Лекция 20. Хеш-таблицы, множества, словари.

1. Введение

Хеш-таблица – это одна из самых популярных и широко используемых структур данных. Она позволяет эффективно хранить пары «ключ-значение» и выполнять основные операции (добавление, удаление, поиск) в среднем за время **O(1)**.

Основные интерфейсы:

• Set (**множество**) – хранит уникальные элементы. Поддерживает операции:

1. add(x) – добавить элемент.
2. remove(x) – удалить элемент.
3. contains(x) (или x in set) – проверить наличие элемента.

• Map (**словарь**, **ассоциативный массив**) – хранит пары «ключ-значение». Поддерживает операции:

1. put(key, value) (или map[key] = value) – добавить или обновить пару.
2. get(key) (или map[key]) – получить значение по ключу.
3. remove(key) – удалить пару по ключу.

Реализации:

На практике Set часто реализуется поверх Map, где ключ – это сам элемент, а значение – это заглушка (например, True).

2. Идеализированный случай: маленькие целочисленные ключи

Самый простой способ реализовать Map – это использовать обычный массив (список).

Идея. Если наши ключи – это маленькие целые числа из диапазона [0, n-1], то мы можем использовать их как индексы массива.

Реализация:

```
from typing import Any, List, Tuple, Optional, Union

# Тип для элемента в цепочке
Pair = Tuple[Any, Any]


class SimpleArrayMap:
    """Простая реализация Map для маленьких целочисленных ключей."""

    def __init__(self, size: int):
        # Инициализируем массив фиксированного размера, заполненный None
        # None означает, что ячейка пуста
        self._array = [None] * size

    def put(self, key: int, value: str):
        """
        Добавляет пару (key, value).
```

```
        Предполагается, что key всегда находится в диапазоне [0, size-1]
        """
        # Просто записываем значение в ячейку с индексом = key
        self._array[key] = value

    def get(self, key: int) -> Optional[str]:  # Исправлено: | заменено на
Optional
        """
        Возвращает значение по ключу или None, если ключ отсутствует.
        """
        # Возвращаем значение из ячейки с индексом = key
        # Если там None - значит ключа нет
        return self._array[key]

    def remove(self, key: int):
        """
        Удаляет пару по ключу.
        """
        # Устанавливаем значение ячейки в None, отмечая её как пустую
        self._array[key] = None


# Пример использования
simple_map = SimpleArrayMap(10)
simple_map.put(3, "Яблоко")
simple_map.put(7, "Банан")

print(simple_map.get(3))  # Вывод: Яблоко
print(simple_map.get(5))  # Вывод: None
```

Сложность. Все операции **O(1)**. Память: **O(n)**, где n – размер диапазона ключей.

3. Реальный случай: большие или нецелочисленные ключи

Проблема возникает, когда ключи большие (например, 10^9), нецелочисленные (строки, объекты) или их диапазон неизвестен. Заводить массив размером 10^9 непрактично.

Решение: хеш-функция

Хеш-функция – это функция h(key), которая преобразует произвольный ключ в целое число (хеш) из фиксированного диапазона [0, m-1], где m – размер внутреннего массива (хэш-таблицы).

- **Цель:** равномерно распределять ключи по ячейкам массива.
- **Требования:**

1. Должна быть детерминированной: один и тот же ключ всегда дает один и тот же хеш.
2. Должна быть быстрой для вычисления.
3. **Желательное свойство:** минимизировать количество **коллизий** – ситуаций, когда разные ключи имеют одинаковый хеш (h(key1) == h(key2) для key1 != key2).

Пример простой хеш-функции для целых чисел:

h(x) = x % m, где m – размер массива.

Базовая структура хеш-таблицы:

