Лекция 21. Идеальное хеширование, хеширование кукушки, фильтр Блума.

1. Введение

Мы изучили стандартные хеш-таблицы, которые в среднем обеспечивают операции за **O(1)**, но в худшем случае (при коллизиях) время деградирует до **O(n)**. Это происходит, когда многие ключи попадают в одну корзину, и поиск требует линейного прохода по списку.

На практике такое поведение не всегда приемлемо, особенно в системах реального времени, где важна **предсказуемость времени выполнения**. Также есть сценарии, когда данные известны заранее (статический набор ключей), и хочется построить структуру, гарантирующую **константное время поиска без коллизий**.

В этой лекции рассмотрим:

1. **Идеальное хеширование** – метод построения хеш-таблицы без коллизий для статического набора ключей.
2. **Хеширование кукушки** – динамическая схема с гарантированным O(1) для поиска и вставки, использующая две хеш-функции.
3. **Фильтр Блума** – вероятностная структура для проверки принадлежности элемента множеству с экономией памяти.

2. Идеальное хеширование

2.1 Основная идея

Идеальное хеширование строится для **статического набора ключей** (ключи известны заранее и не меняются). Цель – создать хеш-таблицу, в которой **нет коллизий** и поиск всегда работает за **O(1)**.

Подход:

- Используем **двухуровневую схему**.
- Первый уровень: обычная хеш-таблица с m корзинами.
- Второй уровень: для каждой корзины строится **отдельная маленькая хеш-таблица** без коллизий.

2.2 Двухуровневая схема

1. Выбираем хеш-функцию h1, которая распределяет ключи по m корзинам.
2. Для каждой корзины i строится своя хеш-таблица размера mi, где mi – квадрат количества ключей в этой корзине (это важно для гарантии отсутствия коллизий).
3. Для каждой корзины подбирается своя хеш-функция h2_i, пока не будет достигнуто отсутствие коллизий.

2.3 Почему квадрат размера?

Если в корзине k ключей, то для таблицы размера k^2 вероятность коллизий при случайной хеш-функции мала. Можно показать, что **матожидание общего размера** всех маленьких таблиц остаётся **линейным** относительно общего числа ключей n.

2.4 Реализация

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


class PerfectHashTable:
    """Идеальная хеш-таблица для статического набора ключей."""

    def __init__(self, keys: List[Any], values: List[Any]):
        """
        Строит идеальную хеш-таблицу для заданных ключей и значений.
        Предполагается, что ключи уникальны.
        """
        self.n = len(keys)
        self.m = self.n  # размер первой таблицы (можно взять другой,
например, n//2)
        self.level1 = [None] * self.m  # первая таблица - хранит ссылки на
таблицы второго уровня
        self.level2 = [None] * self.m  # вторая таблица - массив массивов для
каждой корзины
        self.level2_size = [0] * self.m  # размеры таблиц второго уровня
        self.level2_hash = [None] * self.m  # параметры хеш-функций для
второго уровня

        # 1. Распределяем ключи по корзинам первого уровня
        # Создаем список корзин (buckets) - каждая корзина это список пар
(ключ, значение)
        buckets = [[] for _ in range(self.m)]
        for k, v in zip(keys, values):
            idx = self._hash1(k)  # вычисляем, в какую корзину попадет ключ
            buckets[idx].append((k, v))  # добавляем пару в соответствующую
корзину

        # 2. Для каждой корзины строим свою таблицу без коллизий
        for i in range(self.m):
            if not buckets[i]:  # если корзина пустая, пропускаем
                continue
            self._build_secondary(i, buckets[i])  # строим таблицу второго
уровня для корзины i

    def _hash1(self, key: Any) -> int:
        """Хеш-функция для первого уровня. Простой остаток от деления."""
        return hash(key) % self.m
```

```python
    def _hash2(self, key: Any, a: int, b: int, size: int) -> int:
        """
        Универсальная хеш-функция для второго уровня: (a * hash(key) + b) %
size.
        a, b - случайные коэффициенты, size - размер таблицы.
        """
        return (a * hash(key) + b) % size

    def _build_secondary(self, bucket_idx: int, pairs: List[Tuple[Any,
Any]]):
        """Строит таблицу второго уровня для одной корзины."""
        k = len(pairs)  # количество элементов в корзине
        size = k * k  # квадрат количества элементов (чтобы уменьшить
вероятность коллизий)
        if size == 0:
            size = 1  # на случай пустой корзины (защита от деления на 0)

        # Подбираем хеш-функцию, пока не найдем без коллизий
        while True:
            # Выбираем случайные коэффициенты для хеш-функции
            a = random.randint(1, 1_000_000)
            b = random.randint(0, 1_000_000)
            table = [None] * size  # создаем новую таблицу
            collision = False  # флаг коллизии

