

# ОСНОВЫ ПРОГРАММНОГО КОНСТРУИРОВАНИЯ

Лекция № 12  
20 ноября 2017 г.

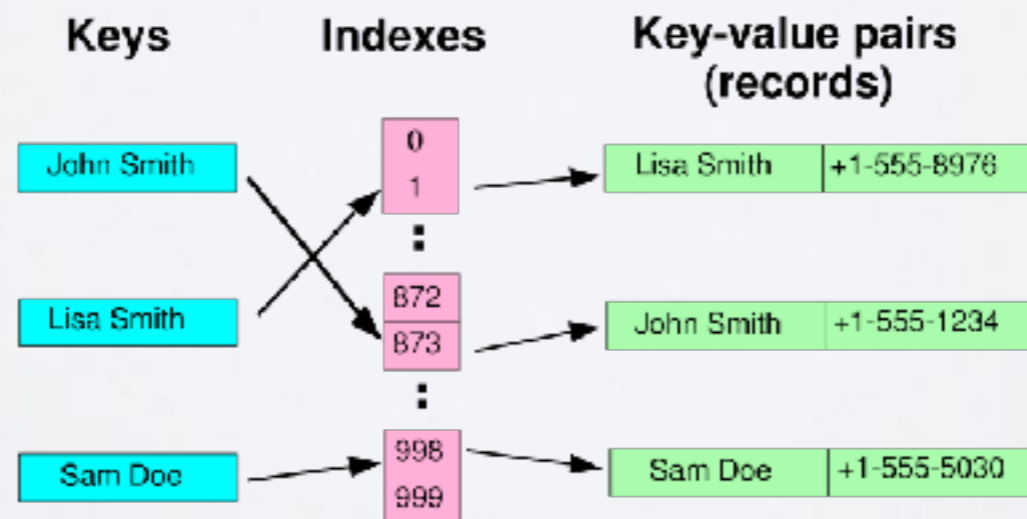




ОЧЕНЬ БЫСТРЫЙ ПОИСК

# ХЕШ-ТАБЛИЦА

- *to hash* (англ.) – мелко нарубить и перемешать.
- Каждому ключу ставится в соответствие целочисленный код (хеш-код):  $h(\text{key}) \rightarrow n$ .
- Пары (ключ, значение) помещаются в массив, индексированный по хеш-коду.
- Поиск: вычисляем хеш-код и отправляемся прямо в нужную точку массива.



# ХЕШ-ФУНКЦИЯ

- Для массива длиной  $N$  должна выдавать значения  $0 \leq h_N(k) < N$ .

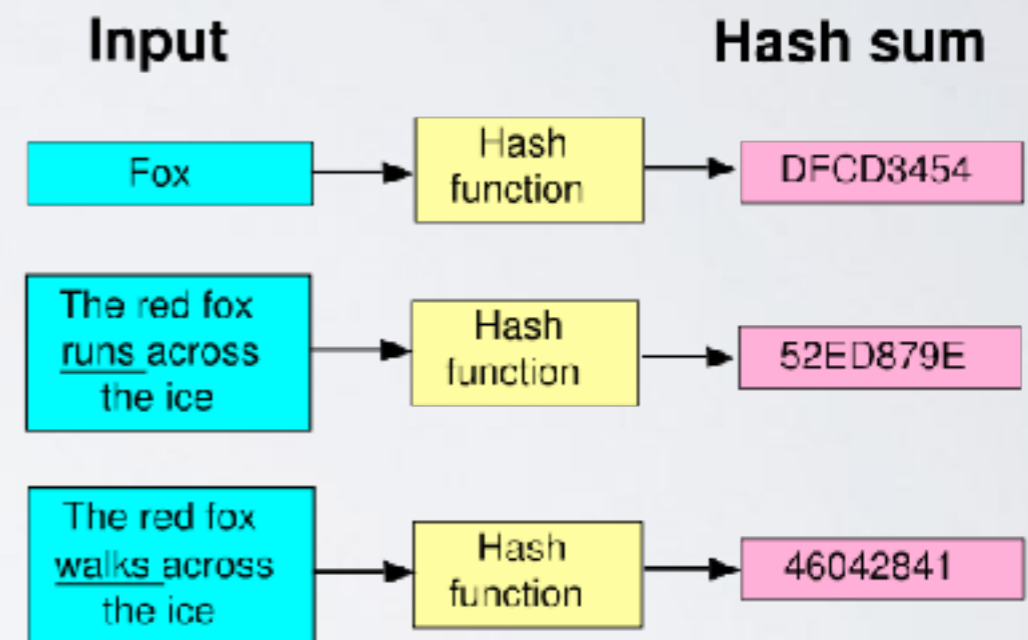
- Должна быть детерминистичной.

