САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ

ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ

ФАКУЛЬТЕТ ИНФОКОММУНИКАЦИОННЫХ ТЕХНОЛОГИЙ

Отчет по лабораторной работе №6

по курсу «Алгоритмы и структуры данных»

Тема: Хеширование. Хеш-таблицы.

Вариант 21

Выполнил:

Ступичев М. Н.

К3139

Проверил:

…

Санкт-Петербург

2024 г.

# Содержание отчета

[Содержание отчета 2](#_Toc185333299)

[Задачи по варианту 3](#_Toc185333300)

[Задача №1. Множество 3](#_Toc185333301)

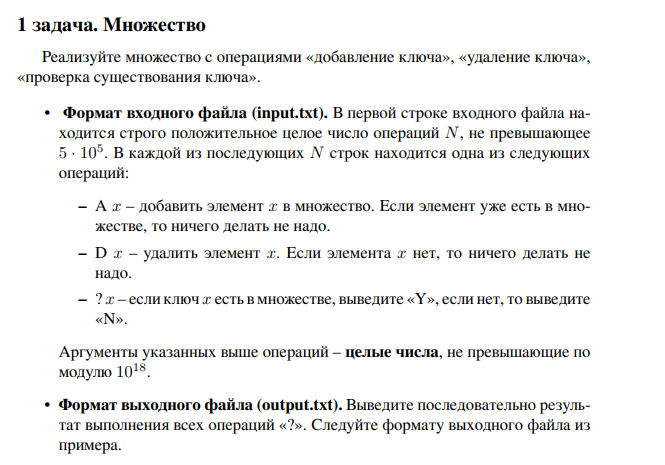
[Задача №2. Телефонная книга. 6](#_Toc185333302)

[Задача №7. Драгоценные камни. 9](#_Toc185333303)

[Вывод 10](#_Toc185333304)

# Задачи по варианту

## Задача №1. Множество



Решение:

class Set:

   def \_\_init\_\_(self, capacity: int = 10 \*\* 6, hash\_func: Callable = hash):

       self.capacity = capacity

       self.hash\_func: Callable = hash\_func

       self.buckets: List[List[int]] = [[] for \_ in range(self.capacity)]

​

   def get\_bucket(self, value: int) -> List[int]:

       bucket\_ind = self.hash\_func(value) % self.capacity

       return self.buckets[bucket\_ind]

​

   def add(self, value: int) -> None:

       bucket = self.get\_bucket(value)

​

       if value not in bucket:

           bucket.append(value)

​

   def remove(self, value: int) -> None:

       bucket = self.get\_bucket(value)

​

       if value in bucket:

           bucket.remove(value)

       else:

           raise ValueError(f"Number {value} not found.")

​

   def \_\_contains\_\_(self, value: int) -> bool:

       return value in self.get\_bucket(value)

​

   def \_\_iter\_\_(self):

       for bucket in self.buckets:

           for value in bucket:

               yield value

Текстовое объяснение решения:

Класс Set реализует простую хеш-таблицу, которая имитирует поведение множества (set). Он позволяет добавлять элементы, удалять их, проверять наличие элемента в множестве, а также итерироваться по множеству.

Разберем класс по частям:

**1. \_\_init\_\_(self, capacity: int = 10 \*\* 6, hash\_func: Callable = hash):**

* Конструктор класса. Принимает необязательные аргументы:
  + capacity: Вместимость хеш-таблицы (количество бакетов/списков). По умолчанию 10\*\*6.
  + hash\_func: Хеш-функция для вычисления хеш-значения элемента. По умолчанию использует встроенную функцию hash.
* Инициализирует:
  + self.capacity: Сохраняет вместимость.
  + self.hash\_func: Сохраняет хеш-функцию.
  + self.buckets: Создает список бакетов (списков), каждый из которых изначально пуст.

**2. get\_bucket(self, value: int) -> List[int]:**

* Вычисляет индекс бакета для заданного элемента, используя хеш-функцию и операцию взятия по модулю: self.hash\_func(value) % self.capacity.
* Возвращает бакет, соответствующий вычисленному индексу.

