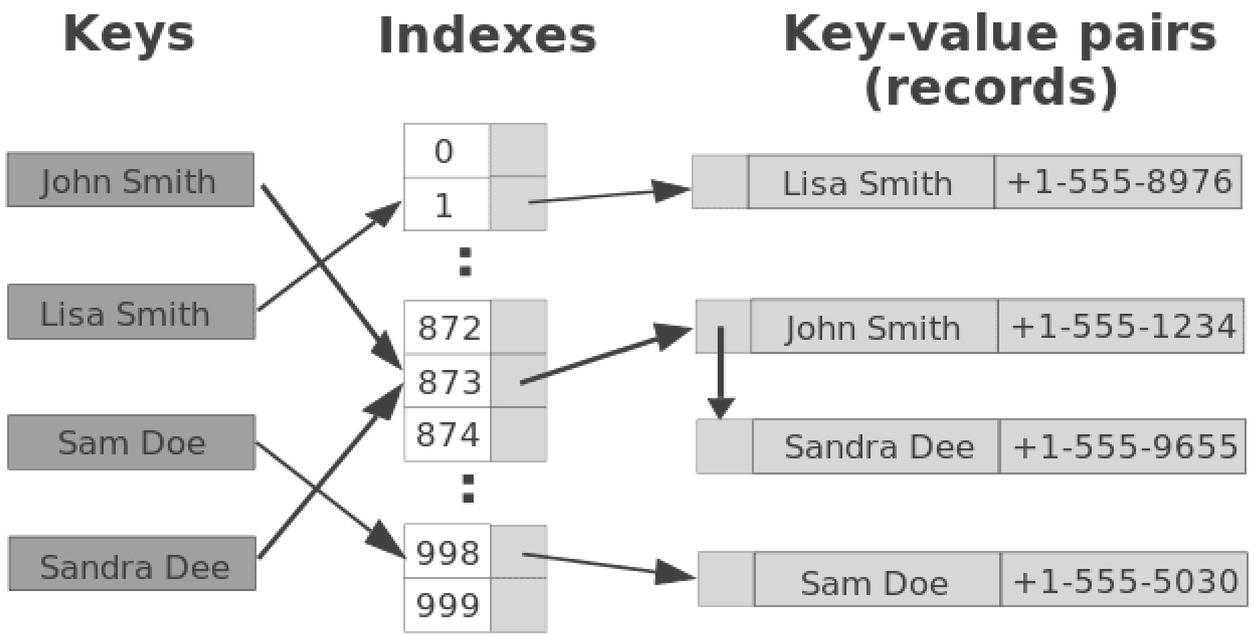
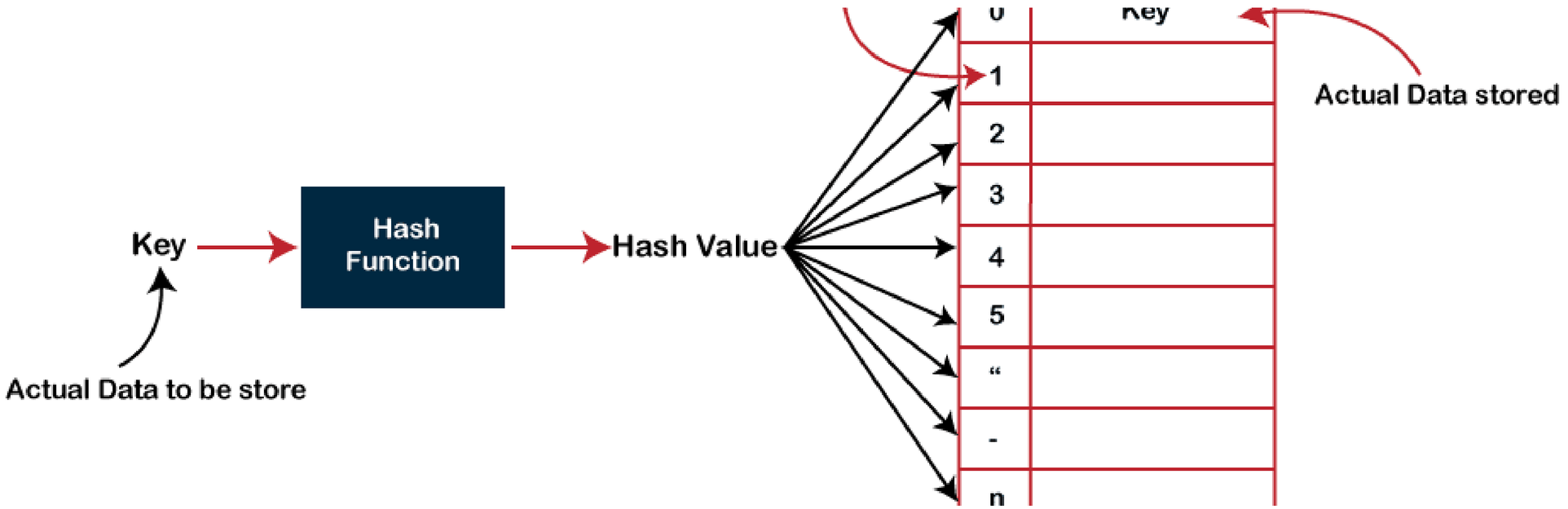
Хеш-таблиці з відкритою адресацією

