

# Алгоритмы и структуры данных

## Лекция 9

### Хеш-функции.

Сергей Леонидович Бабичев

## План лекции

- 1 Ещё раз об абстракции *Отображение*.
- 2 Обобщённый быстрый поиск.
- 3 Хеш-функции.
- 4 Применение хеш-функций.
- 5 Алгоритм Карпа-Рабина

# Ещё раз об абстракции *Отображение*

# Абстракция отображение

Интерфейс абстракции *отображение* как ассоциативного массива.

- $m[key] = value$  — добавить элемент с ключом *key* и значением *value*
- $value = m[key]$  — найти элемент с ключом *key* и вернуть его.
- $m[key] = nil$  — удалить элемент с ключом *key*
- $for (auto x: m)$  — получить все ключи (или все пары ключ/значение) в каком-либо порядке.

## Абстракция отображение как хранилище

Структура данных операция	Худшее время	Среднее время
Balanced BST: <b>C</b> reate	$O(\log N)$	$O(\log N)$
Balanced BST: <b>R</b> ead	$O(\log N)$	$O(\log N)$
Balanced BST: <b>U</b> psert	$O(\log N)$	$O(\log N)$
Balanced BST: <b>D</b> elete	$O(\log N)$	$O(\log N)$

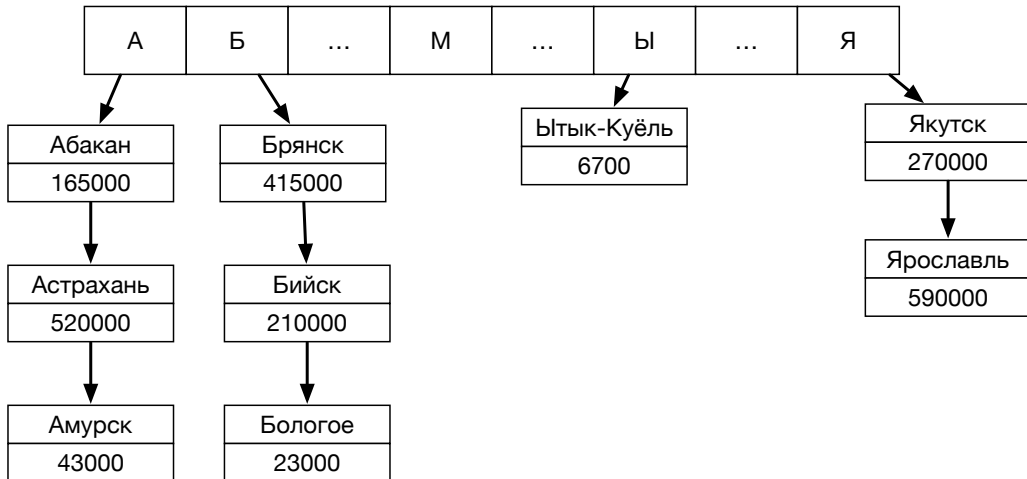
# Обобщённый быстрый поиск

# Обобщённый быстрый поиск

- Требуется:
  - ▶ Уменьшить амортизационную стоимость поиска.
  - ▶ Уменьшить сложность функции, например,  $O(\log N) \rightarrow O(1)$ .

# Обобщённый быстрый поиск

- База данных названий городов и их численности.

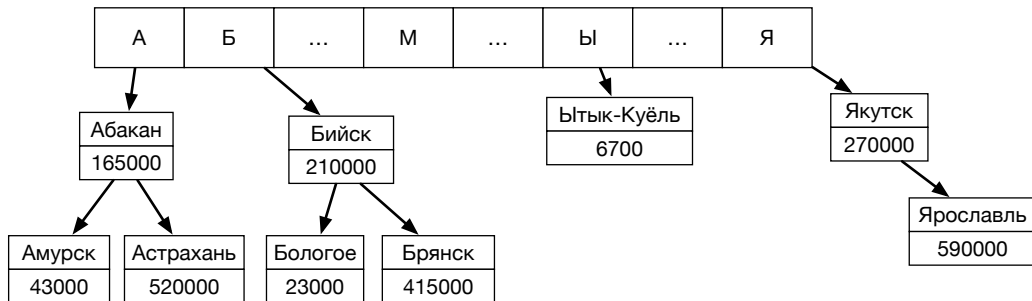


33 связанных списка.



# Обобщённый быстрый поиск

- База данных названий городов и численности их населения.



33 сбалансированных дерева.

