Оглавление

Функции хеширования

Хеширование делением

- Числовое значение ключа делится на размер хеш-таблицы: $hash(k) = k \mod M$
- Хорошо работает в ситуации, когда ключи взяты из равномерного распределения
- Такое способ не очень быстр (потому что деление - "медленная" операция)

Проблемы выбора M

- M не должно быть степенью двойки. Иначе для $M = 2^p$ хеш будет просто p младших битов ключа (что допустимо только в случае, если несколько последних знаков числа распраделены равномерно)
- Хороши простые числа, не очень близкие к степени 2.
- Идущие подряд значения ключей "порождают" идущие подряд значения хешей. Это может быть как плюсом, так и минусом алгоритма.

Сравните с 10й системой счисления при делении на 10^k: нет неожиданности, что в остатке будет k наименее значащих цифр изначального числа. То же верно и для систем с основанием 2.

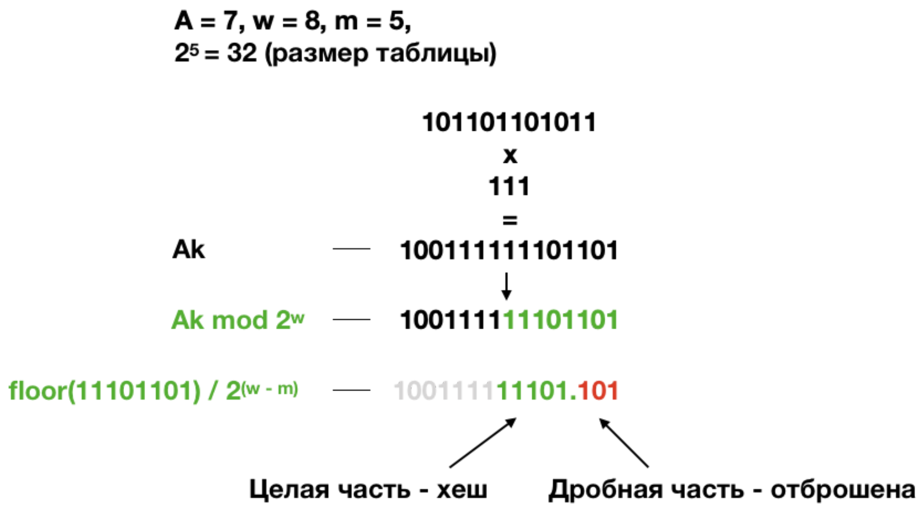
1010011011	2^1
1010011011	2^2
1010011011	2^3
1010011011	2^4
1010011011	2^5
1010011011	2^6
1010011011	2^7
1010011011	2^8

Хеширование при помощи умножения

- $hash(k) = \lfloor (Ak \mod W) \ / \ (W/M) \rfloor$
- A взаимно простое с W
- на практике удобно $W = 2^w, M = 2^m$, w - размер машинного слова (16, 32, 64, как правило) Тогда хеш-функция превращается в $hash(k) = \lfloor (Ak \mod 2^w) \ / \ 2^{w-m} \rfloor$, которую можно записать как простую функцию со "сдвигом":

```
hash(key, A, M):
    return A * x >> (w - M)
```

- $A \cdot k \mod 2^w$ - аналогично хешированию делением, но A вносит "*возмущение*" в ключ k
- Операция "сдвига" раскладывается на две операции:
  - собственно "деление" на $2^{(w-m)}$
  - удаление дробной части происходит естественным образом за счет "сдвига"



Такой подход *очень быстр* из-за простоты базовых операций (выполняющихся быстрее деления).

Более того, подобная функция близка к универсальной, если A - случайное нечетное число из $\{1, \ldots, 2^{w-1}\}$. Про это мы еще поговорим.

Вопрос: почему нечетное?