- Должна вычисляться быстро.

- Отсутствие кластеризации:

- Очень желательно  $h_N(k_1) \neq h_N(k_2)$  для  $k_1 \neq k_2$ .

- Малое изменение ключа (1 бит) должно давать большое изменение хеш-кода.



# ИНТЕРФЕЙС ХЕШ-ТАБЛИЦЫ

- `get(key) → data.`
- `put(key, data).`
- `has_key(key).`
- `all_keys().` // список не отсортирован

# РАЗМЕЩЕНИЕ В МАССИВЕ

- Метод деления:

$$h_N(k) = h(k) \bmod N.$$

- Метод умножения:

$$h_N(k) = [N \cdot \{h(k) \cdot A\}] \text{ для } 0 < A < 1.$$

Например,  $A = (\text{sqrt}(5) - 1) / 2 \approx 0,6180339887\dots$

# ТРИВИАЛЬНЫЕ ХЕШ-ФУНКЦИИ

- Если  $k$  – целое число,  $h(k) = k$ .
- Если  $k \in [0, 1)$ , то  $h_N(k) = [N \cdot k]$ .
- Если  $k$  – произвольное число с плавающей точкой, можно взять мантиссу, приведенную к  $[0, 1)$ .

# ОБЩИЙ СЛУЧАЙ: ЦЕПОЧКА БАЙТ

Хорошая ли хеш-функция?

$$h(\mathbf{b}) = \sum b_i.$$

**Нет.**

$$h(\text{"abc"}) = h(\text{"bac"}) = h(\text{"aad"})$$



# ХЕШ-ФУНКЦИЯ ДЖЕНКИНСА

```
uint32_t jenkins_one_at_a_time_hash(  
    unsigned char *key, size_t key_len) {  
  
    uint32_t hash = 0;  
    for (size_t i = 0; i < key_len; i++) {  
        hash += key[i];  
        hash += (hash << 10);  
        hash ^= (hash >> 6);  
    }  
    hash += (hash << 3);  
    hash ^= (hash >> 11);  
    hash += (hash << 15);  
    return hash;  
}
```

# X.-Φ. ΦΑΥΛΕΡΑ-ΗΟΛΛΑ-ΒΟ

```
uint32_t fnv_32_buf(void *buf, size_t len) {
    // start of buffer
    unsigned char *bp = (unsigned char *)buf;
    // beyond end of buffer
    unsigned char *be = bp + len;
    uint32_t hval = 0x811c9dc5;

    while (bp < be) {
        hval += (hval<<1) + (hval<<4) + (hval<<7) +
                (hval<<8) + (hval<<24);

        /* xor the bottom with the current octet */
        hval ^= (uint32_t)*bp++;
    }

    /* return our new hash value */
    return hval;
}
```