# 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  $O(1)$ .

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

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

# Обобщённый быстрый поиск

- При увеличении  $M$

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

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

- Имеется зона оптимальности при  $M \approx N$

# Обобщённый быстрый поиск

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

$$|K_1| \approx |K_2| \approx \dots \approx |K_M|$$

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

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

# Хеш-функции

- Функция преобразования:

$$H(K) \rightarrow V$$

$$|D(V)| = M$$

- Отображение пространства ключей  $K$  на пространство значений  $V$ .
- $M$  — мощность множества пространства значений.

# 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.**

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

где  $L(K)$  — мера длины ключа  $K$ .

- **Равномерность.** Каждое выходное значение равновероятно.

$$P_{H(K_1)} = P_{H(K_2)} = \dots = P_{H(K_M)}$$

- **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.



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.

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

- Совпадение значений функции для разных значений ключа называется **коллизией**.
- Введём  $H^*$  — 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  $\frac{|H^*|}{m}$ .

# 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  $\frac{1}{m}$

# Теорема об универсальном множестве хеш-функций

## Теорема.

- Пусть множество  $Z_p = \{0, 1, \dots, p - 1\}$ , множество  $Z_p^* = \{1, 2, \dots, p - 1\}$ ,  $p$  — 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.

# Теорема об универсальном множестве хеш-функций

## Теорема.

- Пусть множество  $Z_p = \{0, 1, \dots, p - 1\}$ , множество  $Z_p^* = \{1, 2, \dots, p - 1\}$ ,  $p$  — 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.

**Доказательство.** См. книгу Кормена.

# Хеш-функции

- Не универсальная, не не столь уж и отвратительная функция

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

Обратная схема Горнера:

```
unsigned
hash_sum(string const &s, unsigned HASHSIZE)
{
    unsigned sum = 0;
    for (auto c: s) {
        sum <<= 3;
        sum += c;
    }
    return sum % HASHSIZE;
}
```

# Хеш-функции

- Хеш-функция получше

```
unsigned
hash_sedgwick(string const &s, unsigned HASHSIZE)
{
    unsigned h, i, a = 31415, b = 27183;
    for (h = 0, i = 0; i < s.size();
        i++, a = a * b % (HASHSIZE-1)) {
        h = (a * h + s[i]) % HASHSIZE;
    }
    return h;
}
```



# Хеш-функции

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

# Весьма хорошая хеш-функция

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

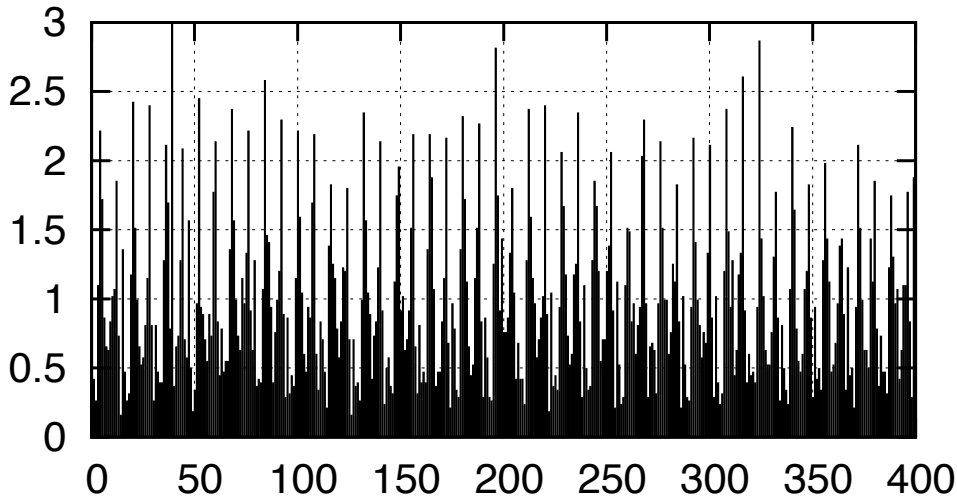
```
uint32 hash(string const &s) {
    uint32 ret = 0xFFFFFFFF;
    for (auto c: s) {
        ret ^= c & 0xFF;
        ret = (ret >> 8) ^ table[ret & 0xFF];
    }
    return ret ^ 0xFFFFFFFF;
}
```

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

# Хеш-функции: исследование свойств

Распределение значений для случайных идентификаторов. Плохая функция.

