

ИВАНОВ А.Б.

# АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

В-ДЕРЕВЬЯ. ХЕШ-ТАБЛИЦЫ.

# В-ДЕРЕВЬЯ

(Bayer-McCreight, 1972)

- Данные читаются поблочно (постранично)
- Время чтения блока относительно велико

Применение:

- Файловые системы:  
NTFS (Windows), HFS, HFS+ (Mac),  
ReiserFS, XFS, Ext3FS, JFS (Linux)
- Базы данных:  
Oracle, MS SQL, DB2, PostgreSQL

## В-ДЕРЕВЬЯ

В-деревья обобщают 2-3 деревья: максимально разрешается хранить до  $M - 1$  ключа в вершине

- Во всех вершинах кроме корневой – не менее  $M/2$  ключей
- Внешние узлы хранят ключи (+данные)
- Внутренние узлы хранят ключи для обеспечения поиска

## B-ДЕРЕВЬЯ

Для поиска или вставки в дерево с  $N$  ключами требуется количество чтений между  $\log_{M-1} N$  и  $\log_{M/2} N$ .

На практике: не более 4 ( $M = 1024 \Rightarrow N = 64 \cdot 10^9$ )

Хинт: всегда держать корневую страницу загруженной в память.

# ХЕШ-ТАБЛИЦА (HASH TABLE)

Структура данных, реализующая ассоциативный массив.

Соглашение по сложности (контракт):

- ▶ Вставка:  $O(1)^*$
- ▶ Поиск по ключу:  $O(1)^*$
- ▶ Удаление по ключу:  $O(1)^*$

Память -  $O(n)$

Применение:

- ▶ Анализ потока данных (уникальные пользователи, etc)
- ▶ Сетевое оборудование (firewall, балансировщик, IPS)
- ▶ 3-SUM (vs Subset sum)
- ▶ ...

# ХЕШ-ФУНКЦИЯ

$$h : U \rightarrow \{0, \dots, m - 1\}$$

Коллизия:

$$x, y \in U : h(x) = h(y)$$

## ХЕШ-ФУНКЦИЯ

**Парадокс дней рождения.** В группе, состоящей из 23 или более человек, вероятность совпадения дней рождения (число и месяц) хотя бы у двух людей превышает 50%.

$U$  – группа людей,  $h : U \rightarrow \{1, \dots, 365\}$ .

[https://en.wikipedia.org/wiki/Birthday\\_problem](https://en.wikipedia.org/wiki/Birthday_problem)

Количество  $n$  необходимое для получения коллизий с вероятностью  $p$ :

$$n(p, m) \approx \sqrt{2m \cdot \ln \left( \frac{1}{1-p} \right)}$$

## ПРИМЕР РЕАЛИЗАЦИИ В JAVA (КОГДА-ТО)

---

```
/** Cache the hash code for the string */
private int hash; // Default to 0

public int hashCode(char[] value) {
    int h = hash;
    if (h == 0 && value.length > 0) {
        for (int i = 0; i < value.length; i++) {
            h = 31 * h + value[i];
        }
        hash = h;
    }
    return h;
}
```

---

## ПРИМЕР РЕАЛИЗАЦИИ В JAVA: ПОЧЕМУ 31?

Joshua Bloch “Effective Java”

- Always override hashCode when you override equals

The value 31 was chosen because it is an odd prime. If it were even and the multiplication overflowed, information would be lost, as multiplication by 2 is equivalent to shifting. The advantage of using a prime is less clear, but it is traditional. A nice property of 31 is that the multiplication can be replaced by a shift and a subtraction for better performance:  $31 * i == (i \ll 5) - i$ . Modern VMs do this sort of optimization automatically.

## РЕАЛИЗАЦИЯ: МЕТОД ЦЕПОЧЕК (CHAINING)

- ▶ Массив размера  $m$
- ▶ В каждой ячейке храним список (key, value)

---

```
buckets = array of length m
function find(key):
  h = hash(key)
  node = buckets[h] // linked list
  while node != null:
    if node.key == key:
      return node.value
    node = node.next
  return null
end
```

---

## РЕАЛИЗАЦИЯ: ОТКРЫТАЯ АДРЕСАЦИЯ

- ▶ Массив размера  $m$
- ▶ В каждой ячейке храним (key, value) (не список!)
- ▶ Приведена одна из возможных реализаций открытой адресации (линейное исследование)  
probing = опробывание, исследование

---

```
buckets = array of length m
function find(key):
  h = hash(key)
  node = buckets[h] // array element
  if node.key == key:
    return node.value
  i = (h + 1) % m
  while node != null && i != h:
    if node.key == key:
      return node.data
  return null
end
```