Теперь наш массив хранит не просто значения, а «ведра» (buckets) – списки пар (key, value). Это метод разрешения коллизий, называемый **цепочками (chaining)**.

```
from typing import Any, List, Tuple, Optional, Union

# Тип для элемента в цепочке
Pair = Tuple[Any, Any]


class NaiveHashMap:
    """Наивная реализация HashMap с использованием цепочек."""

    def __init__(self, capacity: int = 10):
        # capacity - размер внутреннего массива (количество корзин)
        self._capacity = capacity
        # Создаем массив пустых списков - это наши "корзины"
        # Каждая корзина будет хранить список пар (key, value)
        self._buckets: List[List[Pair]] = [[] for _ in range(self._capacity)]

    def _hash(self, key: Any) -> int:
        """
        Вычисляет индекс корзины для ключа.
        hash(key) - встроенная функция Python, возвращает хеш ключа
        % self._capacity - берем остаток от деления, чтобы попасть в диапазон
[0, capacity-1]
        """
        return hash(key) % self._capacity

    def put(self, key: Any, value: Any):
        """Добавляет или обновляет пару ключ-значение."""
        # 1. Вычисляем индекс корзины
        index = self._hash(key)
        # 2. Получаем саму корзину (список)
        bucket = self._buckets[index]

        # 3. Проверяем, нет ли уже такого ключа в цепочке
        for i, (k, v) in enumerate(bucket):
            if k == key:
                # Ключ найден - обновляем значение
                bucket[i] = (key, value)
                return
        # 4. Если ключ не найден - добавляем новую пару в конец списка
        bucket.append((key, value))

    def get(self, key: Any) -> Optional[Any]:  # Исправлено: | заменено на
Optional
        """Возвращает значение по ключу или None."""
        # 1. Вычисляем индекс корзины
        index = self._hash(key)
        # 2. Получаем корзину
```

```
        bucket = self._buckets[index]

        # 3. Линейно ищем ключ в корзине
        for k, v in bucket:
            if k == key:
                return v  # Нашли - возвращаем значение
        return None  # Не нашли - возвращаем None

    def remove(self, key: Any):
        """Удаляет пару по ключу."""
        index = self._hash(key)
        bucket = self._buckets[index]

        # Ищем ключ в корзине
        for i, (k, v) in enumerate(bucket):
            if k == key:
                # Нашли - удаляем элемент из списка
                del bucket[i]
                return
        # Ключ не найден - ничего не делаем


# Пример использования
naive_map = NaiveHashMap()
naive_map.put("apple", 1)
naive_map.put("banana", 2)
naive_map.put("apple", 3)  # Обновит значение для "apple"

print(naive_map.get("apple"))   # Вывод: 3
print(naive_map.get("cherry"))  # Вывод: None
```

4. Анализ сложности и коллизии

Сложность операций в наивной реализации:

• **В худшем случае (все ключи попали в одну корзину)**: **O(n)**, где n – количество элементов.

• **В среднем случае (при хорошей хеш-функции и равномерном распределении)**: **O(1 + α)**, где α = n/m – коэффициент заполнения.

Чем меньше α (чем больше «свободных» мест), тем ближе сложность к **O(1)**. На практике поддерживают α < 0.75 - 1.0.

Как бороться с коллизиями?

1. **Увеличить размер массива (m).** Если сделать m ~ n^2, то вероятность коллизии мала, но это расточительно по памяти.

2. **Использовать хорошую хеш-функцию.** На практике используют **универсальные семейства хеш-функций**, выбирая случайную функцию из набора при создании таблицы, чтобы избежать атак, когда злоумышленник специально создает ключи с коллизиями.

5. Динамическое расширение (rehashing)

Поскольку мы заранее не знаем количество элементов, массив (список) buckets нужно уметь расширять.

Идея. Когда коэффициент заполнения α превышает некоторый порог (например, 0.75), мы:

1. Создаем новый массив buckets в 2 раза больше (или другого простого размера).
2. Проходим по всем элементам старой таблицы и заново вычисляем их хеш (индекс) для нового массива.
3. Перемещаем их в новые корзины.