# ХЕШ-ФУНКЦИЯ ДЛЯ НАБОРА ЭЛЕМЕНТОВ

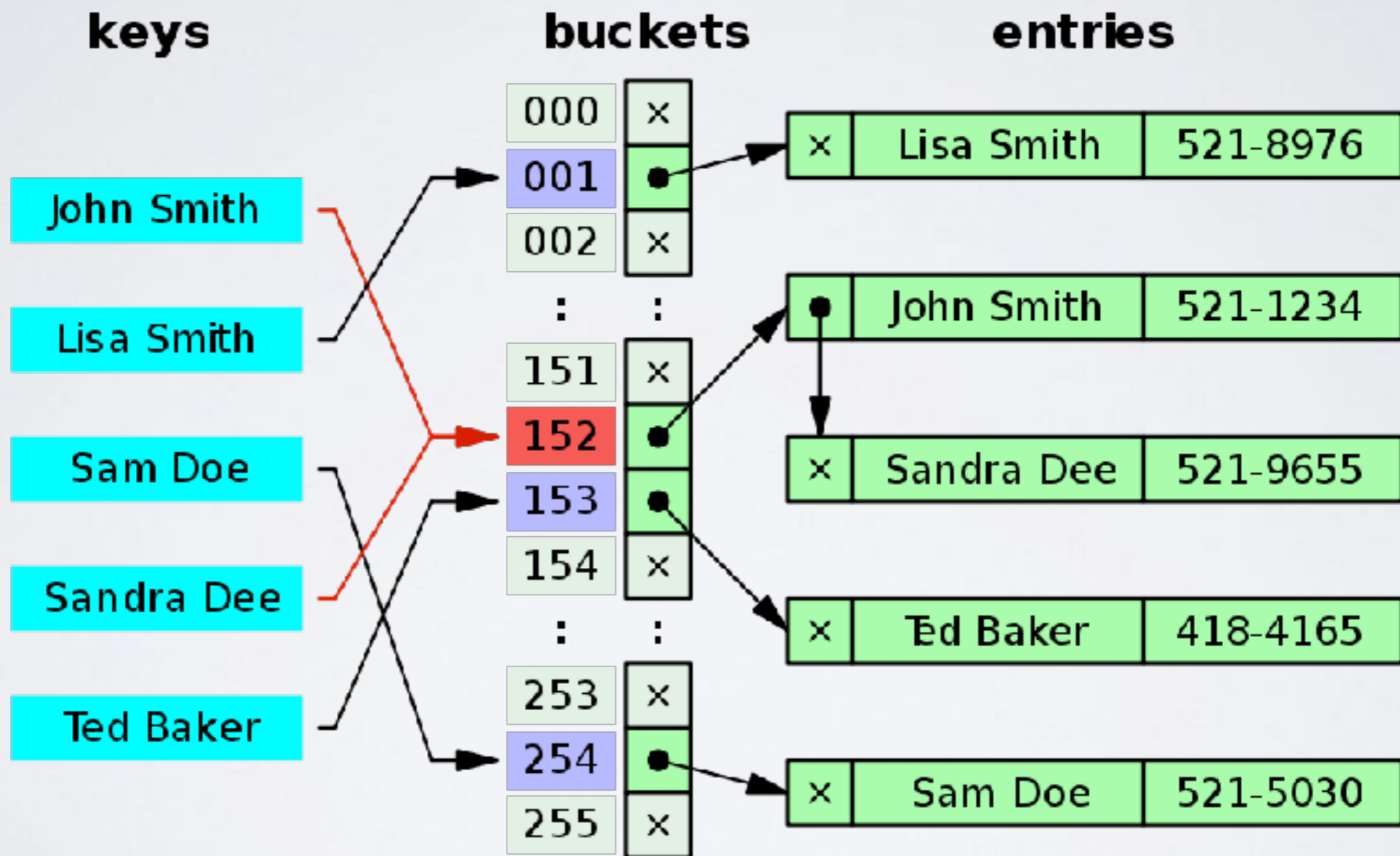
Не отличная, но вполне пригодная хеш-функция для набора элементов (массив, структура и др.):

$$h(e_1, e_2, \dots, e_n) = h(e_1) + 31 \cdot (h(e_2) + 31 \cdot (\dots 31 \cdot h(e_n)))$$

