Разработка и анализ алгоритмов

Лекция 8
Хеширование

Сергей Леонидович Бабичев

План лекции

1. Обобщённый быстрый поиск.
2. Хеш-функции.
3. Применение хеш-функций.
4. Алгоритм Карпа-Рабина

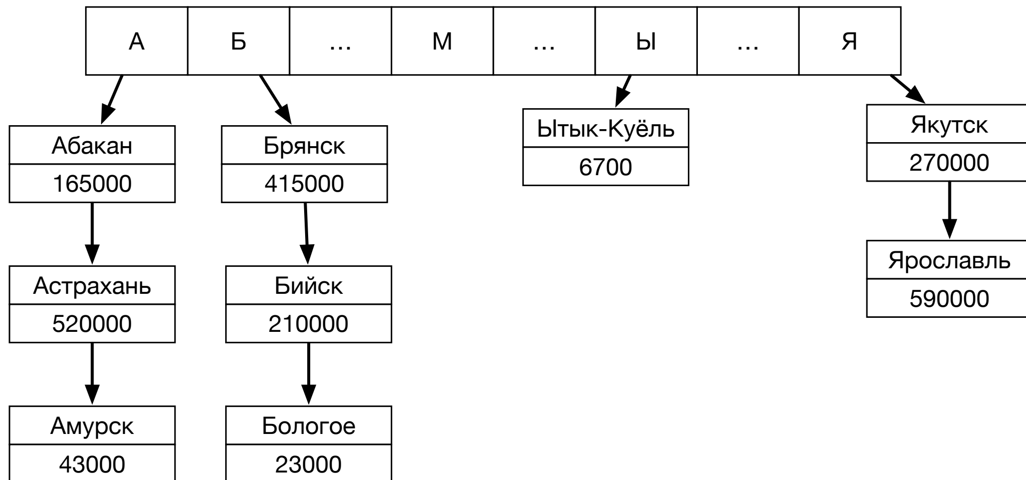
Обобщённый быстрый поиск

Обобщённый быстрый поиск

- Требуется:
  - Разработать эффективную `CRUD-структуру`
  - Уменьшить амортизационную стоимость поиска.
  - Уменьшить сложность функции, например, $O(\log N) \to O(1)$.

Обобщённый быстрый поиск

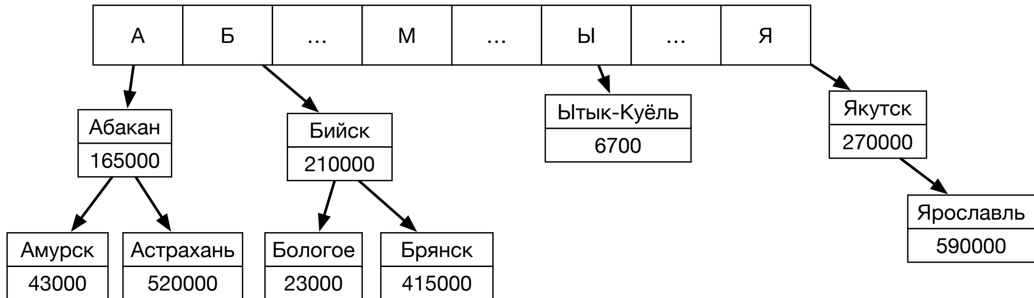
- База данных названий городов и их численности.



33 вторичные поисковые структуры — связные списки.

Обобщённый быстрый поиск

- База данных названий городов и численности их населения.



33 вторичные поисковые структуры — деревья поиска.

Обобщённый быстрый поиск

- Основная идея — разбиение пространства ключей на независимые подпространства (*partitioning*).
- При независимом разбиении на M подпространств сложность уменьшается.

Для разбиения множества N ключей на примерно равные M подмножеств сложность вычисляется по главной теореме о рекурсии при числе подзадач M, коэффициенте размножения 1 и консолидации $O(1)$.

$$C \cdot O(N) \to \frac{C}{M} O(N)$$

$$C \cdot O(N \log N) \to \frac{C}{M} O(N \log N)$$

Обобщённый быстрый поиск

- При увеличении M

$$\lim_{K\to\infty} T(N, M) = O(1)$$

$$\lim_{K\to\infty} Mem(N, M) = \infty$$

- Имеется зона оптимальности при $M \approx N$

Обобщённый быстрый поиск

- Требуется иметь детерминированный способ разбиения пространства ключей на M независимых подпространств.
- Условия разбиения:

$$|K_1| \approx |K_2| \approx \cdots \approx |K_M|$$

$$\sum_{i=1}^{M} |K_i| = |K|$$

- Эврика! Создаём функцию $H(K)$, удовлетворяющую некоторым условиям.

Хеш-функции

Хеш-функции

Definition (Хеш-функция)

Сюръективное отображение H множества K в множество V называется *хеш-функцией* $H(K) \to V$ при условии $|K| > |V|$.

- Отображение пространства ключей K на пространство значений V.
- $M = |V|$ — мощность множества пространства значений.
- Обычно $V \equiv \mathbb{Z}_M$.

Хеш-функции

- Введём понятие *соперника*, то есть того, кто предоставляет нам ключи.
- Цель *соперника* — предоставлять ключи таким образом, чтобы значения функции оказались не равновероятными.
- *Соперник* знает хеш-функцию и может выбирать ключи.