Хеш-функции для строк и последовательностей

- хеш должен зависеть от каждого символа
- *и* хеш должен зависеть от каждого символа по-разному!
- строки с одинаковыми символами в разном порядке, желательно, должны хешироваться по-разному

Пример простой хеш-функции для последовательностей: хеш-функция Пирсона.

```
PearsonHash(S, Table):
   h := 0
   for (i = 0; i < len(S); i++):
       h := Table[h xor S[i]]  // Выбор значения из таблицы Table
   return h
```

- Что такое Table? Это *перемешанный* список чисел 0..255. Он может быть как случайно перемешан, так и подобран специальным образом, чтобы обеспечить *идеальное* хеширование для определенного набора данных (см. далее)

Свойства хеш-функции Пирсона:

- простой код и очень быстрая работа
- две строки, различающиеся на один символ, *никогда* не создадут коллизии

- Модифицировав работу с `Table`, моджно применять для архитектур с большим, чем 8, машинным словом

```
PearsonHash64(S, Table):
   for i in 0..7:
       h = Table[(S[0] + i) % 256]  // +i для учета сдвига
       for each item in S:
           h = Table[h xor item]
       hh[i] = h
   return concatenated hh
```

ARX (add-rotate-xor)

Быстрые и используемые на практике наборы функций для хеширования последовательностей.

Базируются на следующем принципе:

- Сложение двух чисел по моудлю
- Сдвиг элементов
- XOR

Варианты ARX-алгоритма SipHash используются во многих языках, в том числе, python3.4+, Ruby, Perl.

[в начало (https://otus.ru/media-private/09/df/universal_hashing-31272-09df9e.html?hash=GZmHA7wY0m4ZZq1V3etV0g&expires=1593465693#index)](https://otus.ru/media-private/09/df/universal_hashing-31272-09df9e.html?hash=GZmHA7wY0m4ZZq1V3etV0g&expires=1593465693#index)

Схемы адресации

Квадратичный пробинг

- $hash(k, i) = (hash'(k) + \mathrm{c}_1 i + \mathrm{c}_2 i^2) \mod M$.
- Лучше линейного, но нужно подбирать $\mathrm{c}_1, \mathrm{c}_2, M$.

Несколько популярных вариантов выбора констант

- $hash(k) = (hash'(k) + i^2) \mod M$, где $c_1 = 0, c_2 = 1, M - \text{простое} > 3$, фактор заполнения $\alpha < 1/2$

- $hash(k) = (hash'(k) + (i + i^2) / 2) \mod M$, где $c_1 = c_2 = 1/2, M = 2^k$
- $hash(k) = (hash'(k) + -1^i \cdot i^2) \mod M$, где $M \equiv 3\ mod\ 4$

Почему $\alpha < 1/2$? Пусть есть сдивги x и y, указывающие на одну локацию, но $x \neq y$, и $0 \leq x, y \leq M/2$.

$$hash(k) + x^2 \equiv hash(k) + y^2 modM$$
$$x^2 \equiv y^2 modM$$
$$x^2 - y^2 \equiv 0\ modM$$
$$(x - y) \cdot (x + y) \equiv 0\ modM$$

$x - y \neq 0, x + y \neq 0$ - значит, существует M/2 различных мест для записи.

Чем квадратичный пробинг лучше линейного?

Равномерность распределения хешей по таблице зависит от количества возможных вариантов обхода при пробинге.

- Максимально $M!$ вариантов обхода, столько требуется для "честного" равномерного хеширования
- Для линейного пробинга - всего M вариантов, по числу стартовых точек
- Для квадратичного пробинга так же всего M, но сами последовательности пробинга распределены более *равномерно*

Двойное хеширование

Идея: использовать 2 хеш-функции, чтобы получить M^2 последовательностей пробинга.

Формально, это

$$hash(k, i) = (hash_1(k) + i \cdot hash_2(k)) \mod M$$

Для обхода *всех* ячеек при пробинге $hash_2(k)$ должна всегда возвращать взаимно простое с M число. Простой способ сделать это:

- Выбирать размер хеш-таблицы как степень 2: $M = 2^p$
- $hash_2$ всегда возвращает нечетное число
- К сожалению, на практике, если размер таблицы $\neq 2^p$, выбрать функцию может быть тяжело

Тем не менее, C# HashTable использует двойное хеширование (https://referencesource.microsoft.com/#mscorlib/system/collections/hashtable.cs)

Количество попыток при пробинге

- В случае, если распределение ключей близко к равномерному (хорошая хеш-функция и пробинг)
- фактор заполненности таблицы равен $\alpha < 1$,

1

Универсальное хеширование

Универсальное хеширование *случайно* выбирает хеш-функцию из семейтсва хеш-функций, чтобы осложнить жизнь злоумышленнику (или просто избежать ситуации с "плохими" данными).