Это дорогая операция (**O(n)**), но если делать ее достаточно редко, ее стоимость **амортизируется** до **O(1)** на одну операцию вставки.

```
from typing import Any, List, Tuple, Optional, Union

# Тип для элемента в цепочке
Pair = Tuple[Any, Any]


class ImprovedHashMap:
    """Улучшенная HashMap с динамическим расширением."""

    def __init__(self, capacity: int = 8, load_factor: float = 0.75):
        self._capacity = capacity
        self._load_factor = load_factor  # Коэффициент заполнения для
расширения
        self._buckets: List[List[Pair]] = [[] for _ in range(self._capacity)]
        self._size = 0  # Счетчик количества элементов в таблице

    def _hash(self, key: Any) -> int:
        return hash(key) % self._capacity

    def _resize(self):
        """
        Увеличивает размер таблицы в 2 раза и перехеширует все элементы.
        Вызывается когда таблица становится слишком заполненной.
        """
        # Сохраняем старые данные
        old_buckets = self._buckets
        # Увеличиваем capacity в 2 раза
        self._capacity *= 2
        # Создаем новый массив корзин
        self._buckets = [[] for _ in range(self._capacity)]
        self._size = 0  # Временно сбрасываем размер

        # Перемещаем все элементы из старых корзин в новые
        for bucket in old_buckets:
            for key, value in bucket:
                # put автоматически пересчитает хеш для нового размера
                self.put(key, value)
```

```
    def put(self, key: Any, value: Any):
        # Проверяем, не нужно ли расширение перед добавлением
        if (self._size + 1) / self._capacity > self._load_factor:
            self._resize()  # Выполняем расширение если нужно

        index = self._hash(key)
        bucket = self._buckets[index]

        for i, (k, v) in enumerate(bucket):
            if k == key:
                bucket[i] = (key, value)  # Обновление
                return
        bucket.append((key, value))
        self._size += 1  # Увеличиваем счетчик только для новых элементов

    def get(self, key: Any) -> Optional[Any]:  # Исправлено: | заменено на
Optional
        # Аналогично NaiveHashMap
        index = self._hash(key)
        bucket = self._buckets[index]
        for k, v in bucket:
            if k == key:
                return v
        return None

    def remove(self, key: Any):
        index = self._hash(key)
        bucket = self._buckets[index]
        for i, (k, v) in enumerate(bucket):
            if k == key:
                del bucket[i]
                self._size -= 1  # Уменьшаем счетчик при удалении
                return


# Пример использования
naive_map = ImprovedHashMap()
naive_map.put("apple", 1)
naive_map.put("banana", 2)
naive_map.put("apple", 3)  # Обновит значение для "apple"

print(naive_map.get("apple"))   # Вывод: 3
print(naive_map.get("cherry"))  # Вывод: None
```

6. Альтернативный метод: открытая адресация (Open Addressing)

В этом методе все пары хранятся непосредственно в массиве, а не в списках. При коллизии мы ищем следующую свободную ячейку по определенному алгоритму (пробингу).

Линейное пробирование. Если ячейка h(key) занята, проверяем h(key)+1, h(key)+2, … по модулю m.

Преимущества:

• Не требует создания списков, что экономит память и хорошо для кэширования процессора.

• В целом может быть быстрее на практике для небольших таблиц.

Недостатки:

• Более сложное удаление элементов (нельзя просто удалить, иначе разорвется цепочка пробинга). Обычно ставят специальную метку «удалено».

• Сильнее подвержены эффекту кластеризации (образованию длинных последовательностей занятых ячеек).

