right.child.pop(0)
        return left

    def move_key_right(self, node: BTreeNode, index: int) -> BTreeNode:
        left = node.child[index]
        right = node.child[index + 1]
        right.keys.insert(0, node.keys[index])
        right.child.insert(0, left.child[-1])
        node.keys[index] = left.keys[-1]
        left.keys.pop()
        left.child.pop()
        return right
```

Название файла: HashTable.py

```
class HashTable:
    def __init__(self, size=128) -> None:
        self.size = size
```

```
        self.table = [[], ] * self.size
        self.element = 0

    def first_hash_func(self, key):
        return key * 19 % self.size

    def second_hash_func(self, key):
        h = key * 19 % self.size
        if h % 2:
            return h
        return (h - 1 + self.size) % self.size

    def find(self, key):
        first_hash = self.first_hash_func(key)
        if len(self.table[first_hash]) != 0 and key ==
self.table[first_hash][0]:
            return self.table[first_hash][1]
        second_hash = self.second_hash_func(key)
        elem_hash = (first_hash + second_hash) % self.size
        i = 1
        while len(self.table[elem_hash]) != 0 and i != self.size:
            if self.table[elem_hash][0] == key:
                return self.table[elem_hash][1]
            i += 1
            elem_hash = (first_hash + i * second_hash) % self.size
        return None

    def _find(self, key):
        first_hash = self.first_hash_func(key)
        if len(self.table[first_hash]) != 0 and key ==
self.table[first_hash][0]:
            return first_hash
        second_hash = self.second_hash_func(key)
        elem_hash = (first_hash + second_hash) % self.size
        i = 1
        while i != self.size:
            if len(self.table[elem_hash]) != 0 and
self.table[elem_hash][0] == key:
                return elem_hash
            i += 1
            elem_hash = (first_hash + i * second_hash) % self.size
        return None

    def _insert(self, key, data):
        first_hash = self.first_hash_func(key)
        if len(self.table[first_hash]) == 0:
            self.table[first_hash] = data
            return
        second_hash = self.second_hash_func(key)
        elem_hash = (first_hash + second_hash) % self.size
        i = 1
        while len(self.table[elem_hash]) != 0:
            i += 1
            elem_hash = (first_hash + i * second_hash) % self.size
        if len(self.table[elem_hash]) == 0:
            self.table[elem_hash] = data
            return
```

```
    def insert(self, key, data):
        first_hash = self.first_hash_func(key)
        if len(self.table[first_hash]) == 0:
            self.table[first_hash] = data
            self.element += 1
            self.resize()
            return
        second_hash = self.second_hash_func(key)
        elem_hash = (first_hash + second_hash) % self.size
        i = 1
        while len(self.table[elem_hash]) != 0 and i != self.size:
            i += 1
            elem_hash = (first_hash + i * second_hash) % self.size
        if len(self.table[elem_hash]) == 0:
            self.table[elem_hash] = data
            self.element += 1
            self.resize()
            return
        print('No place to insert')

    def delete(self, key):
        index = self._find(key)
        if index != None:
            self.table[index] = []
            self.element -= 1
            self.resize(True)
            return
        print('No such element')

    def resize(self, delete=False):
        alpha = self.element / self.size * 100
        if alpha > 70:
            info = self.table
            self.size *= 2
            self.table = [[], ] * self.size
            for elem in info:
                if len(elem) != 0:
                    self._insert(elem[0], elem)
        elif alpha < 30 and delete and self.size > 128:
            info = self.table
            self.size //= 2
            self.table = [[], ] * self.size
            for elem in info:
                if len(elem) != 0:
                    self._insert(elem[0], elem)
```

Название файла: main.py

```
from random import randint
import matplotlib.pyplot as plt
import hash_table
import btree
import time
```

```
import math

tree = btree.BTree(50)
hash = hash_table.HashTable(128)

x = [i for i in range(1_000_000)]
total = []
total_del = []
ins=[]
fnd=[]
dlt=[]

for i in range(1000):
    total.append(i)
    start = time.time_ns()
    # hash.insert(i, [i, 's'])
    for j in x[i * 1000: (i + 1) * 1000]:
        tree.insert([j, j*17])
    end = time.time_ns()
    ins.append(end - start)


for i in range(1000):
    start = time.time_ns()
    # hash.find(i)
    for j in x[i * 1000: (i + 1) * 1000]:
        tree.find(j)
    end = time.time_ns()
    fnd.append(end - start)


for i in range(1000):
    total_del.append(1000 - i)
    start = time.time_ns()
    # hash.delete(i)
    for j in x[i * 1000: (i + 1) * 1000]:
        tree.delete(tree.root, j)
    end = time.time_ns()
```

```
    dlt.append(end - start)
```

```
plt.title('Операция сложности операций в хеш-таблице в лучшем случае')
plt.xlabel("Число элементов")
plt.ylabel("Время, нс")
plt.grid()
plt.scatter(total, ins, s = 5, color = ["red"])    #Вставка элемента
plt.scatter(total, fnd , s = 5, color = ["green"])  #Поиск элемента
plt.scatter(total_del, dlt , s = 5, color = ["blue"])   #Удаление
элемента
plt.legend(['Сложность вставки', 'Сложность поиска', 'Сложность
удаления'])
plt.show()
```