**3. add(self, value: int) -> None:**

* Добавляет элемент value в множество.
* Получает бакет для элемента.
* Проверяет, есть ли уже такой элемент в бакете.
* Если элемента нет, добавляет его в конец бакета.

**4. remove(self, value: int) -> None:**

* Удаляет элемент value из множества.
* Получает бакет для элемента.
* Проверяет, есть ли элемент в бакете.
* Если элемент есть, удаляет его из бакета.
* Если элемента нет, выбрасывает исключение ValueError.

**5. \_\_contains\_\_(self, value: int) -> bool:**

* Реализует оператор in для проверки наличия элемента в множестве.
* Получает бакет для элемента.
* Возвращает True, если элемент есть в бакете, в противном случае False.

**6. \_\_iter\_\_(self):**

* Реализует возможность итерации по множеству (перебирает все элементы).
* Итерирует по всем бакетам.
* Итерирует по элементам в каждом бакете.
* Использует yield для генерации элементов (ленивая итерация).

**Асимптотическая сложность:**

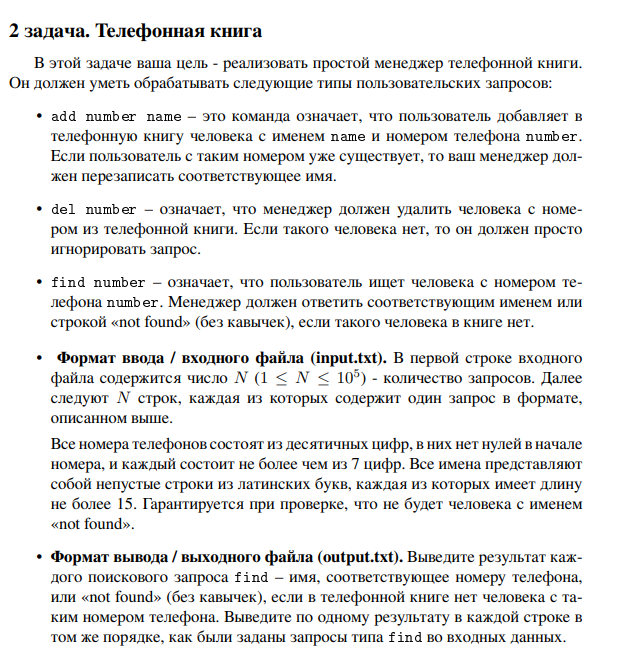
* **В среднем случае (равномерное распределение хешей):**
  + get\_bucket, add, remove, \_\_contains\_\_: O(1) — операции с бакетами выполняются за константное время.
  + \_\_iter\_\_: O(n) — необходимо пройти по всем элементам, где n - количество элементов в множестве.
* **В худшем случае (когда все элементы попадают в один бакет):**
  + get\_bucket: O(1).
  + add, remove, \_\_contains\_\_: O(n) – линейная сложность из-за прохода по бакету.
  + \_\_iter\_\_: O(n), где n - количество элементов в множестве.

**Пространственная сложность:** Пространственная сложность равна O(N + capacity), где N - это колличество элементов и capacity - это размер массива бакетов.

Вывод по задаче:

Реализовано простейшее множество с несколькими операциями.

## Задача №2. Телефонная книга.



Решение:

class PhoneBook:

   def \_\_init\_\_(self, capacity: int = 10 \*\* 6, hash\_func: Callable = hash):

       self.capacity: int = capacity

       self.hash\_func: Callable = hash\_func

       self.buckets: List[List[Tuple[int, str]]] = [[] for \_ in range(self.capacity)]

​

   def get\_bucket(self, key: int) -> List[Tuple[int, str]]:

       bucket\_ind = self.hash\_func(key) % self.capacity

       return self.buckets[bucket\_ind]

