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

КУРСОВАЯ РАБОТА

по дисциплине «Алгоритмы и структуры данных»

Тема: Хэш-Таблицы

Студент гр. 0303 _____ Болкунов В.О.

Преподаватель _____ Берленко Т.А.

Санкт-Петербург
2021

ЗАДАНИЕ
НА КУРСОВУЮ РАБОТУ

Студент Болкунов Владислав Олегович

Группа 0303

Тема работы: Исследование: «Хэш-таблица (двойное хэширование) vs Хэш-таблица (метод цепочек)».

Исходные данные:
Реализовать требуемые структуры данных, сгенерировать входные данные и сравнить количественные характеристики структур данных.

Содержание пояснительной записки:
Разделы: «Аннотация», «Содержание», «Введение», «Реализованные классы», «Тестирование», «Заключение», «Список использованных источников», «Приложение»

Предполагаемый объем пояснительной записки:
Не менее 15 страниц.

Дата выдачи задания: 16.10.2021

Дата сдачи реферата: 08.12.2021

Дата защиты реферата: 10.12.2021

Студент _____ Болкунов В.О.

Преподаватель _____ Берленко Т.А.

АННОТАЦИЯ

Курсовая работа заключается в реализации указанных структур данных - хэш-таблиц с различными алгоритмами разрешения коллизий, а именно «метод цепочек» и «двойное хэширование». Также требуется сравнить работу реализованных структур на различных входных параметрах (их количественные характеристики).

Поставленные задачи были выполнены в ходе работы: структуры реализованы в виде классов, с набором стандартных(и не только) для хэш-таблиц методов. Корректность работы структур была проверена на выдуманных и случайно сгенерированных наборах данных, результаты работы структур сравнивались со стандартным словарём в библиотеке языка `Python`. Отдельно были проведены тесты на оценку количественных характеристик (времени работы) каждой структуры, результаты также сравнивались со временем выполнения той же задачи стандартным словарём в `Python`.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Цели работы:

1. Реализовать структуру данных хэш-таблица с методом разрешения коллизий : «метод цепочек».
2. Реализовать структуру данных хэш-таблица с методом разрешения коллизий: «двойное хэширование» (проба в методе открытой адресации).
3. Реализовать тесты (наборы входных данных) для оценки сравнения количественных характеристик реализованных структур данных.

Теоретические сведения:

Хэш-функция — некоторая функция преобразующая исходный объект (различных типов: числа, строки, структуры и тд.) в битовую последовательность - числа (чаще всего определённого размера — например 32 или 64 бит).

Требования к хэш-функции:

1. Достаточно быстрая скорость вычисления.
2. Детерминирована значением входного объекта.
3. Хэш-функция для двух одинаковых объектов должна быть равна.

* для двух разных объектов она не обязана отличатся.

Хэш-таблиица — это структура данных, реализующая интерфейс ассоциативного массива, а именно, она позволяет хранить пары (ключ, значение) и выполнять три операции: операцию добавления новой пары, операцию поиска и операцию удаления пары по ключу. Особенность хэш-таблиц заключается в использовании хэш-функции для размещения элементов в своём внутреннем массиве. При хорошо подобранных хэш-функциях время выполнения операций (вставки, поиска, удаления) стремится к O(1). В идеальном случае: хэш-функции всех объектов различны по модулю размера таблицы. Добиться такого результата в общем случае почти невозможно, что

приводит к коллизиям — совпадению значения хэш-функции для двух различных объектов.

Для разрешения коллизий существуют несколько методов:

Метод Цепочек — его суть заключается в создании связных списков в качестве элементов хэш-таблицы, при совпадении хэш-функций двух объектов их ключи будут сравниваться с ключами в объектов в списке. В худшем случае метод цепочек приводит к созданию большого связанного списка, что уже не даёт хороший результат для операций в хэш-таблице.

Открытая адресация — принцип его работы схож с методом цепочек, но в качестве элементов таблицы уже берутся просто пары ключ-значение. При совпадении хэш-функции будет осуществляться поиск следующей ячейки таблицы по заданному правилу (проба).

Конкретно **метод двойного хэширования** предполагает заведение дополнительной хэш-функции, которая будет использоваться для взятия проб в таблице. Требования к такой функции: она не должна быть равна 0 по модулю таблицы и все ещё значения должны быть взаимно простыми с числом размера таблицы (этого можно добиться если размер таблицы будет всегда простым числом), и желательно она должна быть как можно менее зависима от первой хэш-функции (что довольно редко возможно).

Открытая адресация в худшем случае всё также приведёт к образованию больших кластеров (цепочек проб), что значительно замедлит скорость операций.

1. РЕАЛИЗОВАННЫЕ КЛАССЫ

1.1. Вспомогательные функции и классы

class Pair — пара ключ-значение (key, value)

defautlHash(hm, key) — стандартная функция для хэширования объекта (использует функцию hash : стандартная функция __hash__ для объектов в python) .

class Node (в ChainedMap.py) — узел связанного списка.

class LinkedList — связный список для использования в методе цепочек, используется фиктивный узел — голова.

def isPrime(n: int) -> bool — функция проверки числа на простоту.

def nextPrime(n: int) -> int — функция находящая следующее за данным. числом простое число.

def defaultHash2(hm: HashMap, key) -> int — стандартная функция для выисления второй хэш-функции в методе двойного хэширования.

class Node (в DHashedMap.py) — узел таблицы в методе двойного хэширования (нужен для проверки того что элемент удалён).