$\mathcal{H}$ - семейство хеш-функций, отображающих ключи во множество $\{0..M-1\}$. При этом, для пары ключей k, l количество хеш-функций, для которых $hash(k) = hash(l)$, не более $|\mathcal{H}|/M$

- то есть шанс коллизии для случайно выбранной хеш-функции и двух случайных ключей не более $1/M$
- это позволяет получить близкое к идеальному поведение

Построение класса универсальных хеш-функций

- Выберем p достаточно большое, чтобы любой ключ k попадал в диапазон от 0 до $p-1$
- $\mathbb{Z}_p = \{0, 1..p-1\}$, $\mathbb{Z}_p^* = \{1, 2..p-1\}$ - множества чисел
- $hash_{ab}(k) = ((ak+b) \mod p) \mod M$ - хеш-функция
- $\mathcal{H} = \{hash_{ab} : a \in \mathbb{Z}_p \wedge b \in \mathbb{Z}_p^*\}$ - всего $p \cdot (p-1)$ функций

Как получилось, что это универсальные хеш-функции

$r = (ak+b) \mod p$
$s = (al+b) \mod p$

$r - s \equiv a(k-l) (\mod p)$ - так как p - простое, и $(k-l), a$ не равны 0 по модулю p.

Поэтому коллизий $(ak+b) \mod p$ не возникает. Более того, все пары (r, s) будут различными для каждой пары (a, b).

Осталось проверить, что коллизий не возникает и при взятии $\mod M$. Поскольку разные r, s равновероятны, вероятность коллизии $k и l$ равна вероятности того, что $r \equiv s (mod\ M)$. Ее можно вычислить так:

$$\lceil p/m \rceil - 1 \leq ((p+m-1)/m) - 1 = (p-1)/m$$

- количество значений для r таких, что $s \neq r$ и $s \equiv r(mod\ M)$. Поскольку есть $(p-1)$ возможных s, вероятность будет

$$\frac{((p-1)/m)}{(p-1)} = 1/m,$$

что удовлетворяет требованию к универсальной хеш-функции.

[в начало (https://otus.ru/media-private/09/df/universal_hashing-31272-09df9e.html?hash=GZmHA7wY0m4ZZq1V3etV0g&expires=1593465693#index)](https://otus.ru/media-private/09/df/universal_hashing-31272-09df9e.html?hash=GZmHA7wY0m4ZZq1V3etV0g&expires=1593465693#index)

Пример кода на python (нерабочий, но поясняющий идею создания семейства функций)

In [ ]:
```python
# Так

def generate_hash_fh(p):
    a, b = random.randint(0, p-1), random.randint(1, p-1)
    return partial(universal_hash, a, b)


def universal_hash(a, b, k):
    return ((a * k + b) % p) % m  # p простое, 2^64
```

In [ ]:
```
# Или так

class UnivHash:
    def __init__(self, a, b):
        pass

    def __call__(self, key):
        pass
```

Домашняя работа

- Для открытой адресации реализуйте двойное хеширование
- Для метода цепочек реализуйте создание универсальной хеш-функции

(Также см. предыдущую домашнюю работу)

[в начало](https://otus.ru/media-private/09/df/universal_hashing-31272-09df9e.html?hash=GZmHA7wY0m4ZZq1V3etV0g&expires=1593465693#index) (https://otus.ru/media-private/09/df/universal_hashing-31272-09df9e.html?hash=GZmHA7wY0m4ZZq1V3etV0g&expires=1593465693#index)

Ссылки

- Двойное хеширование: C# HashTable использует двойное хеширование (https://referencesource.microsoft.com/#mscorlib/system/collections/hashtable.cs)
- Код pyhash.h: выбор хеш-функции FNV или SipHash-2-4 для использования (https://github.com/python/cpython/blob/master/Include/pyhash.h)

в начало (https://otus.ru/media-private/09/df/universal_hashing-31272-09df9e.html?hash=GZmHA7wY0m4ZZq1V3etV0g&expires=1593465693#index)