Хеш-функции

Хотелось бы обеспечить свойства:

- **Эффективность.**

$$T(H(K)) \leqslant O(L(K)),$$

  где $L(K)$ — мера длины ключа K.
- **Равномерность.** Каждое выходное значение равновероятно.

$$p_{H(K_1)} = p_{H(K_2)} = \cdots = p_{H(K_M)}$$

- **Лавинность.** При изменении одного бита во входной последовательности изменяется значительное число выходных битов.
- Для борьбы с *соперником* — **необратимость**, то есть невозможность восстановления ключа по значению его функции.

Хеш-функции

Следствия их требуемых свойств.

- Функция не должна быть непрерывной. Для близких значений аргумента должны получаться сильно различающиеся результаты.
- В значениях функции не должно образовываться *кластеров*, множеств близко стоящих точек.

Определение непрерывности для дискретных функций может быть дано неформально.

Хеш-функции

Примеры плохих функций:

- $H = K^2$
  Функция монотонно возрастает. Пространство значений ключа слишком велико и часть значений недостижима.
- $H = \sum_{i=0}^{s.size()-1} s[i]$ для строки s.
  Функция даёт одинаковые значения для строк abcd и abdc и отличающиеся на единицу для строк abcd и abde. Сопернику легко найти ключи, которые дают равные значения функции.

Универсальная хеш-функция

Definition (Коллизия)

Совпадение значений функции для разных значений ключа называется **коллизией**.

Definition (Универсальное множество хеш-функций)

Пусть H^* — множество хеш-функций, которые отображают пространство ключей в $m = |D(M)|$ различных значений. Это множество *универсально*, если для каждой пары ключей $K_i, K_j, i \neq j$ количество хеш-функций, для которых $H^*(K_i) = H^*(K_j)$ не более $\dfrac{|H^*|}{m}$.

Универсальная хеш-функция

- Если случайным образом выбирается функция из множества H^*, то для случайной пары ключей $K_i, K_j, i \neq j$ вероятность коллизии не должна превышать $\dfrac{1}{m}$

Theorem (Универсальное множество хеш-функций)

- *Пусть множество* $Z_p = \{0, 1, \ldots, p-1\}$, *множество* $Z_p^* = \{1, 2, \ldots, p-1\}$, p *— простое число,* $a \in Z_p^*$, $b \in Z_p$.
- *Тогда множество*

$$H^*(p, m) = \{H(a, b, K) = ((aK + b) \bmod p) \bmod m\}$$

есть универсальное множество хеш-функций.

Универсальное множество хеш-функций

Доказательство.

- Рассмотрим два разных ключа $k, l \in \mathbb{Z}_p$. Тогда $r = (ak + b) \bmod p \neq s = (al + b) \bmod p$ — мультипликативность группы по модулю p.
- Всего имеется $p(p-1)$ функций для разных (a, b). Каждая пара (a, b) даст биективно различную пару (r, s).
- Вероятность коллизии k и l есть вероятность $r \equiv s \pmod{()}m$.
- В подсчёте матожидания коллизии для каждого r имеется $p-1$ значение s.
- Число таких s, что $s \neq r$ и $s \equiv r \pmod m$ есть $\lceil \frac{p}{m} \rceil - 1 \leqslant \frac{p-1}{m}$.
- Имеется $p-1$ равновероятное событие.

$$\mathrm{Prob}\{H(a,b,k) = H(a,b,l)\} \leqslant \frac{\frac{p-1}{m}}{p-1} \leqslant \frac{1}{m}$$

Хеш-функции

- Не универсальная, не не столь уж и отвратительная функция

$$h = \sum_{i=0}^{n} s_i \times 8^i \bmod HASHSIZE$$

Обратная схема Горнера:

```
unsigned hash_sum(char const *s, unsigned HASHSIZE) {
    unsigned sum = 0;
    const int FACTOR = 5;
    while (*s) {
        sum <<= FACTOR;
        sum += *s++;
    }
    return sum % HASHSIZE;
}
```

Хеш-функции

- Хеш-функция получше

```
unsigned hash_sedgwick(char const *s, unsigned HASHSIZE) {
    unsigned h = 0, a = 31415, b = 27183;
    while (*s) {
        h = (a * h + *s++) % HASHSIZE;
        a = a * b % (HASHSIZE-1);
    }
    return h;
}
```

Хеш-функции

- Лучшие по статистическим показателям функции — криптографические.
- Недостатки:
  - длинный код
  - медленные