1.2. HashMap

class HashMap — базовый класс хэш-таблиц, содержит «интерфейс» хэш-таблицы, конструктор принимает начальный размер таблицы и хэш-фукнцию (по умолчанию — defautlHash).

Интерфейс хэш-таблицы:

- mapSize(self) -> int — фактический размер массива.
- itemsSize(self) -> int — количество хранимых элементов.
- fillCoef(self) -> float — коэффициент заполнения таблицы.
- calcHash(self, key) -> int — рассчёт хэш-функции.

- insert(self, obj: Pair) -> bool - вставка пары ключ-значение.
- get(self, key) - получение значения по ключу.
- remove(self, key) -> bool - удаление пары по ключу.
- refresh(self) - обновление таблицы (создание новой и копирование в текущую — требуется для расширения таблицы).
- keyset(self) -> list - список хранящихся ключей.
- expand(self, add: int) - расширение таблицы на заданное число ячеек.
- __str__(self) и __repr__(self) — для вывода таблицы в текстовом виде.

1.3. ChainedMap

class ChainedMap — класс хэш-таблицы, реализующий интерфейс HashMap, решает проблему коллизий методом цепочек.

Дополнительные методы:

__find(self, key) -> Node - универсальная функция поиска узла для вставки, удаления и получения. Возвращает узел предыдущей перед найденным, если следующий узел не None — узел считается найденным. В случае если узел не найден — следующий узел (None) будет узлом куда возможно вставить элемент. Данный подход позволяет минимизировать методы вставки, поиска и удаления, однако получается что новые элементы будут вставляться не в голову списков а в их конец (что в общем случае не влияет на корректность и скорость работы).

getListsFill(self) -> List[int] - возвращает список с количеством элементов в списках таблицы.

Дополнительные константы:

EXPAND_COEF = 0.5 - определяет во сколько раз увеличится массив при «максимальном» заполнении.

MAX_FILL_COEFF = 1.5 - определяет «максимальное» заполнение — максимальное значение коэффициента заполнения для расширения таблицы.

1.4. DHashedMap

class DHashedMap — класс хэш-таблицы, реализующий интерфейс HashMap, решает проблему коллизий методом двойного хэширования (открытая адресация). Дополнительно принимает в конструктор вторую хэш-функцию.

Дополнительные методы:

__probe(self, h1: int, h2: int, i: int) -> int - функция для взятия i-той пробы.

__find(self, key) -> int: - универсальная функция для поиска, удаления и вставки элемента. Возвращает индекс элемента, если тот найден, иначе возвращает место куда этот элемент возможно вставить (для кластеров с удалёнными элементами возвращает последний найденный помеченный `deleted` элемент).

Дополнительные константы:

EXPAND_COEF = 0.5 - определяет во сколько раз увеличится массив при «максимальном» заполнении.

MAX_FILL_COEFF = 0.7 - определяет «максимальное» заполнение — максимальное значение коэффициента заполнения для расширения таблицы. Данное значение было выбрано из следующего соображения: при максимальном (или близкому к максимальному) заполнению таблицы, функции поиска будут в общем случае перебирать почти всю таблицу.

2. ИССЛЕДОВАНИЕ

2.1. Вспомогательные функции для тестов

str_hash(s: str, k: int = K, p: int = P) -> int - функция для хэширование строки полиномиальным способом.

assertion(cm: ChainedMap, dm: DHashedMap, pm: dict, printing=False) - функция для проверки (с помощью assert) переданных таблиц на равенство: то есть равенство размеров (количества элементов), равенство списков ключей таблиц и непосредственно равенство элементов по каждому ключу.

compare_insertion(cm: ChainedMap, dm: DHashedMap, pm: dict, data: List[Pair], printing: bool = True): - функция для сравнения вставки списка пар `data` в 3 переданные хэш-таблицы: таблица с методом цепочек, с методом двойного хэширования и стандартный словарь в языке `Python`. Осуществляет проверку с помощью функции assertion.

compare_deleting(cm: ChainedMap, dm: DHashedMap, pm: dict, keys: list, printing: bool = True): - функция для удаления из 3ёх переданных таблиц элементов по списку ключей `keys`, производит проверку с помощью функции assertion.

2.2. Тестирование корректности

В этом разделе (*testing/accuracy_tests*) проводится качественное исследование реализованных хэш-таблиц и их сравнение со стандартным словарём dict. Все тесты можно запустить (без вывода промежуточной информации) скриптом `*all.py*`, по итогу все тесты были успешно пройдены. Вставка и удаление данных в таблицы осуществляется с помощью описанных выше вспомогательных функций.

2.2.1. Числа (*numbers.py*)

В данных тестах сравнивается вставка (с коллизиями и без) и удаление подготовленных наборов данных. В качестве ключей хэш-таблиц выступают числа.

2.2.2. Строки (*strings.py*)

В данных тестах сравнивается вставка и удаление подготовленных данных со строками в качестве ключей, также один тест с генерацией случайных строк заданной длины.

2.2.3. Случайный большой набор чисел (random_tests.py)

В данных тестах сравнивается вставка и удаление случайных наборов данных с числами в качестве ключей. Размеры массивов входных значений соответственно 50 и 100_000.

