

Практика по python 11

Хэш-функции. Словари и множества

Hash-таблица

Структура данных, которая реализует АТД ассоциативный массив

- Необходимо определить функцию хеширования для ключей
- Для хороших функций в среднем на операции вставки/поиска/удаления

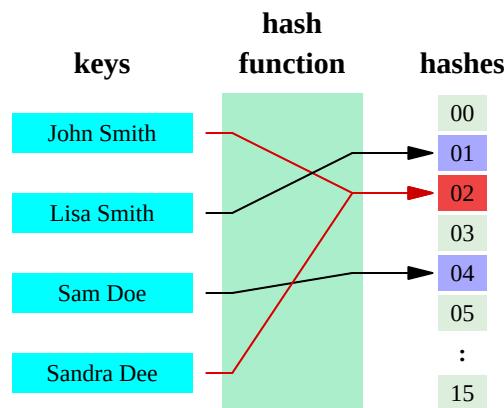
В Python:

- `hash(x)` – для вычисления хеша
- `dict`, `set` – хеш-таблицы

Hash-функция

Функция преобразования большого (потенциально, бесконечного) множества ключей в конечное (и маленькое) множество hash-значений.

- Значения функции можно использовать в качестве индекса в массиве
- Hash-функция не инъективна
- **Коллизия** — для разных значений получили одинаковое значение хеша



Hash-функция

Хорошая hash-функция

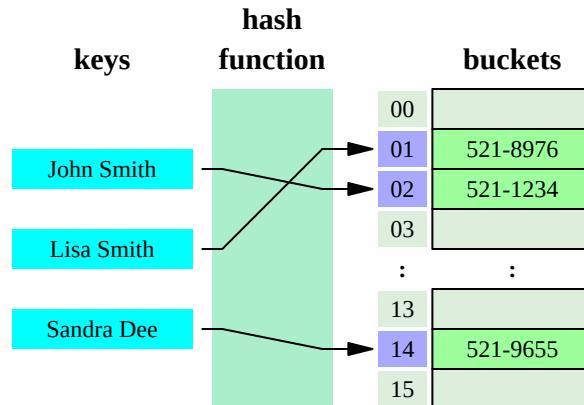
- Минимизирует число коллизий
- Распределяет значения хешей равномерно
- Не отображает связь между ключами

На практике:

- Должна быть простой с вычислительной точки зрения
- Обычно возвращает целое число (hash-таблица сама «загоняет» в нужный интервал)

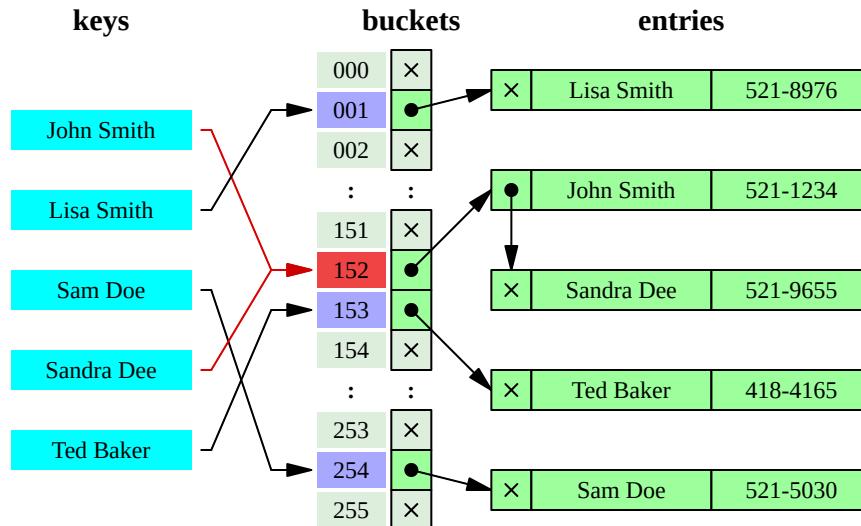
Hash-таблица

- Содержит некоторый массив H , размером N
- Выполнение всех операций начинается с вычисления хеша ключа: $\text{hash}(\text{key}) = i$
- i используется в качестве индекса в H : $H[i \% N] = \text{value}$