Весьма хорошая хеш-функция

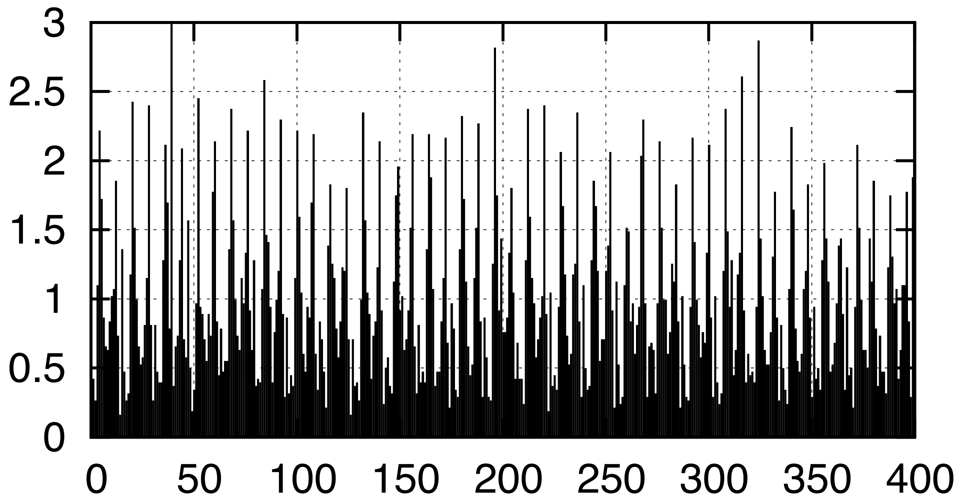
Пользуется свойствами полей Галуа $GF(2^{32})$:

```
unsigned hash_crc32(char const *s) {
   unsigned ret = 0xFFFFFFFF;
   while (*s) {
      ret ^= *s & 0xFF;
      ret = (ret >> 8) ^ table[ret & 0xFF];
   }
   return ret ^ 0xFFFFFFFF;
}
```

table вычисляется заранее по какому-нибудь неприводимому полиному в поле $GF(2^{32})$.

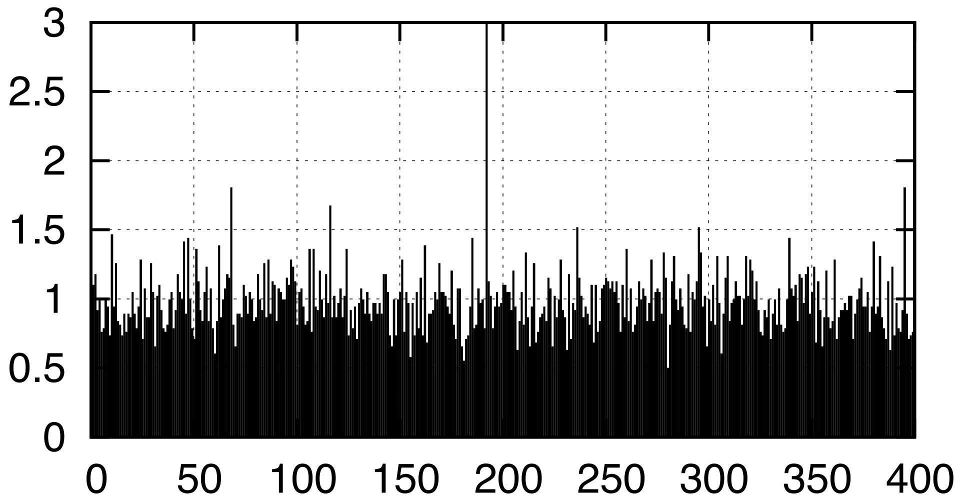
Хеш-функции: исследование свойств

Распределение значений для случайных идентификаторов. Плохая функция.



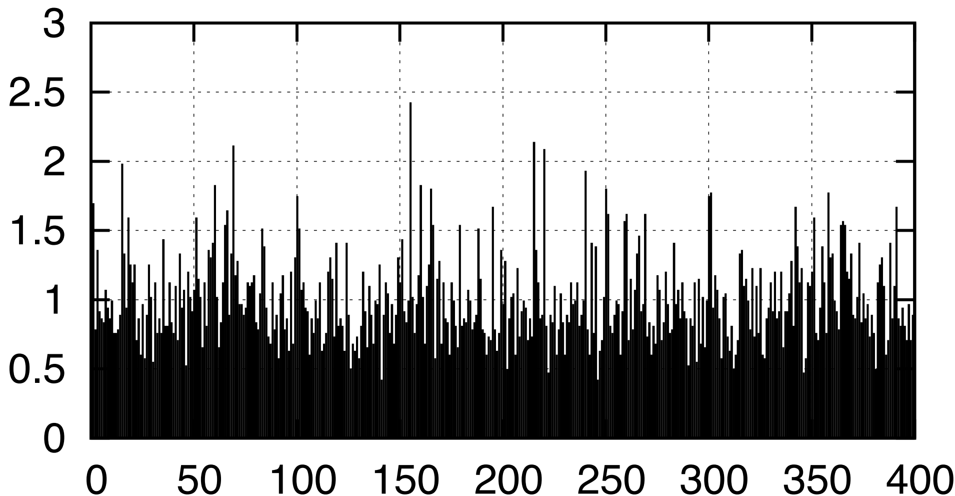
Хеш-функции

Распределение значений для случайных идентификаторов. Плохая функция.



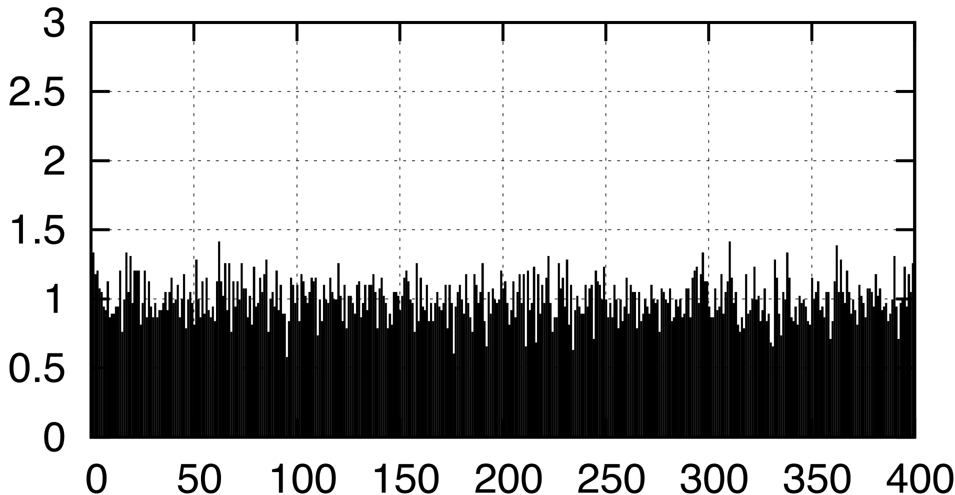
Хеш-функции

Хорошая функция.