            # Пытаемся разместить все элементы корзины
            for key, value in pairs:
                idx = self._hash2(key, a, b, size)  # вычисляем позицию во
второй таблице
                if table[idx] is not None:  # если ячейка уже занята -
коллизия
                    collision = True
                    break  # прерываем цикл - нужна новая хеш-функция
                table[idx] = (key, value)  # размещаем элемент

            if not collision:  # если удалось разместить все без коллизий
                self.level2[bucket_idx] = table  # сохраняем таблицу
                self.level2_size[bucket_idx] = size  # сохраняем размер
                self.level2_hash[bucket_idx] = (a, b)  # сохраняем параметры
хеш-функции
                break  # выходим из цикла подбора

    def get(self, key: Any) -> Optional[Any]:
        """Возвращает значение по ключу или None, если ключа нет."""
        # 1. Вычисляем корзину в первом уровне
        idx1 = self._hash1(key)

        # 2. Если для этой корзины нет таблицы второго уровня, ключа нет
        if self.level2[idx1] is None:
            return None

        # 3. Получаем параметры хеш-функции для этой корзины
        a, b = self.level2_hash[idx1]
        size = self.level2_size[idx1]
```

```
        # 4. Вычисляем позицию во втором уровне
        idx2 = self._hash2(key, a, b, size)

        # 5. Получаем элемент из таблицы второго уровня
        pair = self.level2[idx1][idx2]

        # 6. Проверяем, что ключ совпадает (на случай ложного срабатывания)
        if pair is not None and pair[0] == key:
            return pair[1]  # возвращаем значение
        return None  # ключа нет


# Пример использования
keys = ["apple", "banana", "cherry", "date", "elderberry"]
values = [1, 2, 3, 4, 5]

ph = PerfectHashTable(keys, values)
print(ph.get("apple"))  # 1
print(ph.get("banana"))  # 2
print(ph.get("fig"))  # None (ключа нет)
```

2.5 Анализ сложности

• **Построение:** O(n) в среднем (подбор хеш-функций может потребовать нескольких попыток).

• **Поиск:** гарантированно **O(1)** – два вычисления хеша и доступ к массиву.

• **Память:** O(n) в среднем (сумма квадратов размеров корзин линейна).

Недостатки:

• Только для статических данных.

• Затраты на построение могут быть высокими.

• Потребление памяти больше, чем у обычной хеш-таблицы.

3. Хеширование кукушки

3.1 Основная идея

Хеширование кукушки – это **динамическая** схема, которая использует **две хеш-функции** и **одну таблицу**. Каждый ключ может находиться в одной из двух возможных позиций, определяемых этими функциями.

Принцип работы:

• Для ключа x есть две возможные ячейки: h1(x) и h2(x).

• При вставке, если обе ячейки заняты, один из существующих ключей «вытесняется» (как кукушка подбрасывает яйца в чужие гнёзда) на его альтернативную позицию.

• Процесс может вызвать цепочку вытеснений.

3.2 Алгоритм вставки

1. Вычислить pos1 = h1(key) и pos2 = h2(key).
2. Если одна из ячеек свободна – поместить ключ туда.
3. Если обе заняты:

• Вытеснить ключ из одной из ячеек (например, из pos1).
• Поместить новый ключ в pos1.
• Переместить вытесненный ключ в его альтернативную позицию.
• Если и та занята – повторить процесс рекурсивно.

Если процесс зацикливается (обнаруживается цикл), таблица перестраивается с новыми хеш-функциями.

3.3 Реализация

```
from typing import Any, Optional
import random


class CuckooHashTable:
    """Хеш-таблица с использованием хеширования кукушки."""

    def __init__(self, capacity: int = 8):
        self.capacity = capacity  # начальный размер таблицы
        self.table = [None] * capacity  # основная таблица (массив ячеек)
        self.size = 0  # количество элементов в таблице
        self.load_factor = 0.5  # коэффициент заполнения для расширения
        self._init_hash_functions()  # инициализируем хеш-функции

    def _init_hash_functions(self):
        """Инициализирует две случайные хеш-функции."""
        self.a1 = random.randint(1, 1000)  # коэффициент для первой функции
        self.b1 = random.randint(0, 1000)  # свободный член для первой
        self.a2 = random.randint(1, 1000)  # коэффициент для второй функции
        self.b2 = random.randint(0, 1000)  # свободный член для второй

    def _hash1(self, key: Any) -> int:
        """Первая хеш-функция: (a1 * hash(key) + b1) % capacity."""
        return (self.a1 * hash(key) + self.b1) % self.capacity

    def _hash2(self, key: Any) -> int:
        """Вторая хеш-функция: (a2 * hash(key) + b2) % capacity."""
        return (self.a2 * hash(key) + self.b2) % self.capacity

    def _rehash(self):
        """Перестраивает таблицу с новыми хеш-функциями и вдвое большим
размером."""
        old_table = self.table  # сохраняем старую таблицу
        self.capacity *= 2  # удваиваем размер
        self.table = [None] * self.capacity  # создаем новую пустую таблицу
        self.size = 0  # сбрасываем счетчик
        self._init_hash_functions()  # генерируем новые хеш-функции
```