Що це таке?

Хеш-таблиця - це структура даних, що реалізує інтерфейс асоціативного масиву, в якому можна зберігати пари Ключ:Значення.





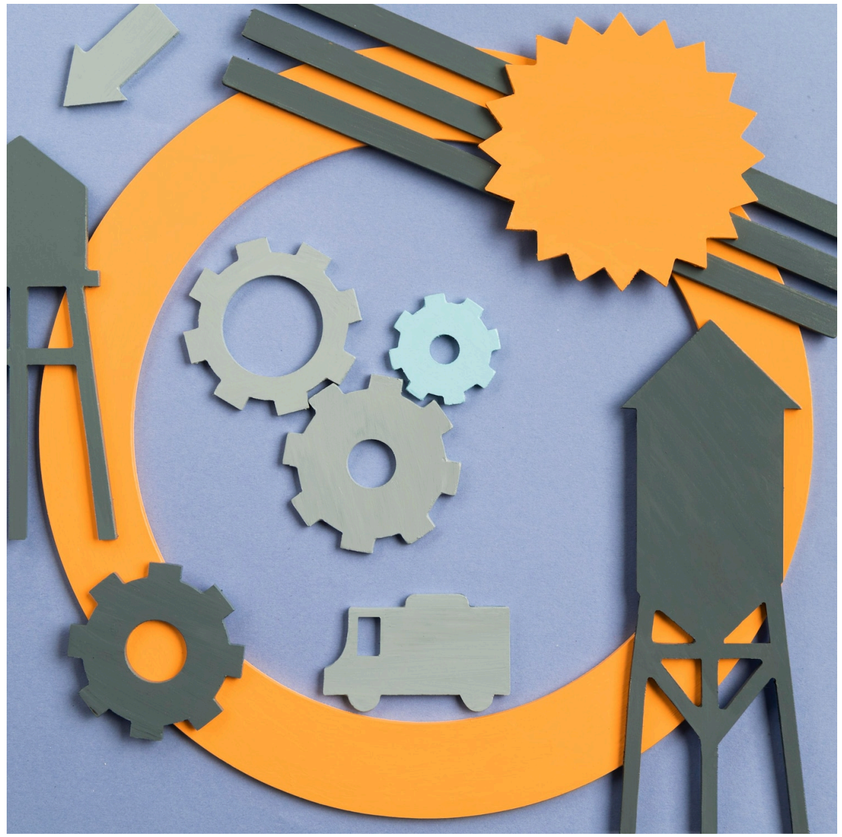
Хеш-функція

Хеш-функція - функція, що використовує ключ для знаходження індексу, за яким дані значення будуть занесені в таблицю на позицію, що дорівнює індексу.