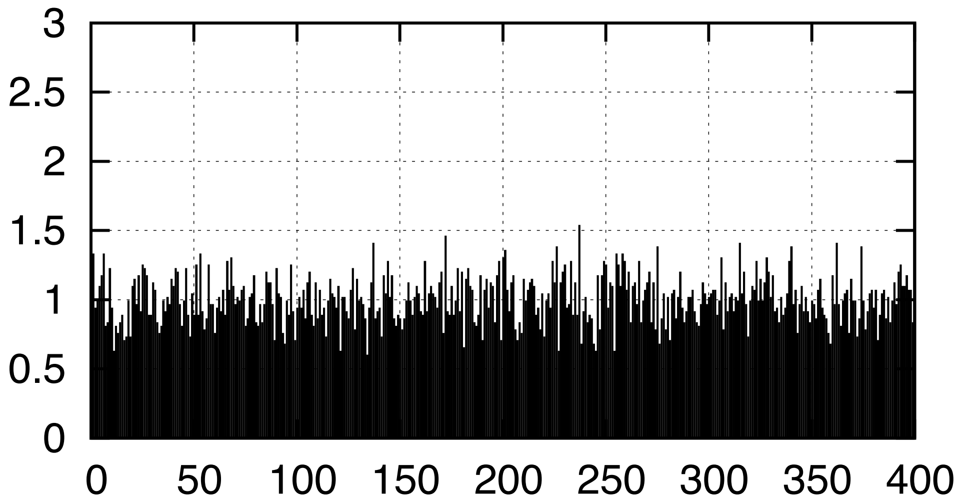
Хеш-функции

Хорошая функция.



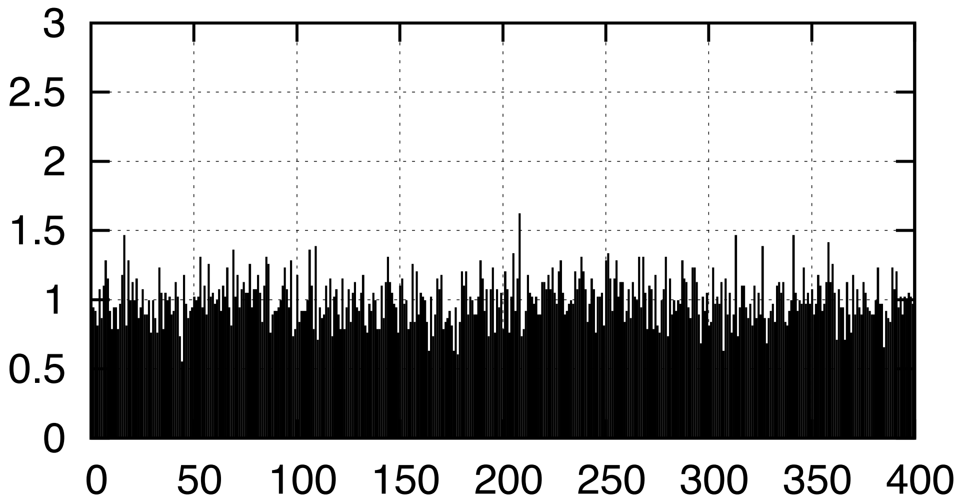
Хеш-функции

Отличная функция.



Хеш-функции

Отличная функция.



Затраты времени на исполнение хеш-функций

Алгоритм/набор	include.txt	source.txt
hash_sum	890	786
hash_sedgewick	2873	2312
hash_crc	912	801

Применение хеш-функций

Вероятностный подход к надёжности

Надёжны ли современные вычислительные системы?

- Производитель серверной памяти с коррекцией ошибок `IBM` измерил, что произошло 6 отказов на 10000 серверов за три года с 4ГБ памяти.
- Один отказ на 10^{20} обработанных байт.
- Сравним два блока памяти во 4096 байт. Вероятность получения неверного ответа при их равенстве есть $\dfrac{4096}{10^{20}} \approx 2.5 \cdot 10^{-16}$.
- Вероятность совпадения значений хорошей 64-битной хеш-функции для двух блоков данных размером в 4096 байт есть $\dfrac{4096}{2^{64}} = 2^{-52} \approx 10^{-17.1}$, то есть сравнима!

Синхронизация больших объектов

Условия применения:

- Синхронизируемый объект имеет значительный размер.
- Объект регулярно изменяет своё содержимое.
- Размер изменяемой зоны относительно невелик.

Обычное копирование расходует ресурс: пропускную способность.

