# Лекция: Хеш-таблицы и хеш-функции

## Введение

Хеш-таблица (hash table) - это структура данных, которая реализует интерфейс ассоциативного массива, то есть позволяет хранить пары ключ-значение и выполнять три основные операции: вставку, поиск и удаление элементов. Основное преимущество хеш-таблиц - это средняя константная временная сложность O(1) для этих операций в лучшем случае.

## 1. Основные понятия

### 1.1 Хеш-функция

Хеш-функция - это функция, которая преобразует ключ в индекс массива (хеш-код). Идеальная хеш-функция:

- Быстро вычисляется

- Равномерно распределяет ключи по всему диапазону индексов

- Минимизирует коллизии (ситуации, когда разные ключи дают одинаковый хеш)

### 1.2 Коллизии

Коллизия возникает, когда два разных ключа хешируются в один и тот же индекс. Существует два основных метода разрешения коллизий:

1. Метод цепочек (chaining)

2. Метод открытой адресации (open addressing)

## 2. Реализация хеш-таблицы с методом цепочек

Рассмотрим реализацию на Python:

class HashTable:

def \_\_init\_\_(self, size=10):

self.size = size

self.table = [[] for \_ in range(size)] # Создаем список списков

def \_hash(self, key):

# Простая хеш-функция: сумма ASCII-кодов символов ключа по модулю size

return sum(ord(c) for c in str(key)) % self.size

def insert(self, key, value):

hash\_key = self.\_hash(key)

bucket = self.table[hash\_key]

# Проверяем, нет ли уже такого ключа в цепочке

for i, (k, v) in enumerate(bucket):

if k == key:

bucket[i] = (key, value) # Обновляем значение, если ключ существует

return

bucket.append((key, value)) # Добавляем новую пару ключ-значение

def get(self, key):

hash\_key = self.\_hash(key)

bucket = self.table[hash\_key]

for k, v in bucket:

if k == key:

return v

raise KeyError(key)

def delete(self, key):

hash\_key = self.\_hash(key)

bucket = self.table[hash\_key]

for i, (k, v) in enumerate(bucket):

if k == key:

del bucket[i]

return

raise KeyError(key)

def \_\_str\_\_(self):

return str(self.table)

# Пример использования

ht = HashTable()

ht.insert("apple", 10)

ht.insert("orange", 20)

ht.insert("banana", 30)

ht.insert("apple", 15) # Обновляем значение для "apple"

print(ht) # Выводим всю таблицу

print(ht.get("orange")) # 20

ht.delete("orange")

print(ht.get("orange")) # KeyError

## 3. Хеш-функции

### 3.1 Требования к хорошей хеш-функции

1. \*\*Детерминированность\*\*: один и тот же ключ всегда должен давать один и тот же хеш

2. \*\*Равномерность\*\*: ключи должны равномерно распределяться по всему диапазону

3. \*\*Эффективность\*\*: функция должна быстро вычисляться

### 3.2 Примеры хеш-функций

#### Простая хеш-функция для строк (не рекомендуется для production):

```python

def simple\_hash(key, table\_size):

return sum(ord(c) for c in str(key)) % table\_size

```

#### Хеш-функция djb2 (одна из лучших для строк):

```python

def djb2\_hash(key, table\_size):

hash\_value = 5381

for c in str(key):

hash\_value = (hash\_value \* 33) + ord(c)

return hash\_value % table\_size

```

#### Хеш-функция для чисел (метод умножения):

```python

def multiplicative\_hash(key, table\_size):

# Константа A ≈ (√5 - 1)/2 \* 2^w, где w - разрядность машины

A = 2654435769 # Для 32-битных систем

return (key \* A) % (2\*\*32) % table\_size

```

## 4. Реализация хеш-таблицы с открытой адресацией

При открытой адресации все элементы хранятся непосредственно в массиве, а при коллизии ищется следующая свободная ячейка.

```python

class OpenAddressingHashTable:

def \_\_init\_\_(self, size=10):

self.size = size

self.table = [None] \* size

self.deleted = object() # Маркер для удаленных элементов

def \_hash(self, key, attempt=0):

# Линейное пробирование: hash(key) + attempt

return (hash(key) + attempt) % self.size

def insert(self, key, value):

for attempt in range(self.size):

hash\_key = self.\_hash(key, attempt)

if self.table[hash\_key] is None or self.table[hash\_key] is self.deleted:

self.table[hash\_key] = (key, value)

return

elif self.table[hash\_key][0] == key:

self.table[hash\_key] = (key, value) # Обновление

return

raise Exception("Хеш-таблица переполнена")

def get(self, key):

for attempt in range(self.size):

hash\_key = self.\_hash(key, attempt)

if self.table[hash\_key] is None:

break

elif self.table[hash\_key] is not self.deleted and self.table[hash\_key][0] == key:

return self.table[hash\_key][1]

raise KeyError(key)

def delete(self, key):

for attempt in range(self.size):

hash\_key = self.\_hash(key, attempt)

if self.table[hash\_key] is None:

break

elif self.table[hash\_key] is not self.deleted and self.table[hash\_key][0] == key:

self.table[hash\_key] = self.deleted

return

raise KeyError(key)

def \_\_str\_\_(self):

return str(self.table)

# Пример использования

oht = OpenAddressingHashTable()

oht.insert(1, "one")

oht.insert(11, "eleven") # Возможна коллизия с 1

oht.insert(21, "twenty-one") # Еще одна коллизия

print(oht.get(1)) # "one"

print(oht.get(11)) # "eleven"

oht.delete(11)

print(oht.get(11)) # KeyError

```

## 5. Решение проблем хеш-таблиц

### 5.1 Увеличение размера таблицы (rehashing)

Когда таблица заполняется (обычно при коэффициенте загрузки > 0.7), нужно увеличить ее размер и перехешировать все элементы.

```python

class ResizableHashTable(HashTable):

def \_\_init\_\_(self, initial\_size=10, load\_factor=0.7):

super().\_\_init\_\_(initial\_size)

self.load\_factor = load\_factor

self.count = 0

def insert(self, key, value):

if self.count / self.size > self.load\_factor:

self.\_resize()

super().insert(key, value)

self.count += 1

def \_resize(self):

new\_size = self.size \* 2

new\_table = [[] for \_ in range(new\_size)]

old\_table = self.table

self.table = new\_table

self.size = new\_size

self.count = 0

for bucket in old\_table:

for key, value in bucket:

self.insert(key, value)

# Пример использования

rht = ResizableHashTable(5, 0.6)

for i in range(10):

rht.insert(f"key{i}", i)

print(f"Inserted key{i}, size: {rht.size}, count: {rht.count}")

```

### 5.2 Уменьшение влияния коллизий

1. \*\*Хорошая хеш-функция\*\*: как показано выше

2. \*\*Увеличение размера таблицы\*\*: простые числа как размер уменьшают коллизии

3. \*\*Двойное хеширование\*\*: использование второй хеш-функции для пробирования

## 6. Применение хеш-таблиц

1. \*\*Базы данных\*\*: для индексации

2. \*\*Кеширование\*\*: например, memoization

3. \*\*Сети\*\*: таблицы маршрутизации

4. \*\*Компиляторы\*\*: таблицы символов

5. \*\*Алгоритмы\*\*: подсчет частот, поиск дубликатов

## Заключение

Хеш-таблицы - это мощная структура данных, обеспечивающая эффективные операции вставки, поиска и удаления. Понимание принципов их работы, хеш-функций и методов разрешения коллизий важно для каждого программиста. Реализации могут варьироваться в зависимости от языка программирования и конкретных требований приложения.