```
        # Перемещаем все элементы из старой таблицы в новую
        for item in old_table:
            if item is not None:  # если ячейка не пуста
                self.put(item[0], item[1])  # вставляем элемент заново

    def put(self, key: Any, value: Any):
        """Вставляет пару ключ-значение в таблицу."""
        # Проверяем, не нужно ли расширить таблицу
        if self.size / self.capacity > self.load_factor:
            self._rehash()

        # Пытаемся вставить элемент, делаем не более 100 попыток
        for _ in range(100):  # ограничение на длину цепочки вытеснений
            pos1 = self._hash1(key)  # первая возможная позиция

            # Если первая позиция свободна - размещаем элемент
            if self.table[pos1] is None:
                self.table[pos1] = (key, value)
                self.size += 1
                return  # успешно вставили

            # Если первая позиция занята - вытесняем элемент оттуда
            evicted_key, evicted_value = self.table[pos1]  # сохраняем
вытесняемый элемент
            self.table[pos1] = (key, value)  # размещаем новый элемент

            # Теперь нужно разместить вытесненный элемент
            key, value = evicted_key, evicted_value  # теперь это наш текущий
элемент для вставки

            # Пробуем вторую позицию для вытесненного элемента
            pos2 = self._hash2(key)
            if self.table[pos2] is None:  # если вторая позиция свободна
                self.table[pos2] = (key, value)
                self.size += 1
                return  # успешно вставили

            # Если и вторая позиция занята - вытесняем элемент и оттуда
            evicted_key, evicted_value = self.table[pos2]
            self.table[pos2] = (key, value)
            key, value = evicted_key, evicted_value  # продолжаем цепочку
вытеснений

        # Если цепочка вытеснений слишком длинная (более 100 шагов)
        # - перестраиваем таблицу с новыми хеш-функциями
        self._rehash()
        self.put(key, value)  # пробуем вставить элемент снова

    def get(self, key: Any) -> Optional[Any]:
        """Возвращает значение по ключу или None, если ключа нет."""
        # Проверяем первую возможную позицию
        pos1 = self._hash1(key)
        if self.table[pos1] is not None and self.table[pos1][0] == key:
            return self.table[pos1][1]  # нашли в первой позиции
```

```
        # Проверяем вторую возможную позицию
        pos2 = self._hash2(key)
        if self.table[pos2] is not None and self.table[pos2][0] == key:
            return self.table[pos2][1]  # нашли во второй позиции

        return None  # ключа нет ни в одной позиции

    def remove(self, key: Any):
        """Удаляет ключ из таблицы."""
        # Ищем ключ в первой позиции
        pos1 = self._hash1(key)
        if self.table[pos1] is not None and self.table[pos1][0] == key:
            self.table[pos1] = None  # удаляем элемент
            self.size -= 1
            return

        # Ищем ключ во второй позиции
        pos2 = self._hash2(key)
        if self.table[pos2] is not None and self.table[pos2][0] == key:
            self.table[pos2] = None  # удаляем элемент
            self.size -= 1


# Пример использования
ct = CuckooHashTable()
ct.put("apple", 1)
ct.put("banana", 2)
ct.put("cherry", 3)

print(ct.get("apple"))  # 1
print(ct.get("banana"))  # 2
print(ct.get("cherry"))  # 3
ct.remove("banana")
print(ct.get("banana"))  # None (удалили)
```

3.4 Анализ сложности

Поиск: O(1) – проверка двух ячеек.
Вставка: в среднем **O(1)**, но возможны дорогие перестройки.
Удаление: O(1).
Преимущества:
- Гарантированное O(1) для поиска.
- Нет списков – лучше использование кэша процессора.
- Динамическое расширение.

Недостатки:
- Сложная вставка с возможными перестройками.
- Требует две хеш-функции и больше памяти для избегания циклов.

4. Фильтр Блума

4.1 Основная идея

Фильтр Блума – это **вероятностная структура данных**, которая позволяет **проверять принадлежность элемента множеству** с небольшим объёмом памяти. Он может давать **ложноположительные срабатывания** (сказать, что элемент есть, когда его нет), но **не даёт ложноотрицательных**.

Применение:

- Проверка орфографии.
- Кэширование в базах данных.
- Сетевые фильтры.

4.2 Структура

Фильтр Блума – это **битовый массив** длины m и k хеш-функций.

Операции:

Добавление: для элемента вычисляются k хешей, соответствующие биты устанавливаются в 1.

Проверка: вычисляются те же k хешей; если все биты равны 1 – элемент, вероятно, в множестве.

4.3 Вероятность ошибки

Вероятность ложноположительного срабатывания зависит от:

- размера массива m,
- числа хеш-функций k,
- числа элементов n.

Оптимальное значение $k \approx (m/n) * \ln(2)$.

4.4 Реализация