​

   def \_\_getitem\_\_(self, key: int) -> str:

       bucket = self.get\_bucket(key)

       for cur\_key, cur\_val in bucket:

           if cur\_key == key:

               return cur\_val

​

   def \_\_setitem\_\_(self, key: int, value: str) -> None:

       bucket = self.get\_bucket(key)

       for i, (cur\_key, cur\_val) in enumerate(bucket):

           if cur\_key == key:

               bucket[i] = (key, value)

               return

​

       bucket.append((key, value))

​

   def \_\_delitem\_\_(self, key: int) -> None:

       bucket = self.get\_bucket(key)

       for i, (cur\_key, \_) in enumerate(bucket):

           if cur\_key == key:

               del bucket[i]

               return

​

   def \_\_contains\_\_(self, key: int) -> bool:

       is\_contain = False

       for cur\_key, \_ in self.get\_bucket(key):

           if cur\_key == key:

               is\_contain = True

               break

       return is\_contain

​

   def \_\_iter\_\_(self):

       for bucket in self.buckets:

           for pair in bucket:

               yield pair

Текстовое объяснение решения:

Класс PhoneBook реализует простую хеш-таблицу (хеш-карту) для хранения телефонных номеров (ключей) и их соответствующих имен (значений). Разберем класс по частям:

**1. \_\_init\_\_(self, capacity: int = 10 \*\* 6, hash\_func: Callable = hash):**

* Конструктор класса. Принимает необязательные аргументы:
  + capacity: Вместимость хеш-таблицы (количество бакетов/списков). По умолчанию 10\*\*6.
  + hash\_func: Хеш-функция для вычисления хеш-значения ключа. По умолчанию использует встроенную функцию hash.
* Инициализирует:
  + self.capacity: Сохраняет вместимость.
  + self.hash\_func: Сохраняет хеш-функцию.
  + self.buckets: Создает список бакетов (списков), каждый из которых изначально пуст.

**2. get\_bucket(self, key: int) -> List[Tuple[int, str]]:**

* Вычисляет индекс бакета для заданного ключа, используя хеш-функцию и операцию взятия по модулю: self.hash\_func(key) % self.capacity.
* Возвращает бакет, соответствующий вычисленному индексу.

**3. \_\_getitem\_\_(self, key: int) -> str:**

* Реализует оператор [] для получения значения по ключу.
* Получает бакет для ключа.
* Итерирует по бакету, ища пару с ключом.
* Если находит пару, возвращает соответствующее значение. Если не находит, явно ничего не возвращает, а python автоматически вернёт None

**4. \_\_setitem\_\_(self, key: int, value: str) -> None:**

* Реализует оператор [] = для добавления или изменения значения по ключу.
* Получает бакет для ключа.
* Итерирует по бакету, ища пару с ключом.
* Если находит пару, обновляет ее значение.
* Если не находит, добавляет новую пару в конец бакета.

**5. \_\_delitem\_\_(self, key: int) -> None:**

* Реализует оператор del[] для удаления записи по ключу.
* Получает бакет для ключа.
* Итерирует по бакету, ища пару с ключом.
* Если находит пару, удаляет ее из бакета.

**6. \_\_contains\_\_(self, key: int) -> bool:**

* Реализует оператор in для проверки наличия ключа в хеш-таблице.
* Получает бакет для ключа.
* Итерирует по бакету, ища пару с ключом.
* Возвращает True, если находит пару, в противном случае False.

**7. \_\_iter\_\_(self):**

* Реализует возможность итерации по хеш-таблице (перебирает все пары).
* Итерирует по всем бакетам.
* Итерирует по парам в каждом бакете.
* Использует yield для генерации пар (ленивая итерация).

**Асимптотическая сложность:**

* **В среднем случае (равномерное распределение хешей):**
  + get\_bucket, \_\_getitem\_\_, \_\_setitem\_\_, \_\_delitem\_\_, \_\_contains\_\_: O(1) — операции с бакетами выполняются за константное время.
  + \_\_iter\_\_: O(n) — необходимо пройти по всем элементам, где n - количество элементов в хеш таблице.
* **В худшем случае (когда все ключи попадают в один бакет):**
  + get\_bucket: O(1).
  + \_\_getitem\_\_, \_\_setitem\_\_, \_\_delitem\_\_, \_\_contains\_\_: O(n) – линейная сложность из-за прохода по бакету.
  + \_\_iter\_\_: O(n), где n - количество элементов в хеш таблице.