2.2. Тестирование времени работы

В этом разделе (*testing/time_tests*) производится количественная оценка и сравнение характеристик реализованных структур данных (время работы). Также для сравнения те же тесты запускаются на стандартном словаре dict. Время измерялось в микросекундах (мкс.).

2.2.1. Вставка (*insertion.py*)

В данном тесте проверяется время вставки целого массива случайных даннх в хэш-таблицы с заданными параметрами: `N` - количество тестов, `S` - начальные размеры таблиц. При проведении теста считается среднее время за количество экспериментов равных указанному параметру (во всех проведённых тестах число экспериментов было равно 100). Результаты представлены в следующих таблицах.

ChainedMap:

Таблица 1: Результаты вставки в таблицу с методом цепочек

t, мкс.	S = 10	S = 100	S = 1000	S = 10_000
N = 10	35	36	39	60
N = 100	813	190	201	230
N = 1000	8842	9456	2303	2557
N = 10_000	136339	129522	109494	31968

DHashedMap:

Таблица 2: Результаты вставки в таблицу с двойным хэшированием

t, мкс.	S = 10	S = 100	S = 1000	S = 10_000
N = 10	94	44	53	55
N = 100	956	487	236	244
N = 1000	10489	10797	5089	2657
N = 10_000	111838	106055	116472	65042

dict:

Таблица 3: Результаты вставки в стандартный словарь

N	t, мкс.
10	1
100	10
1000	109
10_000	1630

2.2.2. Поиск (*searching.py*)

В данном тесте производится оценка времени поиска элементов в таблицах. Поиск производится по всему существующему набору ключей.

Таблица 4: Результаты времени поиска в различных таблицах

t, мкс.	N = 100	N = 1000	N = 10_000	N = 100_000
ChainedMap	1.0	1	1.5	1.6
DHashedMap	2.5	2.8	3	3
dict	0.1	0.2	0.2	0.2

2.2.3. Удаление (*deleting.py*)

В данном тесте производится оценка времени удаления всех элементов из таблиц.

Таблица 5: Результаты времени удаления

t, мкс.	N = 10	N = 100	N = 1000	N = 10_000
ChainedMap	1085	1059	1551	1691
DHashedMap*	2569	2819	2982	2978
dict	115	170	130	216

*для таблицы с методом двойного хэширование реальный размер таблицы был больше (выбран таким образом чтобы не приходилось делать дополнительного расширения).

2.2.4. Тестирование на реальной задаче

В данном тесте (`*counting_task.py*`) хэш-таблицы были протестированы на задаче поиска количества уникальных чисел в сгенерированной последовательности. Сами числа расположены в диапазоне от 0 до 100. Начальный размер таблиц = 10.

Таблица 6: Результаты решения задачи

t, мкс.	N = 10	N = 100	N = 1000	N = 10_000
ChainedMap	38	1687	29091	299401
DHashedMap*	56	1194	15016	152750
dict	1	16	138	1576

В усложнённом тесте (`*counting_task_improved.py*`), после вставки удаляются случайно выбранные n ключей (n равно половине размера исходного массива данных), затем вставляется новая случайная последовательность чисел. Результаты представлены в (табл. 7)

Таблица 7: Результаты решения усложнённой задачи

t, мкс.	N = 10	N = 100	N = 1000	N = 10_000
ChainedMap	141	4710	56611	592642
DHashedMap*	213	3148	30265	311453
dict	6	42	269	2553

2.2.5. Выводы по произведённым тестам

По результатам оценки времени работы различных тестов можно сделать вывод что сложность основных операций примерно соответствует теоретическим ожиданиям. Интересно заметить что хэш-таблица на основе метода цепочек в простых тестах (на вставку, удаление и поиск) работает быстрее (примерно в 2 раза), однако при решении задачи ситуация обратная: таблица на основе двойного хэширования справляется с задачей почти в 2 раза быстрее. Возможно хэш-таблица с методом цепочек хуже справляется с постоянным операция обновления своих пар.

Затраты по пямяти у двух методов не отличаются так как создаётся одинаковое количество узлов (пар) в каждом методе.

ЗАКЛЮЧЕНИЕ

В ходе работы были реализованы требуемые структуры данных: хэш-таблицы с методом цепочек и с двойным хэшированием, проверена корректность их работы в сравнении со стандартными средствами языка, и проведены тесты по оценке времени работы, результаты работы сравнены для двух таблиц между собой и со стандартными средствами языка.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

https://ru.wikipedia.org/wiki/Хеш-таблица

https://neerc.ifmo.ru/wiki/index.php?title=Хеш-таблица

ПРИЛОЖЕНИЕ А

КОД ПРОГРАММЫ

Файл `hash_maps/HashMap.py`

```
class Pair:
    def __init__(self, key, value):
        self.key = key
        self.value = value

    def __str__(self) -> str:
        return f'( {self.key} : {self.value} )'

    def __repr__(self):
        return str(self)


def defautlHash(hm, key):
    return hash(key)


class HashMap:
    def __init__(self, size: int, hashFunc=defautlHash):
        self.hashFunc = hashFunc
        self.arr = [None] * size
        self._itemsSize: int = 0

    def mapSize(self) -> int:
        return len(self.arr)

    def itemsSize(self) -> int:
        return self._itemsSize

    def fillCoef(self) -> float:
        return self.itemsSize() / self.mapSize()

    def calcHash(self, key) -> int:
        pass

    def insert(self, obj: Pair) -> bool:
        pass

    def get(self, key):
        pass

    def remove(self, key) -> bool:
        pass

    def refresh(self):
        pass

    def keyset(self) -> list:
        pass

    def expand(self, add: int):
        self.arr.extend([None] * add)
```

```
    def __str__(self) -> str:
        return '{\n' + '\n'.join([str(i) for i in self.arr]) + '\n}'

    def __repr__(self):
        return str(self)
```