Синхронизация больших объектов

Два паттерна использования:

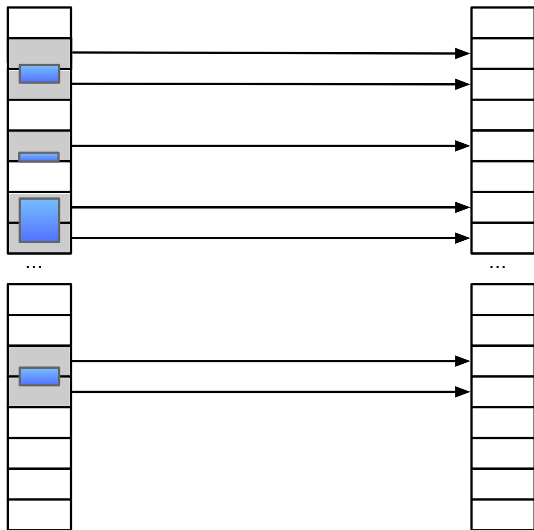
1. первичная пересылка объекта. Может потребовать передачи полного объёма.
2. пересылка изменённых фрагментов.

Синхронизация больших объектов: алгоритм

Задача: клиент синхронизирует большой объект с сервера.
Условия: на клиенте и сервере имеются реплики большого объекта, возможно, уже изменившегося на сервере. Используется одна и та же хеш-функция.

1. клиент и сервер разбивают объект на (виртуальные) блоки. Для каждого блока подсчитывается хеш.
2. клиент передаёт серверу номера блоков, для которых нужно вычислить хеш
3. сервер передаёт хеш запрошенных блоков
4. клиент сравнивает хеш и обнаруживает блоки с несовпадающем хешем
5. клиент запрашивает блоки с несовпадающем хешем

Синхронизация больших объектов



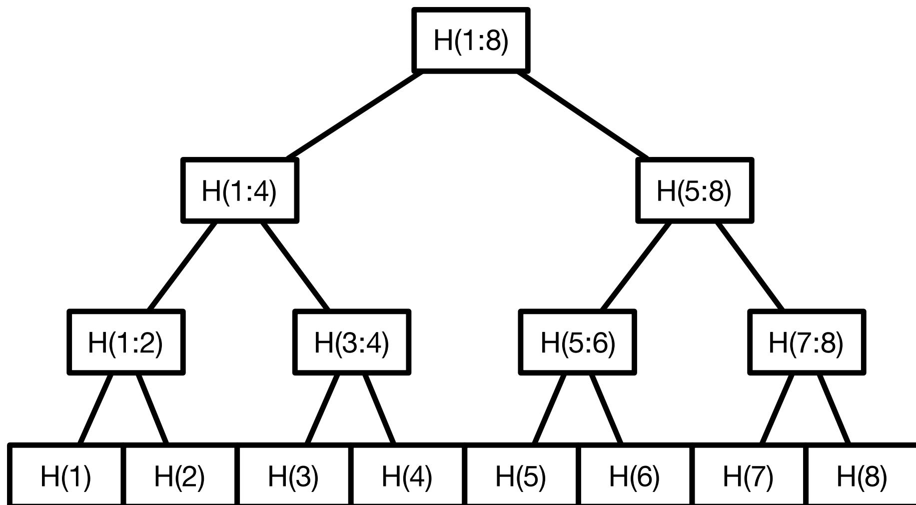
Синхронизация больших объектов: синий цвет — изменённые данные, серый — передаваемые блоки

Синхронизация больших объектов: продвинутый алгоритм

- Классические хеш-функции отображают множество ключей на множество значений.
- Зная значения функции для сообщений `A` и `B`, соответственно как `H(A)` и `H(B)` мы обычно не можем вычислить `H(AB)`, где `AB` — конкатенация сообщений `A` и `B`.
- *аддитивная* хеш-функция по `H(A)` и `H(B)` способна вычислить `H(AB)`.

Синхронизация больших объектов

Пусть синхронизируемый объект состоит из 2^N блоков.



Структура данных для усовершенствованной репликации

Синхронизация больших объектов

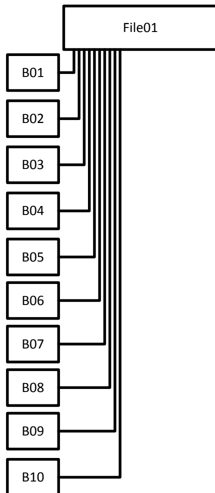
- Пусть `H(U:V)` — значение аддитивной хеш-функции от множества блоков от `U` до `V` включительно.
- Структура данных — дерево, на вершине которого находится узел, содержащий значение хеш-функции от всего объекта.
- Уровнем ниже — два узла, содержащие значения хеш-функции от половины объекта и так далее.
- Терминальные узлы содержат значения хеш-функции от отдельных блоков.
- Свойство аддитивности позволяет нам восстановить любой узел дерева по значению его потомков.

Синхронизация больших объектов: алгоритм синхронизации

- После первой фазы построения имеются деревьев с обоих сторон.
- Если значения хеш-функций для корня дерева совпали — алгоритм завершается.
- Рассматриваются потомки узла и спуск по дереву производится только в случае несовпадения значений хеш-функции на стороне оригинала и на стороне копии.