Separate chaining

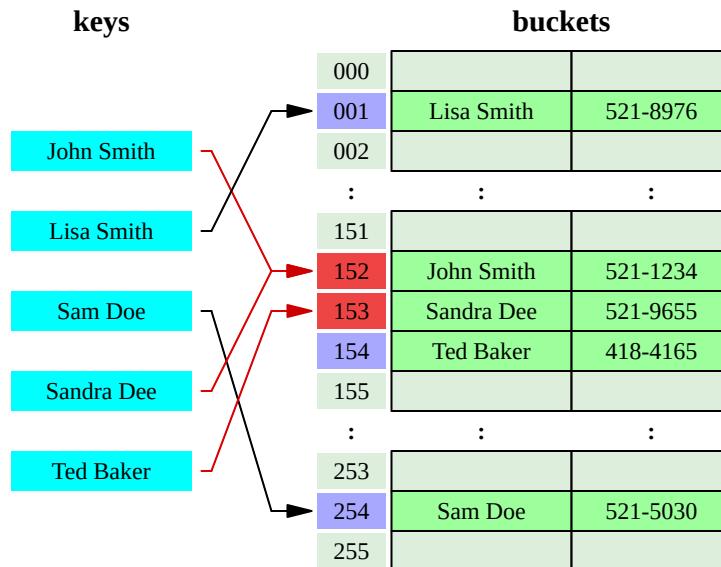
- В каждой ячейке находится связный список
- Проходимся по списку пока не найдём нужный элемент
- Вместо списка можно использовать динамический массив или сбалансированное дерево поиска



Open addressing

Будем всё хранить "честно", а если произошла коллизия, то двигаемся

- Если ячейка занята, то ищем "следующую" свободную
- Последовательность проб — правило перехода в следующую ячейку
- Пустая ячейка — конец цепочки, кладем новое значение
- Для удаления требуется отдельный флаг: при вставке ячейка занимается, при поиске идут дальше



Последовательность проб

Линейное пробирование

- Зададим шаг k (обычно 1)
- i элемент последовательности — $(\text{hash}(\text{key}) + k*i) \% N$

Последовательность проб

Квадратичное пробирование

- Интервал между пробами увеличивается на константу
- i элемент последовательности — $(\text{hash}(\text{key}) + k*i + l*(i**2)) \% N$
- Если $N=2p$, то для обхода всех ячеек полезно использовать **треугольные числа** ($k=l=1/2$)
 $\text{hash}(\text{key})+1, \text{hash}(\text{key})+3, \text{hash}(\text{key})+6, \dots$

Последовательность проб

Двойное хеширование

- Размер шага вычисляется второй хеш-функцией (важно правильно выбрать!)
- Как линейное, но для разных ключей разный шаг

Load factor

Пусть n — число элементов в таблице, а N — общий размер

Load factor: $L = n/N$

- Для таблицы на списках — поддерживаем $< 1^*$
- Для таблицы с открытой адресацией — поддерживаем $< 0.8^*$

*подбираем под каждый кейс и реализацию

Увеличиваем размер

Когда коэффициент заполнения становится "слишком большим" — необходимо выделить новое место

- Скопировать все элементы разом нельзя — надо пересчитывать хеш
- Создаем таблицу на $2N$ элементов
- Момент "слишком большой" таблицы определяется Load factor-ом

dict

Словари бывают не только пользовательские: области видимости, атрибуты объектов, ...

Python dict:

- Хеш-таблица с открытой адресацией
- Пробирование на основе псевдослучайных чисел
- Начинает с размера 8, увеличивается в 2 раза при $L > \frac{2}{3}$ и уменьшается при $L < \frac{1}{3}$
- Хеш — 32 или 64 битное число
 - Для чисел — само число по модулю $2^{61} - 1$
 - Строки, байты — SipHash

Оптимизированы для работы с небольшим числом элементов: Compact layout, shared key, strings interning

dict

```
typedef struct {
    PyObject_HEAD
    /* Number of items in the dictionary */
    Py_ssize_t ma_used;

    /* This is a private field for CPython's internal use.
     * Bits 0-7 are for dict watchers.
     * Bits 8-11 are for the watched mutation counter (used by tier2 optimization)
     * Bits 12-31 are currently unused
     * Bits 32-63 are a unique id in the free threading build (used for per-thread refcounting)
     */
    uint64_t _ma_watcher_tag;
    PyDictKeysObject *ma_keys;

    /* If ma_values is NULL, the table is "combined": keys and values
     * are stored in ma_keys.

     * If ma_values is not NULL, the table is split:
     * keys are stored in ma_keys and values are stored in ma_values */
    PyDictValues *ma_values;
} PyDictObject;
```

set

Немного отличается от `dict`

- Максимальный коэффициент загрузки 0.6
- При размере до 50,000 элементов увеличение происходит в 4 раза

set

```
typedef struct {
    PyObject_HEAD

    Py_ssize_t fill;           /* Number active and dummy entries*/
    Py_ssize_t used;          /* Number active entries */

    /* The table contains mask + 1 slots, and that's a power of 2.
     * We store the mask instead of the size because the mask is more
     * frequently needed.
     */
    Py_ssize_t mask;

    /* The table points to a fixed-size smalltable for small tables
     * or to additional malloc'ed memory for bigger tables.
     * The table pointer is never NULL which saves us from repeated
     * runtime null-tests.
     */
    setentry *table;
    Py_hash_t hash;            /* Only used by frozenset objects */
    Py_ssize_t finger;         /* Search finger for pop() */

    setentry smalltable[PySet_MINSIZE];
    PyObject *weakreflist;     /* List of weak references */
} PySetObject;
```