Файл `hash_maps/ChainedMap.py`

```
from typing import List, Callable

from .HashMap import Pair, HashMap, defautlHash


class Node:
    def __init__(self, item, nxt=None):
        self.item = item
        self.nxt = nxt


class LinkedList:
    def __init__(self):
        self.head = Node(None, None)

    def spread(self) -> List[Pair]:
        node = self.head.nxt
        lst = []
        while node is not None:
            lst.append(node.item)
            node = node.nxt
        return lst

    def __str__(self):
        return '[' + ', '.join(list(map(str, self.spread()))) + ']'

    def __repr__(self):
        return str(self)


class ChainedMap(HashMap):
    EXPAND_COEF = 0.5
    MAX_FILL_COEFF = 1.5

    def __init__(self, size: int, hashFunc: Callable[[HashMap, object], int] = defautlHash):
        super().__init__(size, hashFunc)

    def calcHash(self, key) -> int:
        return self.hashFunc(self, key) % self.mapSize()

    def __find(self, key) -> Node:
        i = self.calcHash(key)
        if self.arr[i] is None:
            self.arr[i] = LinkedList()
        node = self.arr[i].head
```

```
        while node.nxt is not None:
            if node.nxt.item.key == key:
                return node
            node = node.nxt
        return node

    def insert(self, obj: Pair) -> bool:
        if self.fillCoef() > ChainedMap.MAX_FILL_COEFF:
            self.expand(int(self.mapSize() * ChainedMap.EXPAND_COEF))
        node = self.__find(obj.key)
        if node.nxt is not None:
            node.nxt.item.value = obj.value
            return False
        else:
            self._itemsSize += 1
            node.nxt = Node(obj, None)
            return True

    def get(self, key):
        node = self.__find(key)
        if node.nxt is not None:
            return node.nxt.item.value
        else:
            return None

    def remove(self, key) -> bool:
        node = self.__find(key)
        if node.nxt is not None:
            node.nxt = node.nxt.nxt
            self._itemsSize -= 1
            return True
        else:
            return False

    def keyset(self) -> list:
        return [item.key for items in self.arr if items is not None for item in items.spread()]

    def expand(self, add: int):
        self.arr.extend([LinkedList() for i in range(add)])
        self.refresh()

    def refresh(self):
        newMap = ChainedMap(self.mapSize(), self.hashFunc)
        for i in self.arr:
            if i is not None:
                for j in i.spread():
                    newMap.insert(j)
        self.arr = newMap.arr

    def getListsFill(self) -> List[int]:
        return [len(i.spread()) if i is not None else 0 for i in self.arr]
```

Файл `hash_maps/DhashedMap.py`

```
from typing import Callable, Hashable, Tuple

from .HashMap import Pair, HashMap, defautlHash


def isPrime(n: int) -> bool:
    if n % 2 == 0:
        return False
    for i in range(3, int(n ** (1 / 2)) + 1, 2):
        if n % i == 0:
            return False
    return True


def nextPrime(n: int) -> int:
    while not isPrime(n):
        n += 1
    return n


def defaultHash2(hm: HashMap, key) -> int:
    return 1 + hm.hashFunc(hm, key) % (hm.mapSize() - 1)


class Node:
    def __init__(self, item):
        self.item = item
        self.deleted = False

    def delete(self):
        self.deleted = True

    def __str__(self):
        return str(self.item) if not self.deleted else 'deleted'

    def __repr__(self):
        return str(self)


class DHashedMap(HashMap):
    EXPAND_COEF = 0.5
    MAX_FILL_COEFF = 0.7

    def __init__(self, size: int, hashFunc: Callable[[HashMap, object], int] = defautlHash,
                 hashFunc2: Callable[[HashMap, object], int] = defaultHash2):
        super().__init__(nextPrime(size), hashFunc)
        self.hashFunc2 = hashFunc2

    def __probe(self, h1: int, h2: int, i: int) -> int:
        return (h1 + i * h2) % self.mapSize()

    def calcHash(self, key) -> Tuple[int, int]:
        h2 = self.hashFunc2(self, key)
        if h2 == 0:
            raise ArithmeticError('incorrect second hash function')
        return self.hashFunc(self, key), h2,
```

```
def __find(self, key) -> int:
    h1, h2 = self.calcHash(key)
    j = 0
    i = self.__probe(h1, h2, j)
    free = None
    while j < self.mapSize() and self.arr[i] is not None:
        if self.arr[i].item.key == key:
            return i
        elif self.arr[i].deleted:
            free = i
        j += 1
        i = self.__probe(h1, h2, j)

    return i if free is None else free

def insert(self, obj: Pair) -> bool:
    if self.fillCoef() >= DHashedMap.MAX_FILL_COEFF:
        self.expand(int(self.mapSize() * DHashedMap.EXPAND_COEF))
    i = self.__find(obj.key)
    res = self.arr[i] is not None and not self.arr[i].deleted
    self.arr[i] = Node(obj)
    if not res:
        self._itemsSize += 1
    return not res

def get(self, key):
    i = self.__find(key)
    if self.arr[i] is not None and not self.arr[i].deleted:
        return self.arr[i].item.value
    else:
        return None

def keyset(self) -> list:
    return [i.item.key for i in self.arr if i is not None and not i.deleted]

def remove(self, key) -> bool:
    i = self.__find(key)
    if self.arr[i] is not None and not self.arr[i].deleted:
        self.arr[i].delete()
        self._itemsSize -= 1
        return True
    else:
        return False

def expand(self, add: int):
    super().expand(nextPrime(add))
    self.refresh()

def refresh(self):
    newMap = DHashedMap(self.mapSize(), self.hashFunc, self.hashFunc2)
    for i in self.arr:
        if i is not None and not i.deleted:
            newMap.insert(i.item)
    self.arr = newMap.arr
```