```python
import math
import random
from typing import Any


class BloomFilter:
    """Простой фильтр Блума - вероятностная структура данных."""

    def __init__(self, n: int, p: float):
        """
        Инициализирует фильтр Блума.
        :param n: ожидаемое количество элементов
        :param p: желаемая вероятность ложноположительного срабатывания
        """
        self.n = n  # ожидаемое количество элементов
```

```
        self.p = p  # желаемая вероятность ошибки
        self.m = self._optimal_m(n, p)  # оптимальный размер битового массива
        self.k = self._optimal_k(self.m, n)  # оптимальное количество хеш-
функций
        self.bit_array = [0] * self.m  # битовый массив (0/1)
        self._init_hash_functions()  # инициализируем хеш-функции

    def _optimal_m(self, n: int, p: float) -> int:
        """
        Вычисляет оптимальный размер битового массива m по формуле:
        m = - (n * ln(p)) / (ln(2)^2)
        """
        m = - (n * math.log(p)) / (math.log(2) ** 2)
        return int(m)  # округляем до целого

    def _optimal_k(self, m: int, n: int) -> int:
        """
        Вычисляет оптимальное количество хеш-функций k по формуле:
        k = (m / n) * ln(2)
        """
        k = (m / n) * math.log(2)
        return max(1, int(k))  # минимум 1 функция

    def _init_hash_functions(self):
        """Инициализирует k хеш-функций со случайными параметрами."""
        self.hash_params = []  # список параметров (a, b) для каждой функции
        for i in range(self.k):
            a = random.randint(1, 1000)  # случайный коэффициент
            b = random.randint(0, 1000)  # случайное смещение
            self.hash_params.append((a, b))

    def _hash(self, key: Any, a: int, b: int) -> int:
        """
        Универсальная хеш-функция для одной из k функций.
        Возвращает индекс в битовом массиве.
        """
        return (a * hash(key) + b) % self.m

    def add(self, key: Any):
        """Добавляет элемент в фильтр Блума."""
        # Для каждой из k хеш-функций вычисляем позицию и устанавливаем бит в
1
        for a, b in self.hash_params:
            idx = self._hash(key, a, b)  # вычисляем позицию
            self.bit_array[idx] = 1  # устанавливаем бит

    def contains(self, key: Any) -> bool:
        """
        Проверяет, возможно ли, что элемент находится в фильтре.
        Может вернуть True для элемента, которого нет (ложноположительное).
        Но никогда не вернет False для элемента, который есть
(ложноотрицательных нет).
        """
```

```
        # Для каждой из k хеш-функций проверяем соответствующий бит
        for a, b in self.hash_params:
            idx = self._hash(key, a, b)  # вычисляем позицию
            if self.bit_array[idx] == 0:  # если хотя бы один бит равен 0
                return False  # элемент точно не добавлялся
        return True  # все биты равны 1 - элемент, вероятно, был добавлен


# Пример использования
bf = BloomFilter(n=1000, p=0.01)  # ожидаем 1000 элементов, вероятность ошибки 1%
bf.add("apple")
bf.add("banana")

print(bf.contains("apple"))  # True (точно был добавлен)
print(bf.contains("banana"))  # True (точно был добавлен)
print(bf.contains("cherry"))  # False или True с вероятностью ~1%
```

4.5 Анализ сложности

- **Добавление:** O(k)
- **Проверка:** O(k)

Преимущества:

- Экономия памяти.
- Быстрая вставка и проверка.

Недостатки:

- Ложноположительные срабатывания.
- Нельзя удалять элементы (без модификаций).

5. Сравнение методов

Метод	Гарантия O(1) поиска	Динамичность	Ложноположительные	Память	Применение
Идеальное хеширование	Да	Нет	Нет	O(n)	Статические словари, базы данных
Хеширование кукушки	Да	Да	Нет	O(n)	Кэши, динамические таблицы
Фильтр Блума	Нет (вероятностный)	Да	Да	O(1) на элемент	Орфография, сетевые фильтры