---

## ВАРИАНТЫ ОТКРЫТОЙ АДРЕСАЦИИ

- ▶ Линейное исследование:

$$h(i, x) = h_1(x) + i \pmod{m}$$

- ▶ Квадратичное исследование:

$$h(i, x) = h_1(x) + c_2 i^2 + c_1 i \pmod{m}$$

Например:

$$h(i, x) = h_1(x) + 1 + 2 + \dots + i = h_1(x) + i(i + 1)/2$$

- ▶ Двойное хэширование:

$$h(i, x) = h_1(x) + i h_2(x) \pmod{m}$$

- ▶ Хэширование Робин Гуда ("Robin Hood hashing")
- ▶ Кукушкино хэширование (Cuckoo hashing)

## СЛОЖНОСТЬ: МЕТОД ЦЕПОЧЕК

**Утверждение:** Математическое ожидание сложности неудачного поиска при условии простого равномерного хеширования равна  $\Theta(1 + \alpha)$ , где  $\alpha$  - коэффициент заполнения таблицы. ( $\alpha = \frac{n}{m}$ , где  $m$  - количество ячеек)

## СЛОЖНОСТЬ: ОТКРЫТАЯ АДРЕСАЦИЯ

**Утверждение:** Математическое ожидание сложности неудачного поиска при условии простого равномерного хеширования равна  $O(\frac{1}{1-\alpha})$ , где  $\alpha < 1$  - коэффициент заполнения таблицы. ( $\alpha = \frac{n}{m}$ , где  $m$  - количество ячеек)

## СЛОЖНОСТЬ: ОТКРЫТАЯ АДРЕСАЦИЯ

**Утверждение:** Математическое ожидание сложности неудачного поиска при условии простого равномерного хеширования равна  $O(\frac{1}{1-\alpha})$ , где  $\alpha < 1$  - коэффициент заполнения таблицы. ( $\alpha = \frac{n}{m}$ , где  $m$  - количество ячеек)  
Если  $X$  = количество проверок при неуспешном поиске, то

$$E[X] = \sum_{i=1}^{\infty} P\{X \geq i\} \leq \sum_{i=0}^{\infty} \alpha^i = \frac{1}{1-\alpha}$$

т.к.

$$\sum_{i=0}^{\infty} iP\{X = i\} = \sum_{i=0}^{\infty} i(P\{X \geq i\} - P\{X \geq i+1\}) = \sum_{i=0}^{\infty} P\{X \geq i\}$$

# ОТКРЫТАЯ АДРЕСАЦИЯ VS МЕТОД ЦЕПОЧЕК

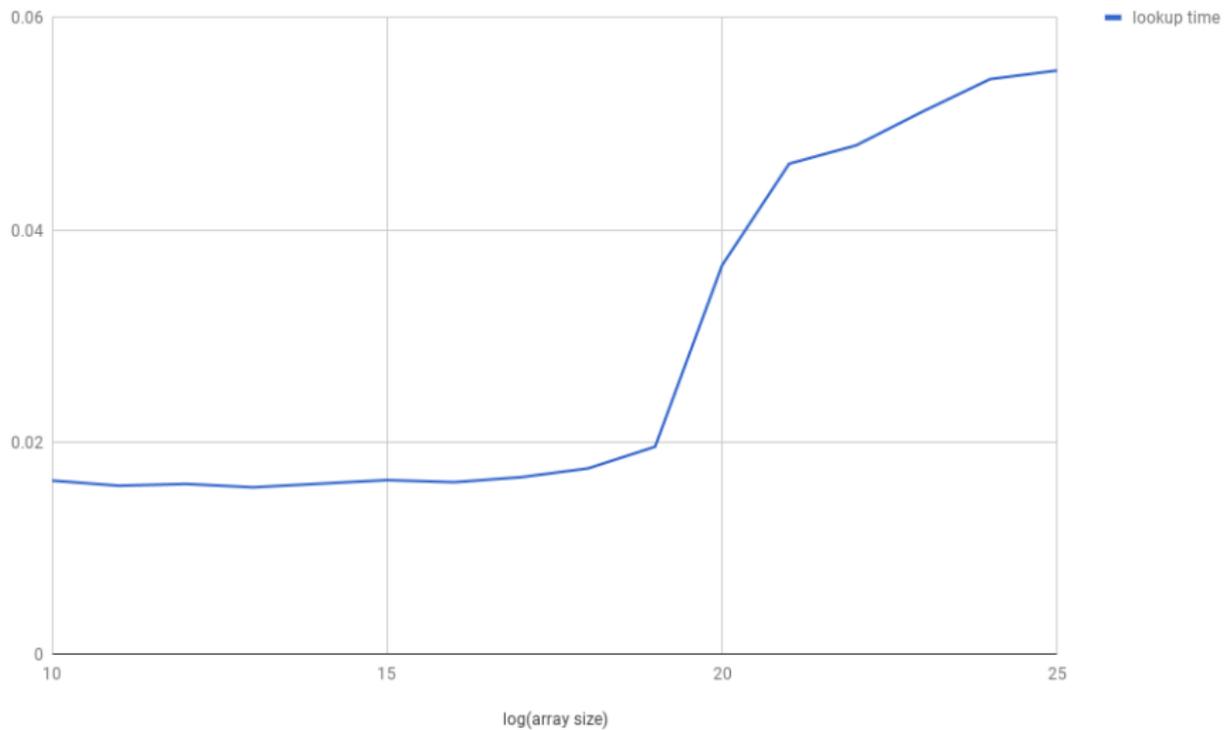
Недостатки:

- ▶ Сложность реализации (относительная)
- ▶ Сложность удаления
- ▶ Большая чувствительность к коэффициенту заполнения

Преимущества:

- ▶ Локальность!!!

# ЛОКАЛЬНОСТЬ



## КУКУШКИНО ХЕШИРОВАНИЕ

- ▶ Используются 2 массива  $A_1, A_2$  каждый размером  $\frac{m}{2}$
- ▶ 2 хеш функции.  $h_1, h_2 : U \rightarrow \{0, \dots, \frac{m}{2} - 1\}$
- ▶ Каждый ключ  $x \in U$  Может находится в одном из двух мест: в ячейке  $h_1(x)$  в первом массиве или в ячейке  $h_2(x)$  во втором массиве
- ▶ Если вставляем ключ  $x$  и место  $h_1(x)$  занято ключом  $y$ , то "выкидываем"  $y$ , кладем  $x$ , а  $y$  перемещаем в  $h_2(y)$  и далее по цепочке
- ▶ Если цепочка перемещений длиннее порога, выбираем новые хэш функции и перестраиваем всю таблицу
- ▶ Сложность поиска и удаления в худшем случае -  $O(1)$
- ▶ Сложность вставки при достаточно низком коэффициенте заполнения в среднем -  $O(1)$

# РАСПРЕДЕЛЕННЫЕ ХЭШ-ТАБЛИЦЫ

Объект  $x \in U$

Ячейки (сервера):  $S = \{S_1, \dots, S_n\}$

Необходимо построить хеш-функцию

$$H : U \rightarrow S$$

Свойства:

- ▶ Равномерное распределение
- ▶ Минимальная перестройка при изменении  $S$   
(добавлении/удалении элемента)

# ПРИМЕНЕНИЕ

- ▶ Распределение нагрузки
- ▶ Распределённые хеш таблицы
- ▶ Распределённые БД

# КОНСИСТЕНТНОЕ ХЕШИРОВАНИЕ (CONSISTENT HASHING/HASH RING)

- ▶ Представляем диапазон  $[1 \dots 2^{32}]$  как окружность  $S^1$
- ▶ Для каждого  $S_i$  генерим  $m$  случайных точек  $s_{ij} \in S^1$
- ▶ Хеш функция  $h : U \rightarrow S^1$
- ▶ Если ближайшая по часовой стрелке к  $h(x)$  – точка  $s_{kj} \in S^1$ , то полагаем  $H(x) = S_k$ .

## RENDEZVOUS ХЕШИРОВАНИЕ

- ▶ Хеш-функция

$$h : U \times S \rightarrow [1 \dots 2^{32}]$$

- ▶ Для  $x \in U$  выбираем ячейку  $S_k$ , на которой хеш-функция принимает максимальное значение:

$$H(x) = \underset{s}{\operatorname{argmax}} h(x, s)$$

Другими словами

$$H(x) = k, \text{ если } h(x, S_k) \geq h(x, S_i) \forall i$$

## ФИЛЬТР БЛУМА (BLOOM FILTER)

Вероятностная структура данных (множество). Операции:

- ▶ Вставка
- ▶ Поиск по ключу (бинарный результат)

vs Хеш-таблица:

- ▶  $\oplus$  память и скорость!!!
- ▶  $\ominus$  нет удалений
- ▶  $\ominus$  нельзя хранить значение
- ▶  $\ominus$  вероятность ложноположительного поиска

## ФИЛЬТР БЛУМА - КОНСТРУКЦИЯ

- ▶ Битовый массив  $B$  длины  $n$  (например,  $n = 8|S|$ )
- ▶  $k$  хеш функций  $h_1, \dots, h_k, h_i : U \rightarrow \{1, \dots, n\}$

Вставка ключа  $x$ :  $B[h_i(x)] = 1$  для всех  $i = 1, \dots, k$

Поиск ключа  $x$ : "да"  $\Leftrightarrow B[h_i(x)] = 1$  для всех  $i = 1, \dots, k$