Файл `testing/str_hash.py`

```
from hash_maps import nextPrime

K = 37
P = nextPrime(2 ** 32)


def str_hash(s: str, k: int = K, p: int = P) -> int:
    h = 0
    for i in s:
        h = (h * k + ord(i)) % p
    return h
```

Файл `testing/maps_comparator.py`

```
from typing import List, Tuple

from hash_maps import ChainedMap, DHashedMap, Pair


def assertion(cm: ChainedMap, dm: DHashedMap, pm: dict, printing=False):
    assert cm.itemsSize() == dm.itemsSize() == len(pm)
    assert sorted(cm.keyset()) == sorted(dm.keyset()) == sorted(pm.keys())
    for i in list(pm.keys()):
        assert cm.get(i) == dm.get(i) == pm[i]
        if printing:
            print(f"key: {i}, values: {cm.get(i)} | {dm.get(i)} | {pm[i]}")


def compare_insertion(cm: ChainedMap, dm: DHashedMap, pm: dict, data: List[Pair], printing:
bool = True):
    for i in data:
        cm.insert(i)
        dm.insert(i)
        pm[i.key] = i.value

    if printing:
        print('Input list: [', *data, ']', sep='\n')
        print('_' * 50)
        print('Chained Map:', cm, f'fill coefficient = {cm.fillCoef()}')
        print('_' * 50)
        print('Double Hashed Map:', dm, f'fill coefficient = {dm.fillCoef()}')
        print('_' * 50)
        print("Python standard dict: {", *pm.items(), '}', sep='\n')
        print('_' * 50)
        print('Comparison:')

    assertion(cm, dm, pm, printing)
    print('insertion finished successfully')
    print('#' * 50)


def compare_deleting(cm: ChainedMap, dm: DHashedMap, pm: dict, keys: list, printing: bool =
True):
```

```
    for i in keys:
        cm.remove(i)
        dm.remove(i)
        del pm[i]

    if printing:
        print('Input keys for deleting:[', *keys, ']', sep='\n')
        print('_' * 50)
        print('Chained Map:', cm, f"fill coefficient = {cm.fillCoef()}")
        print('_' * 50)
        print("Double Hashed Map:", dm, f'fill coefficient = {dm.fillCoef()}')
        print('_' * 50)
        print('Python standard Dict: {', *pm.items(), '}', sep='\n')
        print('_' * 50)
        print('Comparison:')

    assertion(cm, dm, pm, printing)
    print('deleting finished successfully')
    print('#' * 50)
```

Файл `testing/accuracy_tests/numbers.py`

```
from hash_maps import Pair, DHashedMap, ChainedMap
from testing.maps_comparator import compare_insertion, compare_deleting


def nonCollisionTest(printing=True):
    N = 6
    data = [
        Pair(0, 0),
        Pair(8, None),
        Pair(9, 9),
        Pair(13, 'str'),
        Pair(16, 16),
        Pair(23, [1, 2, 3])
    ]
    cm = ChainedMap(N)
    dm = DHashedMap(N)
    pm = dict()
    compare_insertion(cm, dm, pm, data, printing)


def collisionTest(printing=True):
    N = 5
    data = [
        Pair(0, 0),
        Pair(8, None),
        Pair(6, 9),
        Pair(12, 'str'),
        Pair(1, 16),
        Pair(7, [1, 2, 3]),
        Pair(18, 18),
        Pair(14, None),
        Pair(20, 3.14),
        Pair(12, 'new_str')
    ]
    cm = ChainedMap(N)
```

```
    dm = DHashedMap(N)
    pm = dict()
    compare_insertion(cm, dm, pm, data, printing)


def deletingTest1(printing=True):
    N = 6
    data = [
        Pair(0, 0),
        Pair(8, None),
        Pair(9, 9),
        Pair(13, 'str'),
        Pair(16, 16),
        Pair(23, [1, 2, 3])
    ]
    cm = ChainedMap(N)
    dm = DHashedMap(N)
    pm = dict()
    compare_insertion(cm, dm, pm, data, False)
    compare_deleting(cm, dm, pm, [8, 13, 23], printing)


def deletingTest2(printing=True):
    N = 5
    data = [
        Pair(0, 0),
        Pair(8, None),
        Pair(6, 9),
        Pair(12, 'str'),
        Pair(1, 16),
        Pair(7, [1, 2, 3]),
        Pair(18, 18),
        Pair(14, None),
        Pair(20, 3.14),
        Pair(12, 'new_str')
    ]
    cm = ChainedMap(N)
    dm = DHashedMap(N)
    pm = dict()
    compare_insertion(cm, dm, pm, data, printing=False)
    compare_deleting(cm, dm, pm, [8, 12, 0, 14, 20], printing)


if __name__ == "__main__":
    print('Test with collisions:')
    nonCollisionTest()

    print('Test without collisions:')
    collisionTest()

    print('Deleting test 1:')
    deletingTest1()

    print('Deleting test 2:')
    deletingTest2()
```