Реализация открытой адресации с линейным пробированием:

```
from typing import Any, List, Tuple, Optional, Union

# Тип для элемента в цепочке
Pair = Tuple[Any, Any]


class OpenAddrHashMap:
    """Реализация HashMap с открытой адресацией и линейным пробированием."""

    # Специальный маркер для удаленных элементов (не None, чтобы отличать от пустых)
    _DELETED = object()

    def __init__(self, capacity: int = 8, load_factor: float = 0.7):
        self._capacity = capacity
        self._load_factor = load_factor
        # Массив хранит либо None, либо _DELETED, либо пару (key, value)
        self._array: List[Any] = [None] * self._capacity
        self._size = 0

    def _hash(self, key: Any) -> int:
        return hash(key) % self._capacity

    def _find_index(self, key: Any) -> int:
        """
        Находит индекс для ключа, либо индекс, куда его можно вставить.
        Возвращает индекс если ключ найден, или индекс для вставки если не найден.
        """
        index = self._hash(key)
        first_deleted = -1  # Запоминаем первую удаленную ячейку

        # Проходим по цепочке, пока не найдем ключ или пустую ячейку
        while self._array[index] is not None:
            if self._array[index] == self._DELETED:
                # Запоминаем первую удаленную ячейку (можем использовать для вставки)
                if first_deleted == -1:
                    first_deleted = index
            elif self._array[index][0] == key:  # Сравниваем ключ
                return index  # Ключ найден
            # Линейное пробирование: переходим к следующей ячейке
```

```
            index = (index + 1) % self._capacity

        # Ключ не найден. Если нашли удаленную ячейку, вернем её
        return first_deleted if first_deleted != -1 else index

    def put(self, key: Any, value: Any):
        # Проверяем необходимость расширения
        if (self._size + 1) / self._capacity > self._load_factor:
            self._resize()

        # Находим подходящий индекс
        index = self._find_index(key)
        if self._array[index] is None or self._array[index] == self._DELETED:
            # Вставляем новую пару
            self._array[index] = (key, value)
            self._size += 1
        else:
            # Обновляем существующую пару
            self._array[index] = (key, value)

    def get(self, key: Any) -> Optional[Any]:  # Исправлено: | заменено на
Optional
        index = self._find_index(key)
        # Проверяем, что нашли существующий элемент (не None и не _DELETED)
        if index != -1 and self._array[index] is not None and
self._array[index] != self._DELETED:
            return self._array[index][1]  # Возвращаем значение
        return None

    def remove(self, key: Any):
        index = self._find_index(key)
        if index != -1 and self._array[index] is not None and
self._array[index] != self._DELETED:
            # Помечаем ячейку как удаленную (не делаем None, чтобы не порвать
цепочку)
            self._array[index] = self._DELETED
            self._size -= 1

    def _resize(self):
        """Пере хеширование для открытой адресации."""
        old_array = self._array
        self._capacity *= 2
        self._array = [None] * self._capacity
        self._size = 0
        # Перебираем все элементы старого массива
        for item in old_array:
            if item is not None and item != self._DELETED:
                key, value = item
                # Вставляем в новую таблицу
                self.put(key, value)


# Пример использования
naive_map = OpenAddrHashMap()
naive_map.put("apple", 1)
```

```
naive_map.put("banana", 2)
naive_map.put("apple", 3)  # Обновит значение для "apple"

print(naive_map.get("apple"))   # Вывод: 3
print(naive_map.get("cherry"))  # Вывод: None
```

7. Множества (Set) на основе хеш-таблицы

Реализация Set тривиальна на основе Map. Мы храним ключи, а значения-заглушки игнорируем.

```
from typing import Any, List, Tuple, Optional, Union

# Тип для элемента в цепочке
Pair = Tuple[Any, Any]


class ImprovedHashMap:…


class HashSet:
    """Реализация Set с использованием HashMap."""

    def __init__(self):
        # Используем нашу улучшенную HashMap как основу
        self._map = ImprovedHashMap()

    def add(self, item: Any):
        """Добавляет элемент в множество."""
        # Используем элемент как ключ, значение - заглушка (True)
        self._map.put(item, True)

    def remove(self, item: Any):
        """Удаляет элемент из множества."""
        self._map.remove(item)

    def contains(self, item: Any) -> bool:
        """Проверяет, содержится ли элемент в множестве."""
        # Если get возвращает не None - элемент есть в множестве
        return self._map.get(item) is not None

    def __contains__(self, item: Any) -> bool:
        """Специальный метод для поддержки оператора 'in'."""
        return self.contains(item)


# Пример использования
my_set = HashSet()
my_set.add("a")
my_set.add("b")
my_set.add("a")  # Дубликат не добавится

print(my_set.contains("a"))  # Вывод: True
print("c" in my_set)  # Вывод: False
```

9. Сравнение методов разрешения коллизий

Метод	Преимущества	Недостатки	Применение
Цепочки	Проще в реализации, особенно удаление. Устойчив к высокому α.	Требует дополнительной памяти на ссылки в списках. Хуже локализация данных (кэш-промахи).	Хороший универсальный выбор.
Открытая адресация	Лучшая использование памяти (нет накладных расходов на списки). Отличная локализация данных.	Сложное удаление. Сильная деградация при высоком α (кластеризация).	Хорошо для небольших, плотных таблиц, где важен кэш.

10. Итоговая таблица алгоритмов

Структура	Средняя сложность (вставка, удаление, поиск)	Худший случай	Ключевое применение
dict / set (**хеш-таблица**)	**O(1)**	**O(n)**	Универсальное хранение пар «ключ-значение» и уникальных элементов. Быстрый поиск по ключу.
Наивная HashMap (**без рехешинга**)	O(1 + α)	O(n)	Учебные цели, когда количество элементов известно и мало.
Улучшенная HashMap (**с рехешингом**)	**O(1)** (амортизир.)	O(n)	Практическая реализация, когда количество элементов заранее неизвестно.