# КОНФЛИКТЫ

- Конфликт – ситуация, когда  $h_N(k_1) = h_N(k_2)$  для  $k_1 \neq k_2$ .
- Способы обхода проблемы:
  - Идеальное хеширование.
  - Усложнение базовой структуры данных.
  - Размещение конфликтующего ключа в том же массиве, но в другом месте.
  - *Совсем хитрые способы.*

# ДОБАВЛЯЕМ СПИСКИ КОЛЛИЗИЙ



# ДИНАМИЧЕСКИЙ МАССИВ СПИСКОВ

- В ячейках массива хранятся указатели на ГОЛОВЫ СПИСКОВ.
- Двухфазный поиск:
  - Вычисление хеш-кода.
  - Путешествие по списку.
- Удобно вставлять в голову.
- Альтернативы:
  - Динамический массив<sup>2</sup>.
  - Сбалансированное дерево.

# ОТКРЫТАЯ АДРЕСАЦИЯ

- Вставка ключа: если возникает конфликт, начинаем перебирать другие ячейки, пока не найдем свободную:
  - $h_N(k) \rightarrow \langle h_N(k, 0), h_N(k, 1), \dots, h_N(k, N-1) \rangle$ .
- Линейное исследование:  $h_N(k, m) = (h_N(k) + m) \bmod N$ .
  - *Первичная кластеризация* – длинные последовательности занятых ячеек.

# КВАДРАТИЧНОЕ ИССЛЕДОВАНИЕ

- $h_N(k, m) = (h_N(k) + c_1 \cdot m + c_2 \cdot m^2) \bmod N$ .
- Требуется специального выбора  $c_1$ ,  $c_2$  и  $m$ .
- *Вторичная кластеризация*: при конфликте между  $k_1$  и  $k_2$  последовательности  $h_N(k_1, m)$  и  $h_N(k_2, m)$  совпадают.



# ДВОЙНОЕ ХЕШИРОВАНИЕ

- $h_N(k, m) = (h_N(k) + m \cdot h'_N(k)) \bmod N$ , где  $h'_N(k)$  – вторая хеш-функция.
- Значения  $h'_N(k)$  должны быть взаимно просты с  $N$ , чтобы последовательность перебирала все ячейки таблицы.
  - $N = 2^j \rightarrow h'_N(k)$  возвращает нечетные значения.
- Нет ни первичной, ни вторичной кластеризации.
- Лучший способ использования открытой адресации!

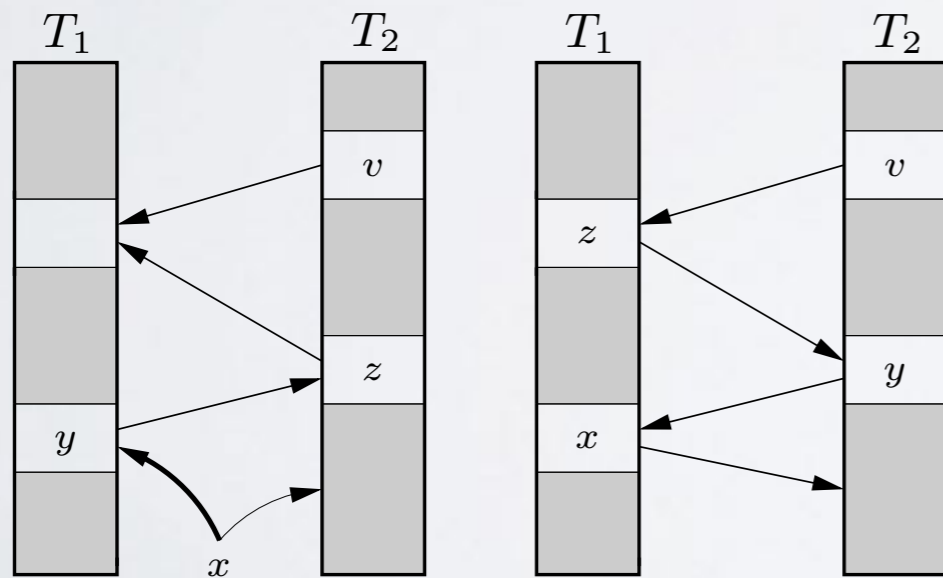
# КУКУШИННОЕ ХЕШИРОВАНИЕ (CUCKOO HASHING)