Файл `testing/accuracy_tests/strings.py`

```
from hash_maps import Pair, DHashedMap, ChainedMap
from testing.maps_comparator import compare_insertion, compare_deleting
from testing.str_hash import str_hash
import string
import random


def strTest1(printing=True):
    N = 6
    data = [
        Pair('hello', 0),
        Pair('world', None),
        Pair('string', 9),
        Pair('hash', 'str'),
        Pair('code', 16),
        Pair('oaoa', [1, 2, 3])
    ]
    cm = ChainedMap(N, lambda hm, s: str_hash(s))
    dm = DHashedMap(N, lambda hm, s: str_hash(s))
    pm = dict()
    compare_insertion(cm, dm, pm, data, printing)
    compare_deleting(cm, dm, pm, ['world', 'hash', 'hello'], printing)


def strTest2(printing=False):
    WORD_LEN = 10
    START = 10
    N = 1000
    data = [
        Pair(''.join(random.choices(string.ascii_letters, k=WORD_LEN)), i) for i in range(N)
    ]
    cm = ChainedMap(START, lambda hm, s: str_hash(s))
    dm = DHashedMap(START, lambda hm, s: str_hash(s))
    pm = dict()
    compare_insertion(cm, dm, pm, data, printing)
    DELETE = 150
    compare_deleting(cm, dm, pm, list(set(random.choices(list(pm.keys()), k=DELETE))),
printing)


if __name__ == "__main__":
    print('Strings as keys test 1:')
    strTest1()

    print('Strings as keys (random test):')
    strTest2()
```

Файл `testing/accuracy_tests/random_test.py`

```
from hash_maps import Pair, DHashedMap, ChainedMap
from testing.maps_comparator import compare_insertion, compare_deleting
import random


def rand1(printing=True):
    START = 10
    N = 50
    data = [Pair(random.randint(0, 1_000), i) for i in range(N)]
    cm = ChainedMap(START)
    dm = DHashedMap(START)
    pm = dict()
    compare_insertion(cm, dm, pm, data, printing)
    DELETE = 25
    compare_deleting(cm, dm, pm, list(set(random.choices(list(pm.keys()), k=DELETE))),
printing)


def rand2(printing=False):
    START = 100
    N = 100_000
    data = [Pair(random.randint(0, 1_000_000), i) for i in range(N)]
    cm = ChainedMap(START)
    dm = DHashedMap(START)
    pm = dict()
    compare_insertion(cm, dm, pm, data, printing)
    DELETE = 50_000
    compare_deleting(cm, dm, pm, list(set(random.choices(list(pm.keys()), k=DELETE))),
printing)


if __name__ == "__main__":
    print('Random test 1:')
    rand1()

    print('Random test 2:')
    rand2()
```

Файл `testing/accuracy_tests/all.py`

```
from numbers import *
from random_tests import *
from strings import *

if __name__ == "__main__":
    nonCollisionTest(False)
    collisionTest(False)
    deletingTest1(False)
    deletingTest2(False)
    strTest1(False)
    strTest2(False)
    rand1(False)
    rand2(False)
```

Файл `testing/time_tests/insertion.py`

```
from hash_maps import Pair, DHashedMap, ChainedMap
import time
import random

MAX_KEY = 2 ** 32
PROBES = 100
TESTS = [10, 100, 1_000, 10_000]
START_LEN = [10, 100, 1_000, 10_000]


def insertionCM(n: int, start: int, probes: int):
    dT = []
    for i in range(probes):
        data = [Pair(random.randint(0, MAX_KEY), i) for i in range(n)]
        t = time.time_ns()
        cm = ChainedMap(start)
        for i in data:
            cm.insert(i)
        dt = time.time_ns() - t
        dT.append(dt)
    avg = sum(dT) / len(dT)
    print(f'added {n} items in Chained Map with starting length = {start} '
          f'with average time {avg // 1e3} mcs for {probes} tests, all time = {sum(dT) / 1e9} s')


def insertionDM(n: int, start: int, probes: int):
    dT = []
    for i in range(probes):
        data = [Pair(random.randint(0, MAX_KEY), i) for i in range(n)]
        t = time.time_ns()
        cm = DHashedMap(start)
        for i in data:
            cm.insert(i)
        dt = time.time_ns() - t
        dT.append(dt)
    avg = sum(dT) / len(dT)
    print(
        f'added {n} items in Double Hashed Map with starting length = {start} '
        f'with average time {avg // 1e3} mcs for {probes} tests, all time = {sum(dT) / 1e9} s')


def insertionPD(n: int, probes: int):
    dT = []
    for i in range(probes):
        data = [Pair(random.randint(0, MAX_KEY), i) for i in range(n)]
        t = time.time_ns()
        cm = dict()
        for i in data:
            cm[i.key] = i.value
        dt = time.time_ns() - t
        dT.append(dt)
    avg = sum(dT) / len(dT)
    print(
        f'added {n} items in python dict with average time {avg // 1e3} mcs  for {probes} tests,
all time = {sum(dT) / 1e9} s')
```

```
if __name__ == "__main__":
    for n in TESTS:
        for sz in START_LEN:
            insertionCM(n, sz, PROBES)
            insertionDM(n, sz, PROBES)
            print('_' * 50)
        insertionPD(n, PROBES)
        print('#' * 50)
```