Дедупликация

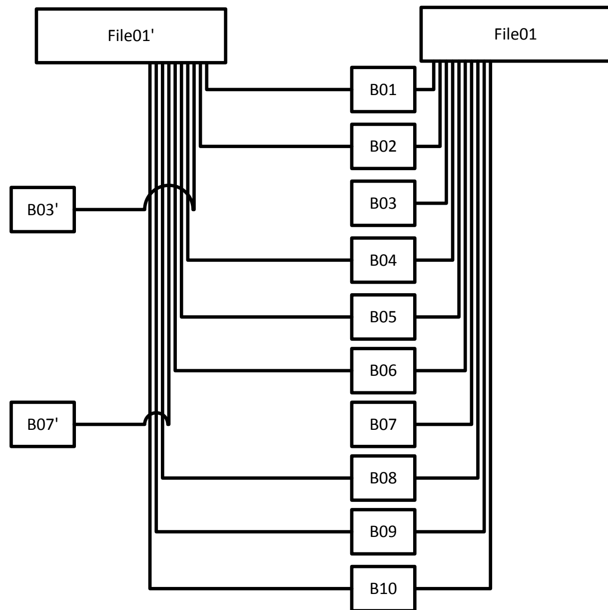
- Имеется блочное хранилище.
- В хранилище копия файла `File01`, 10 блоков, от `B1` до `B10`.



Дедупликация

- Изменились блоки B3 и B7.
- Вариант 1: создать копию нового файла, содержащую все 10 блоков.
- 8 блоков — B1, B2 ... будут совпадать.
- Если есть возможность определить, что изменились именно блоки B3 и B7, то в хранилище достаточно передать именно эти блоки и заменить ими старые блоки B3 и B7.
- Старые блоки B3 и B7 сохраняются.
- Наличие новые блоков B3' и B7' позволит нам иметь два поколения файла.
- Соответствующими запросами можно будет извлечь две разных версии файла размером в 10 блоков, хотя в хранилище находятся только 12 блоков

Дедупликация: схема хранения нескольких версий файла



Дедупликация

- Дедуплицированное хранилище содержит только уникальные блоки.
- Каждый блок при поступлении в хранилище проверяется на уникальность — имеется ли уже блок с таким содержимым.
- При совпадении блока в файле с уже имеющимся блоком, в карте хранения файла делается соответствующая запись.

Дедупликация: алгоритм

1. Определить множество блоков, участвующих в операции.
2. Для каждого из блоков множества вычислить хеш.
3. Если блок с таким хешем имеется в пуле, связать блок файла с блоком пула.
4. Если блока с таким хешем не имеется, создать новый блок пула, связать блок файла с вновь созданным блоком пула.

Дедупликация: проблемы

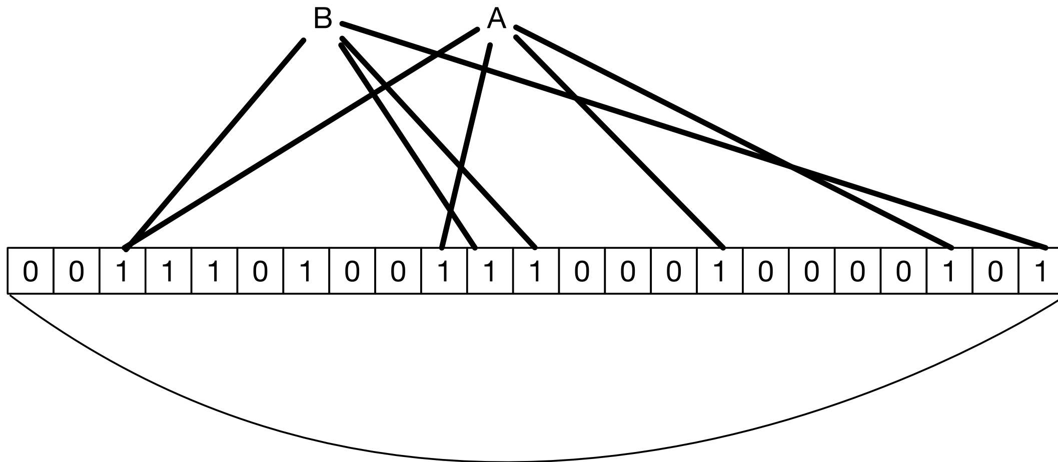
- Основная операция — поиск блока.
- Если блоков немного — создаётся отображение хешей блоков на реальное хранилище самих блоков.
- Подобное отображение — *персистентная* таблица.
- Серьёзное ограничение — размер таблицы.
- Оперативной памяти хватит лишь на миллиарды записей.
- Используют B-дерево (B+-дерево) или хеш-таблицу.
- Для уменьшения количества операций отображения применяют *вероятностные множества*.

Вероятностные множества

- Операции `insert`.
- Операция `find` с отсутствием гарантии точности результата поиска в этом множества.
- Если поиск вернул `true`, то элемент *может* и отсутствовать, и присутствовать.
- Если поиск вернул `false`, то элемент заведомо отсутствует.
- `true` означает `МОЖЕТ БЫТЬ`.
- `false` означает `НЕТ`.

Фильтр Блума

Реализация фильтра Блума: битовый массив из m бит и n различных хеш-функций $h_1, \dots h_n$, равномерно отображающих элементы на номера битов.