- Две отдельные таблицы размера  $N$ :  $T_1$  и  $T_2$  с хеш-функциями  $h_{1,N}$  и  $h_{2,N}$ .
- Любой ключ  $k$  находится либо в ячейке  $T_1[h_{1,N}(k)]$ , либо в  $T_2[h_{2,N}(k)]$ .
- Поиск за  $O(1)$  гарантирован!

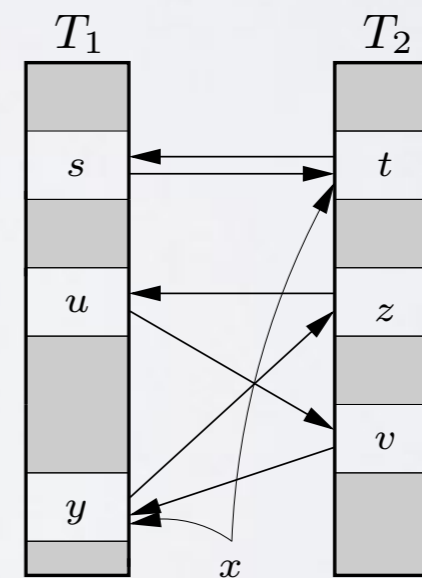
# КУКУШКА ВСТАВЛЯЕТ

- Вставка в  $T_1[h_{1,N}(k)]$ .
- Если ячейка занята, то находящийся в ней ключ перемещается на альтернативную позицию в  $T_2$ , вытесняя находившийся там ключ и т.д.

- Если процесс зацикливается, выбираются новые хеш-функции, и таблицы перестраиваются заново.



Вставка  $x$



«Не лезет»

# ХЕШ VS. ДЕРЕВО

- **Дерево:**

- Нужна операция сравнения:  $k_1 \preceq k_2$ .
- Операции за  $O(\log N)$ .
- Упорядоченная структура данных.

- **Хеш-таблица:**

- Нужна хеш-функция  $\text{hash}(k)$  и операция сравнения  $k_1 = k_2$ .
- Операции где-то между  $O(1)$  и  $O(n)$ .
- Неупорядоченная структура данных.

# КРИПТОГРАФИЧЕСКИЕ ХЕШ-ФУНКЦИИ

- Создают «отпечаток» данных, имеющий фиксированный размер.
- Для  $H = h(k)$  очень трудно найти  $k = h^{-1}(H)$ .
- Имея  $H = h(k)$ , очень трудно организовать коллизию (найти такое  $k_2$ , чтобы  $H = h(k) = h(k_2)$ ).
- MD5 (128 бит), SHA-1 (160 бит), SHA-2 (224-512 бит), SHA-3/Кескак (произвольная длина), GOST и т.д.

# CHECKSUM

Контрольная сумма используется для проверки целостности данных, передаваемых по *незащищенным* каналам. При этом контрольная сумма передается по *защищенному* каналу.

Пользователь может проверить корректность полученных данных, вычислив контрольную сумму и сравнив ее с опубликованной.

```
$ curl http://releases.ubuntu.com/17.10/MD5SUMS
5eb0631f28384665b57bceffe69c29b0 *ubuntu-17.10-desktop-amd64.iso
be2ab38bb169ed9dddc6cd142a8b9d8b *ubuntu-17.10-server-amd64.img
be2ab38bb169ed9dddc6cd142a8b9d8b *ubuntu-17.10-server-amd64.iso
429e909e8e7a060f1149013689d62a94 *ubuntu-17.10-server-i386.img
429e909e8e7a060f1149013689d62a94 *ubuntu-17.10-server-i386.iso
```

# АДРЕСАЦИЯ ПО СОДЕРЖИМОМУ

- Вместо «имени» объекта используем значение криптографической хеш-функции от содержимого.
- Дубликаты отсутствуют как факт!
- Используется в BitTorrent, в системе управления версиями файлов Git и др.