Файл `testing/time_tests/searching.py`

```
from hash_maps import Pair, DHashedMap, ChainedMap
import time
import random

MAX_KEY = 2 ** 32
PROBES = 100
TESTS = [100, 1_000, 10_000, 100_000]


def searchCM(cm: ChainedMap, probes: int):
    keys = cm.keyset()
    dT = []
    for i in range(probes):
        for k in keys:
            t = time.time_ns()
            cm.get(k)
            dt = time.time_ns() - t
            dT.append(dt)
    avg = sum(dT) / len(dT)
    print(f'Found all items in Chained Map with size = {cm.mapSize()} '
          f'with average time {int(avg)} ns for {probes} tests, all time = {sum(dT) / 1e9} s')


def searchDM(dm: DHashedMap, probes: int):
    keys = dm.keyset()
    dT = []
    for i in range(probes):
        for k in keys:
            t = time.time_ns()
            dm.get(k)
            dt = time.time_ns() - t
            dT.append(dt)
    avg = sum(dT) / len(dT)
    print(f'Found all items in Double Hashed Map with size = {dm.mapSize()} '
          f'with average time {int(avg)} ns for {probes} tests, all time = {sum(dT) / 1e9} s')


def searchPD(pm: dict, probes: int):
    keys = pm.keys()
    dT = []
    for i in range(probes):
        for k in keys:
            t = time.time_ns()
            _ = pm[k]
            dt = time.time_ns() - t
            dT.append(dt)
```

```
    avg = sum(dT) / len(dT)
    print(f'Found all items in python dict with size = {len(pm)} '
          f'with average time {int(avg)} ns for {probes} tests, all time = {sum(dT) / 1e9} s')


if __name__ == "__main__":
    for n in TESTS:
        cm = ChainedMap(n)
        dm = DHashedMap(int(n / DHashedMap.MAX_FILL_COEFF) + 1)
        pm = dict()
        data = [Pair(random.randint(0, MAX_KEY), i) for i in range(n)]
        for i in data:
            cm.insert(i)
            dm.insert(i)
            pm[i.key] = i.value
        searchCM(cm, PROBES)
        searchDM(dm, PROBES)
        searchPD(pm, PROBES)
        print('#' * 50)
```

Файл `testing/time_tests/deleting.py`

```
from hash_maps import Pair, DHashedMap, ChainedMap
import time
import random

MAX_KEY = 2 ** 32
PROBES = 100
TESTS = [10, 100, 1_000, 10_000]


def deleteCM(cm: ChainedMap):
    keys = cm.keyset()
    t = time.time_ns()
    for k in keys:
        cm.remove(k)
    return time.time_ns() - t


def deleteDM(dm: DHashedMap):
    keys = dm.keyset()
    t = time.time_ns()
    for k in keys:
        dm.remove(k)
    return time.time_ns() - t


def deletePD(pm: dict):
    keys = list(pm.keys())
    t = time.time_ns()
    for k in keys:
        del pm[k]
    return time.time_ns() - t


if __name__ == "__main__":
```

```
    for n in TESTS:
        cT, dT, pT = [], [], []
        for i in range(PROBES):
            cm = ChainedMap(n)
            dm = DHashedMap(int(n / DHashedMap.MAX_FILL_COEFF) + 1)
            pm = dict()
            data = [Pair(random.randint(0, MAX_KEY), i) for i in range(n)]
            for j in data:
                cm.insert(j)
                dm.insert(j)
                pm[j.key] = j.value
            cT.append(deleteCM(cm))
            dT.append(deleteDM(dm))
            pT.append(deletePD(pm))

        print(f'Deleted all {n} items in Chained Map '
              f'with average time {(sum(cT) / PROBES) // 1e3} mcs for {PROBES} tests, all time =
{sum(cT) / 1e9} s')
        print(f'Deleted all {n} items in Double Hashed Map '
              f'with average time {(sum(dT) / PROBES) // 1e3} mcs for {PROBES} tests, all time =
{sum(dT) / 1e9} s')
        print(f'Deleted all {n} items in python dict '
              f'with average time {(sum(pT) / PROBES) // 1e3} mcs for {PROBES} tests, all time =
{sum(pT) / 1e9} s')
        print('#' * 50)
```