Вирішення колізій

Колізія - ситуація, при якій двом різним значенням ключа відповідає одне хеш-значення, що його отримали, пропустивши ключі через хеш-функцію.

При ініціалізації хеш-таблиці з відкритою адресацією існує декілька основних способів вирішення колізій:
1) Лінійне зондування
2) Квадратичне зондування;
3) Подвійне хешування.



Лінійне зондування

Спосіб вирішення колізій лінійним зондуванням є найпростішим вирішенням колізій у хеш-таблиці з відкритою адресацією.
Сенс цієї схеми полягає у визначенні константи, яку ми будемо додавати до хеш-значення, перед цим помноживши на значення і, що дорівнює номеру елементу, який нам попадається з цим хеш-значенням.
Тобто нове значення індексу і-того елементу буде дорівнювати (mod N - остача при діленні на розмірність хеш-таблиці):

$$(hash(x) + ik) \bmod N.$$

Квадратичне зондування

Квадратичне зондування відрізняється від лінійного лише визначенням константи, яка визначає інтервал між елементами з однаковим значенням хеш-функції. Ця константа виражається поліномом.
У найпростішому випадку можна взяти (hash(x) + i^2)modN.

Подвійне хешування

В цьому способі при утворенні колізії ми використовуємо нову, другу хеш-функцію, яка залежить від першої та ще якоїсь. У своїй роботі я використовую дві популярні 32-бітові хеш-функції murmurOAAT та FNV, що приймають рядок як ключ та за допомогою бітових операцій над кожною буквою утворюють хеш-значення, яке в майбутньому проходить через операцію m[illegible]

```python
def murmurOAAT(key : str, h : int) -> int:
    for c in key:
        h ^= ord(c)
        h = (h * 0x5bd1e995) & 0xFFFFFFFF
        h ^= h >> 15
    return h
```

```python
def FNV(key : str, h : int) -> int:
    h ^= 2166136261

    for byte in key.encode('utf-8'):
        h ^= byte
        h = (h * 16777619) & 0xFFFFFFFF

    return h
```

Який із трьох способів є найкращим?

Лінійне та квадратичне зондування легші в імплементації, але в них є великий мінус, особливо в першому способі вирішення колізій - можливо велике кластерування даних, що буде сповільнювати роботу програми.
Подвійне хешування ж не буде змінювати в швидкості програми.

Часова складність

Часова складність хеш-таблиці "базових" операцій структури - видалення, вставка та пошук елементу мають часову складність O(1). Тобто всі операції мають виконуватись за фіксований час, який залежить від швидкості функцій хешування. В моєму випадку функції виконують прості бінарні операції, що є швидкими у підрахунку.

За умови, якщо потрібно перебудувати таблицю (у більшості випадків найкращий час для перебудови таблиці в більшу - таблиця заповнена більш ніж на половину), часова складність операцій може досягати O(n), що означає, що час роботи лінійно залежить від кількості елементів таблиці.

Time complexity in big O notation

Algorithm	Average	Worst case
Space	O(*n*)[1]	O(*n*)
Search	O(1)	O(*n*)
Insert	O(1)	O(*n*)
Delete	O(1)	O(*n*)

Часова складність

В моїй роботі resize() відбувається тільки при наповненні хеш-таблиці більше, ніж на половину. Тобто тільки в цьому випадку часова складність буде залежати від розмірності таблиці.

```
def resize(self):
  if self.actUsedSlots >= (0.5 * self.size):
    self.resizeCount *= 2
    self.size = self.size * self.resizeCount

    self.usedSlots = 0
    self.actUsedSlots = 0

    newTable = [None] * self.size

    for i in range(int(self.size/self.resizeCount)):
      if self.table[i] is not None and self.table[i].isDeleted != 1:
        newItem = self.table[i]
        index = self.hash(newItem.itemName)
        attempt = 0

        while True:
          if newTable[index] is None:
            newTable[index] = newItem
            self.actUsedSlots += 1
            self.usedSlots += 1
            break
          attempt += 1
          index = self.secondaryHash(newItem.itemName, attempt)

    self.table = newTable
```

Дякую!