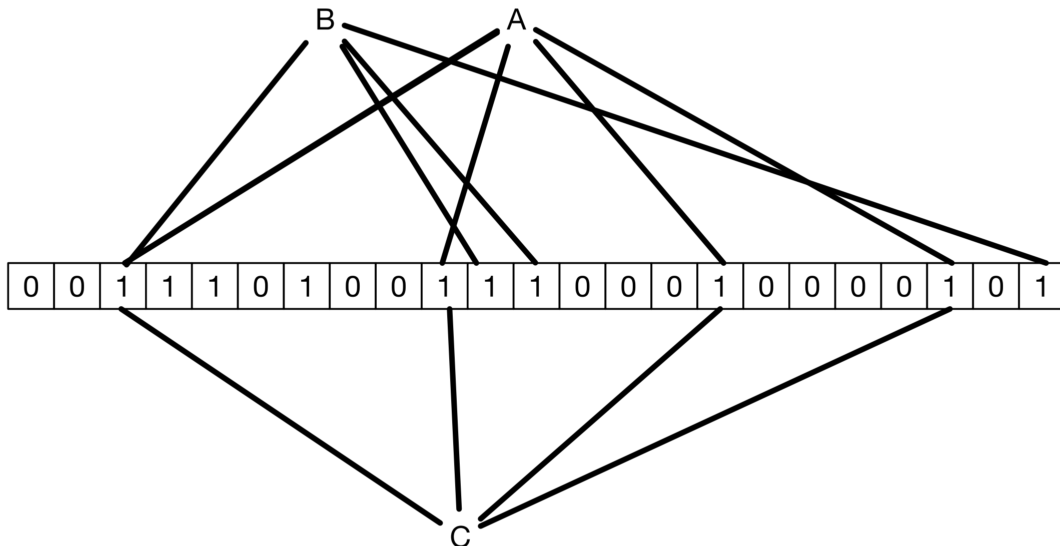


Фильтр Блума

- **create**: все биты равны нулю.
- **insert**: вычисляются все n хеш-функций и устанавливаются соответствующие биты.
- **find**: вычисляются все n хеш-функций.
  - Если хотя бы один бит не совпал, то ответ точен: `НЕТ`.
  - Если совпали все биты, то ответ: `МОЖЕТ БЫТЬ`.

Фильтр Блума

Для элементов A и C не равных друг другу все их хеши могут совпасть:



Фильтр Блума: свойства

- Это действительно фильтр, который помогает отсеять заведомо ненужные элементы.
- При добавлении элементов количество установленных битов увеличивается и его точность уменьшается.
- Предельный случай: все биты установлены. Любой элемент `МОЖЕТ БЫТЬ`.
- Оптимальное число хеш-функций для m битов и t элементов

$$b = \log_2 \frac{m}{t}.$$

- Идеально приспособлен для уменьшения числа сложных операций (обращение к внешней памяти) и замене их более простыми (обращение к оперативной памяти).
- Операции удаления реализуются тяжело (требуется изменение представления).

Алгоритм Карпа-Рабина

Поиск подстрок в строке: алгоритм Карпа-Рабина

Задача 1. Имеется исходная строки и образец. Определить позицию в исходной строке, полностью содержащую образец.

Упростим задачу.

Пусть строки состоят из символов `A`, `B`, `C`, `D`.
Отобразим их в `1`, `2`, `3`, `4`.

Строка-образец — pat=`ABAC` или `1213`.

Строка-источник — src=`ACABAACABACAABCA`

Поиск подстрок в строке: алгоритм Карпа-Рабина

A	B	A	C
1	2	1	3

A	C	A	B	A	A	C	A	B	A	C	A	A	B	C	A
1	3	1	2	1	1	3	1	2	1	3	1	1	2	3	1

Выберем *простое* число, немного превышающее мощность алфавита $P = 5$.
Составим таблицу `T` степеней числа P по модулю 2^{32}

0	1	2	3	4	5	6	7	8	9
1	5	25	125	625	3125	15625	78125	390625	1953125

10	11	12	13	14	15
9765625	48828125	244140625	1220703125	1808548329	452807053

Алгоритм Карпа-Рабина

Хеш-функция от строки `S` в поддиапазоне `[k...r]`:

$$H(S_{[k,r]}) = \sum_{i=k}^{r} S_{i-k} \cdot P^{i-k} = \sum_{i=k}^{r} S_{i-k} \cdot T_{[i-k]}$$

Алгоритм Карпа-Рабина

Вычислим хеш строки `pat` и подстрок строки `src` длиной 4:

$$H\left(pat_{[0,4)}\right) = H(ABAC) = 1 \cdot 5^0 + 2 \cdot 5^1 + 1 \cdot 5^2 + 3 \cdot 5^3 = 411$$

$$\begin{aligned}
H\left(src_{[0,4)}\right) &= 291 \\
H\left(src_{[1,5)}\right) &= 183 \\
H\left(src_{[2,6)}\right) &= 161 \\
H\left(src_{[3,7)}\right) &= 407 \\
H\left(src_{[4,8)}\right) &= 206 \\
H\left(src_{[5,9)}\right) &= 291 \\
H\left(src_{[6,10)}\right) &= 183 \\
H\left(src_{[7,11)}\right) &= 411 \\
H\left(src_{[8,12)}\right) &= 207 \\
H\left(src_{[9,13)}\right) &= 166 \\
H\left(src_{[10,14)}\right) &= 283 \\
H\left(src_{[11,15)}\right) &= 431
\end{aligned}$$