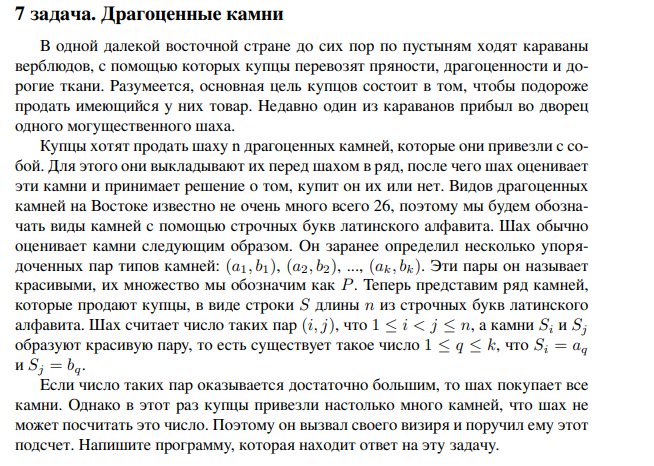
**Пространственная сложность:**

Пространственная сложность равна O(N + capacity), где N - это колличество элементов и capacity - это размер массива бакетов.

Вывод по задаче:

Реализован словарь для хранения имен для определенных номеров.

## Задача №7. Драгоценные камни.



Решение:

def find\_amount\_pairs\_in\_string(string: str, beauty\_pairs: Dict[str, List[str]]):

   counter = [0 for \_ in range(26)]

   ans = 0

   for let in string:

       if let in beauty\_pairs.keys():

           for first\_el in beauty\_pairs.get(let):

               ans += counter[ord(first\_el) - ord('a')]

       counter[ord(let) - ord('a')] += 1

​

   return ans

Текстовое объяснение решения:

Функция find\_amount\_pairs\_in\_string подсчитывает количество пар символов в строке string, которые присутствуют в словаре beauty\_pairs.

Разберем код по частям:

1. **counter = [0 for \_ in range(26)]**: Создается массив counter длиной 26, заполненный нулями. Этот массив будет использоваться для подсчета частоты появления каждого символа от ‘a’ до ‘z’ в строке string.
2. **ans = 0**: Инициализируется переменная ans, которая будет хранить общее количество пар, найденных в строке.
3. **for let in string**: Цикл проходит по каждому символу let в строке string.
4. **if let in beauty\_pairs.keys()**: Если текущий символ let является ключом в словаре beauty\_pairs, то:
   * **for first\_el in beauty\_pairs.get(let)**: Цикл перебирает все значения first\_el, ассоциированные с ключом let в словаре beauty\_pairs. Значения beauty\_pairs.get(let) представляет собой список символов, образующих “красивую пару” с let.
   * **ans += counter[ord(first\_el) - ord('a')]**: Для каждого символа first\_el из списка, функция получает частоту first\_el из массива counter и добавляет ее к ans. ord(first\_el) - ord('a') – это индекс символа в массиве counter.
5. **counter[ord(let) - ord('a')] += 1**: Частота появления текущего символа let увеличивается на 1 в массиве counter.
6. **return ans**: Возвращается общее количество пар ans.

**Асимптотическая сложность:**

* Цикл по символам string: O(n), где n – длина строки.
* Вложенный цикл по beauty\_pairs.get(let): В худшем случае этот цикл может иметь сложность O(m), где m – максимальная длина списка, сопоставленного с каждым ключом в beauty\_pairs.

Вывод по задаче:

Реализован алгоритм подсчета количества “красивых” пар символов в строке.

# Вывод

По условию лабораторной нужно было решить три задачи не стоящие подряд и номер одной найти по формуле H(v) = (A · v mod p) mod 9. Для меня H(21) = (39 · 21 mod 29) mod 9 = 7. Решены три задачи 1, 2, 7 по условию.