Файл `testing/time_tests/counting_task.py`

```
from hash_maps import Pair, DHashedMap, ChainedMap
from testing.maps_comparator import assertion
import time
import random

MAX_KEY = 100
PROBES = 100
TESTS = [10, 100, 1_000, 10_000]
START_LEN = 10


def solveCM(cm: ChainedMap, data):
    t = time.time_ns()
    for k in data:
        if k not in cm.keyset():
            cm.insert(Pair(k, 1))
        else:
            cm.insert(Pair(k, 1 + cm.get(k)))
    return time.time_ns() - t


def solveDM(dm: DHashedMap, data):
    t = time.time_ns()
    for k in data:
        if k not in dm.keyset():
            dm.insert(Pair(k, 1))
        else:
            dm.insert(Pair(k, 1 + dm.get(k)))
    return time.time_ns() - t
```

```
def solvePD(pm: dict, data):
    t = time.time_ns()
    for k in data:
        if k not in pm.keys():
            pm[k] = 1
        else:
            pm[k] = 1 + pm[k]
    return time.time_ns() - t


if __name__ == "__main__":
    for n in TESTS:
        cT, dT, pT = [], [], []
        for i in range(PROBES):
            cm = ChainedMap(START_LEN)
            dm = DHashedMap(START_LEN)
            pm = dict()
            data = [random.randint(0, MAX_KEY) for _ in range(n)]

            cT.append(solveCM(cm, data))
            dT.append(solveDM(dm, data))
            pT.append(solvePD(pm, data))
            assertion(cm, dm, pm)

        print(f'Start length = {START_LEN}, N = {n}:')
        print(f'Task solved by Chained Map '
              f'with average time {(sum(cT) / PROBES) // 1e3} mcs for {PROBES} tests, all time =
{sum(cT) / 1e9} s')
        print(f'Task solved by Double Hashed Map '
              f'with average time {(sum(dT) / PROBES) // 1e3} mcs for {PROBES} tests, all time =
{sum(dT) / 1e9} s')
        print(f'Task solved by python dict '
              f'with average time {(sum(pT) / PROBES) // 1e3} mcs for {PROBES} tests, all time =
{sum(pT) / 1e9} s')
        print('#' * 50)
```

Файл `testing/time_tests/counting_task_improved.py`

```
from hash_maps import Pair, DHashedMap, ChainedMap
from testing.maps_comparator import assertion
import time
import random

MAX_KEY = 100
PROBES = 100
TESTS = [10, 100, 1_000, 10_000]
START_LEN = 10


def insertCM(cm: ChainedMap, data):
    t = time.time_ns()
    for k in data:
        if k not in cm.keyset():
            cm.insert(Pair(k, 1))
        else:
```

```
            cm.insert(Pair(k, 1 + cm.get(k)))
    return time.time_ns() - t


def insertDM(dm: DHashedMap, data):
    t = time.time_ns()
    for k in data:
        if k not in dm.keyset():
            dm.insert(Pair(k, 1))
        else:
            dm.insert(Pair(k, 1 + dm.get(k)))
    return time.time_ns() - t


def insertPD(pm: dict, data):
    t = time.time_ns()
    for k in data:
        if k not in pm.keys():
            pm[k] = 1
        else:
            pm[k] = 1 + pm[k]
    return time.time_ns() - t


def deleteCM(cm: ChainedMap, data):
    t = time.time_ns()
    for k in data:
        cm.remove(k)
    return time.time_ns() - t


def deleteDM(dm: DHashedMap, data):
    t = time.time_ns()
    for k in data:
        dm.remove(k)
    return time.time_ns() - t


def deletePD(pm: dict, data):
    t = time.time_ns()
    for k in data:
        del pm[k]
    return time.time_ns() - t


if __name__ == "__main__":
    for n in TESTS:
        cT, dT, pT = [], [], []
        for i in range(PROBES):
            cm = ChainedMap(START_LEN)
            dm = DHashedMap(START_LEN)
            pm = dict()

            data = [random.randint(0, MAX_KEY) for _ in range(n)]
            ct = insertCM(cm, data)
            dt = insertDM(dm, data)
            pt = insertPD(pm, data)

            assertion(cm, dm, pm)
```

```
            DEL = n // 2
            toDelete = list(set(random.choices(list(pm.keys()), k=DEL)))

            ct += deleteCM(cm, toDelete)
            dt += deleteDM(dm, toDelete)
            pt += deletePD(pm, toDelete)

            assertion(cm, dm, pm)

            newData = [random.randint(0, MAX_KEY) for _ in range(n)]
            ct += insertCM(cm, newData)
            dt += insertDM(dm, newData)
            pt += insertPD(pm, newData)

            assertion(cm, dm, pm)

            cT.append(ct)
            dT.append(dt)
            pT.append(pt)

        print(f'Start length = {START_LEN}, N = {n}:')
        print(f'Task solved by Chained Map '
              f'with average time {(sum(cT) / PROBES) // 1e3} mcs for {PROBES} tests, all time =
{sum(cT) / 1e9} s')
        print(f'Task solved by Double Hashed Map '
              f'with average time {(sum(dT) / PROBES) // 1e3} mcs for {PROBES} tests, all time =
{sum(dT) / 1e9} s')
        print(f'Task solved by python dict '
              f'with average time {(sum(pT) / PROBES) // 1e3} mcs for {PROBES} tests, all time =
{sum(pT) / 1e9} s')
        print('#' * 50)
```