Алгоритм Карпа-Рабина

Хеш-функция для наших строк:

```
unsigned hash(char const *s, unsigned left, unsigned right,
       unsigned const *ptab) {
  unsigned sum = 0;
  for (unsigned i = left; i < right; i++) {
    sum += (s[i]-'A'+1) * ptab[i-left];
  }
  return sum;
}
```

Алгоритм Карпа-Рабина

Поиск подстроки:

```
size_t s1_size = strlen(s1), s2_size = strlen(s2);
unsigned hs1 = hash(s1, 0, s1_size, ptab);
for (unsigned i = 0; i < s2_size - s1_size; i++) {
    unsigned hs2 = hash(s2, i, i+s1_size, ptab);
    if (hs2 == hs1) {
        int ok = 1;
        for (unsigned j = 0; ok && j < s1_size; j++) {
            if (s1[j] != s2[i+j]) {
                ok = 0;
            }
        }
        if (ok) {
            printf("match at: %u\n", i);
        }
    }
}
```

Алгоритм Карпа-Рабина

Какова сложность кода при условии, что `N=src_size`, `M=pat_size`?

Алгоритм Карпа-Рабина

Какова сложность кода при условии, что `N=src_size`, `M=pat_size`?

$O(N \cdot M)$

Что не так?

Алгоритм Карпа-Рабина

Какова сложность кода при условии, что `N=src_size`, `M=pat_size`?

$O(N \cdot M)$

Что не так?

Мы делали много лишних вычислений.

Алгоритм Карпа-Рабина

$$H\left(s_{[0,4)}\right) = s_0 + s_1 \cdot p^1 + s_2 \cdot p^2 + s_3 \cdot p^3$$

Попробуем применить индукцию и вычислить, скажем, $H\left(s_{[1,5)}\right)$.

$$H\left(s_{[1,5)}\right) = s_1 + s_2 \cdot p^1 + s_3 \cdot p^2 + s_4 \cdot p^3$$

Умножим на p^1:

$$H\left(s_{[1,5)}\right) \cdot p^1 = s_1 \cdot p^1 + s_2 \cdot p^2 + s_3 \cdot p^3 + s_4 \cdot p^4$$

Сравним с

$$H\left(s_{[0,5)}\right) = s_0 + s_1 \cdot p^1 + s_2 \cdot p^2 + s_3 \cdot p^3 + s_4 \cdot p^4$$

$$H\left(s_{[k,l)}\right) \cdot p^k = H\left(s_{[0,l)}\right) - H\left(s_{[0,k)}\right)$$

Алгоритм Карпа-Рабина

Достаточно вычислить хеш от всех подстрок строки `src`.

$$
\begin{aligned}
H\left(src_{[0,1)}\right) &= 0\\
H\left(src_{[0,2)}\right) &= 1\\
H\left(src_{[0,3)}\right) &= 16\\
H\left(src_{[0,4)}\right) &= 41\\
H\left(src_{[0,5)}\right) &= 291\\
H\left(src_{[0,6)}\right) &= 916\\
H\left(src_{[0,7)}\right) &= 4041\\
H\left(src_{[0,8)}\right) &= 50916\\
H\left(src_{[0,9)}\right) &= 129041\\
H\left(src_{[0,10)}\right) &= 910291\\
H\left(src_{[0,11)}\right) &= 2863416\\
H\left(src_{[0,12)}\right) &= 32160291\\
H\left(src_{[0,13)}\right) &= 80988416\\
&\dots
\end{aligned}
$$

Алгоритм Карпа-Рабина

```
int karp_rabin(char const *s1, char const  *s2, unsigned const *ptab) {
    size_t s1_size = strlen(s1), s2_size = strlen(s2);
    unsigned hs1 = hash(s1, 0, s1_size, ptab);
    htab = malloc(sizeof(unsigned) * s2_size);
    for (unsigned i = 1; i < s2_size; i++)
        htab[i] = htab[i-1] + (s2[i-1] - 'A' + 1)*ptab[i-1];
    for (unsigned i = 0; i < s2_size - s1_size; i++) {
        unsigned hs2 = htab[i+s1_size] - htab[i];
        if (hs2 == hs1) {
            int ok = 1;
            for (unsigned j = 0; j < s1_size; j++)
                if (s1[j] != s2[i+j])
                    ok = 0;
            if (ok) { free(htab); return i; }
        }
        hs1 *= 5;
    }
    free(htab);
    return -1;
}
```

Алгоритм Карпа-Рабина

Применённая здесь хеш-функция имеет свойство *rolling-hash* и можно идти другим путём.

Если существуют такие p и x, что уравнение

$$p \cdot x = 1 \pmod{M}$$

имеет решение, то элемент x является обратным элементом для p в кольце вычетов по модулю M.

Необходимое условие: $\gcd(p, m) = 1$.
По малой теореме Ферма

$$p^{m-1} \equiv 1 \pmod{m}$$

Соответственно, $p \cdot p^{m-2} \equiv 1 \pmod{m}$, $x = p^{m-2} \pmod{m}$

Алгоритм Карпа-Рабина

Иногда можно заменить деление на p умножением на p^{-1}.
У нас:
$p = 5$,
$m = 2^{32}$,
$p^{-1} \pmod{m} = 3435973837$.

Что выведет программа:

```
#include <stdio.h>

int main() {
    for (unsigned x = 5; x < 1000; x += 5) {
        printf("%u\n", x * 3435973837u);
    }
}
```