# ФИЛЬТР БЛУМА

- Вероятностная структура данных с двумя операциями:
  - Добавить элемент.
  - Проверить присутствие элемента, причем возможен ложноположительный результат.
- Используется для быстрого *отфильтровывания* запросов к более медленному хранилищу.



# ФИЛЬТР БЛУМА: РЕАЛИЗАЦИЯ

- Структура данных состоит из:
  - битовый массив **bs** длины **m**,
  - **k** независимых хеш-функций  $h_1, h_2, \dots, h_k$  (область значений  $[0..m)$ ).
- Реализация операций:
  - Добавить элемент **e**:  
$$bs[h_1(e)] = bs[h_2(e)] = \dots = bs[h_k(e)] = 1$$
  - Проверка принадлежности:  
$$result = bs[h_1(e)] \ \&\& \ bs[h_2(e)] \ \&\& \ \dots \ \&\& \ bs[h_k(e)]$$

# ФИЛЬТР БЛУМА: ПРИМЕР

$$m = 18, k = 3$$

0	1	0	1	1	1	0	0	0	0	0	1	0	1	0	0	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

- Добавить  $\mathbf{x}$  ( $h_1(\mathbf{x}) = 5, h_2(\mathbf{x}) = 1, h_3(\mathbf{x}) = 13$ ).
- Добавить  $\mathbf{y}$  ( $h_1(\mathbf{y}) = 4, h_2(\mathbf{y}) = 16, h_3(\mathbf{y}) = 11$ ).
- Добавить  $\mathbf{z}$  ( $h_1(\mathbf{z}) = 11, h_2(\mathbf{z}) = 3, h_3(\mathbf{z}) = 5$ ).
- Содержит ли  $\mathbf{x}$  ( $h_1(\mathbf{x}) = 5, h_2(\mathbf{x}) = 1, h_3(\mathbf{x}) = 13$ )? Возможно.
- Содержит ли  $\mathbf{u}$  ( $h_1(\mathbf{u}) = 6, h_2(\mathbf{u}) = 11, h_3(\mathbf{u}) = 1$ )? Нет.
- Содержит ли  $\mathbf{v}$  ( $h_1(\mathbf{v}) = 11, h_2(\mathbf{v}) = 4, h_3(\mathbf{v}) = 16$ )? Возможно.

# ФИЛЬТР БЛУМА: АНАЛИЗ

- Вероятность ложного результата  $\approx (1 - e^{-kn/m})^k$ .
- Оптимальное число хеш-функций, минимизирующее количество ложных результатов:  $k = m \ln 2 / n$ .
  - В этом случае размер битового массива в зависимости от количества элементов ( $n$ ) и желаемой вероятности ложных результатов ( $p$ ):  $m = -n \ln p / (\ln 2)^2$ .
- Например, для  $p = 0,01$ ,  $m \approx 9,6n - 10$  бит на элемент.

# ФИЛЬТР БЛУМА: НАБОР ХЕШ-ФУНКЦИЙ

- Как получить  $k$  независимых хеш-функций?
  - Взять хеш-функцию большого размера  $M$  (512 бит и более) и разбить значение на кусочки размера  $M/k$ .
  - Взять две различные и составить из них линейные комбинации.
  - Взять хеш-функцию, параметризуемую начальным «зерном» (seed) (например, MurmurHash), и использовать  $k$  различных зерен.



# КОНЕЦ ДВЕНАДЦАТОЙ ЛЕКЦИИ

Очень долго можно искать черную кошку в темной комнате,  
особенно если ее там нет.