## ФИЛЬТР БЛУМА - АНАЛИЗ

В предположении равномерности и независимости хеш функций  $h_i$ : вероятность того, что конкретный бит установлен в 1 после вставки всего  $S$ , равна

$$1 - \left(1 - \frac{1}{n}\right)^{k|S|} \approx 1 - e^{-\frac{k|S|}{n}}$$

Вероятность ложноположительного поиска:

$$\left(1 - e^{-\frac{k|S|}{n}}\right)^k$$

## ФИЛЬТР БЛУМА - АНАЛИЗ

В предположении равномерности и независимости хеш функций  $h_i$ : вероятность того, что конкретный бит установлен в 1 после вставки всего  $S$ , равна

$$1 - \left(1 - \frac{1}{n}\right)^{k|S|} \approx 1 - e^{-\frac{k|S|}{n}}$$

Вероятность ложноположительного поиска:

$$\left(1 - e^{-\frac{k|S|}{n}}\right)^k$$

$$n/|S| = 8, \quad k = 2 \quad \longrightarrow \quad < 5\%$$

## ФИЛЬТР БЛУМА - АНАЛИЗ

В предположении равномерности и независимости хеш функций  $h_i$ : вероятность того, что конкретный бит установлен в 1 после вставки всего  $S$ , равна

$$1 - \left(1 - \frac{1}{n}\right)^{k|S|} \approx 1 - e^{-\frac{k|S|}{n}}$$

Вероятность ложноположительного поиска:

$$\left(1 - e^{-\frac{k|S|}{n}}\right)^k$$

$$n/|S| = 8, \quad k = 2 \quad \longrightarrow \quad < 5\%$$

$$n/|S| = 9, \quad k = 6 \quad \longrightarrow \quad \approx 2\%$$

## ФИЛЬТР БЛУМА - АНАЛИЗ

В предположении равномерности и независимости хеш функций  $h_i$ : вероятность того, что конкретный бит установлен в 1 после вставки всего  $S$ , равна

$$1 - \left(1 - \frac{1}{n}\right)^{k|S|} \approx 1 - e^{-\frac{k|S|}{n}}$$

Вероятность ложноположительного поиска:

$$\left(1 - e^{-\frac{k|S|}{n}}\right)^k$$

$$n/|S| = 8, \quad k = 2 \quad \longrightarrow \quad < 5\%$$

$$n/|S| = 9, \quad k = 6 \quad \longrightarrow \quad \approx 2\%$$

$$n/|S| = 16, \quad k = 11 \quad \longrightarrow \quad < 0.05\%$$

## ФИЛЬТР БЛУМА - АНАЛИЗ

Вероятность ложноположительного поиска:

$$\left(1 - e^{-\frac{k|S|}{n}}\right)^k$$

## ФИЛЬТР БЛУМА - АНАЛИЗ

Вероятность ложноположительного поиска:

$$\left(1 - e^{-\frac{k|S|}{n}}\right)^k$$

При фиксированном  $n/|S|$  как функция от  $k$  достигает минимума при

$$k^* = \frac{n}{|S|} \ln 2$$

## ФИЛЬТР БЛУМА - АНАЛИЗ

Вероятность ложноположительного поиска:

$$\left(1 - e^{-\frac{k|S|}{n}}\right)^k$$

При фиксированном  $n/|S|$  как функция от  $k$  достигает минимума при

$$k^* = \frac{n}{|S|} \ln 2$$

$$n/|S| = 9, \quad k^* \approx 6.24, \quad k = 6 \quad \longrightarrow \quad \approx 2\%$$

$$n/|S| = 16, \quad k^* \approx 11.09, \quad k = 11 \quad \longrightarrow \quad < 0.05\%$$

## ФИЛЬТР БЛУМА - ИСПОЛЬЗОВАНИЕ

[https://en.wikipedia.org/wiki/Bloom\\_filter](https://en.wikipedia.org/wiki/Bloom_filter)

Akamai Technologies (CDN) - to prevent "one-hit-wonders" from being stored in its disk caches. One-hit-wonders are web objects requested by users just once, something that Akamai found applied to nearly three-quarters of their caching infrastructure. Using a Bloom filter to detect the second request for a web object and caching that object only on its second request prevents one-hit wonders from entering the disk cache, significantly reducing disk workload and increasing disk cache hit rates.

Google Bigtable, Apache HBase and Apache Cassandra and PostgreSQL - to reduce the disk lookups for non-existent rows or columns.

Google Chrome web browser - to identify malicious URLs  
Microsoft Bing (search engine) uses multi-level hierarchical Bloom filters for its search index, BitFunnel. Bloom filters provided lower cost than the previous Bing index which was