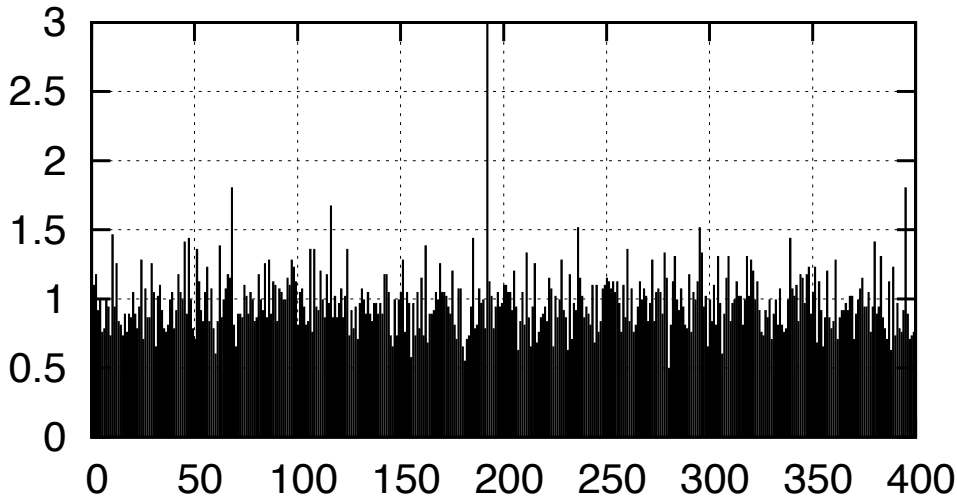
## SUM hash, HASHSIZE=400



# Хеш-функции

Распределение значений для случайных идентификаторов. Плохая функция.

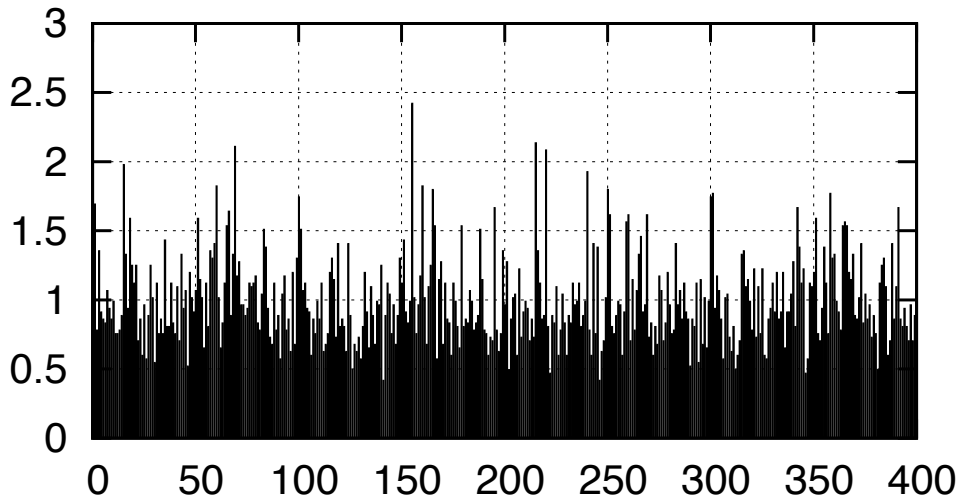
## SUM hash, HASHSIZE=401



# Хеш-функции

Хорошая функция.

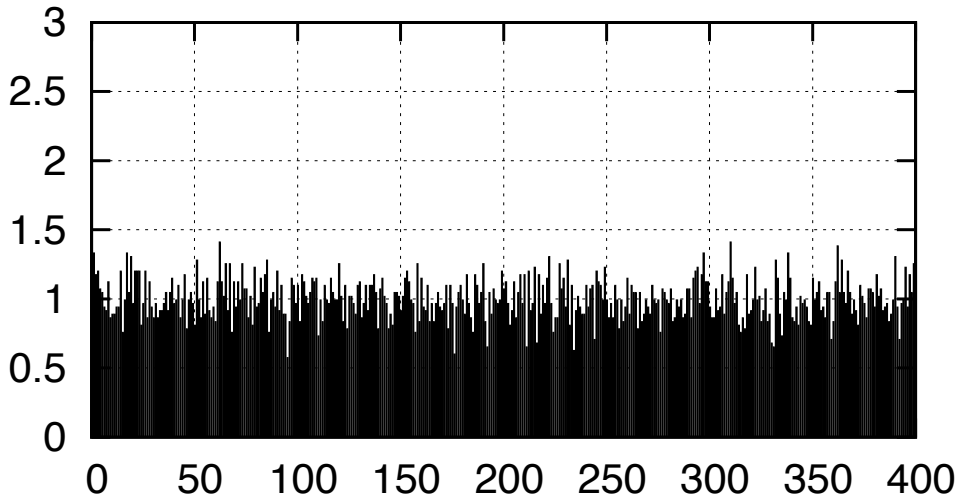
## Sedgewick hash, HASHSIZE=400



# Хеш-функции

Хорошая функция.

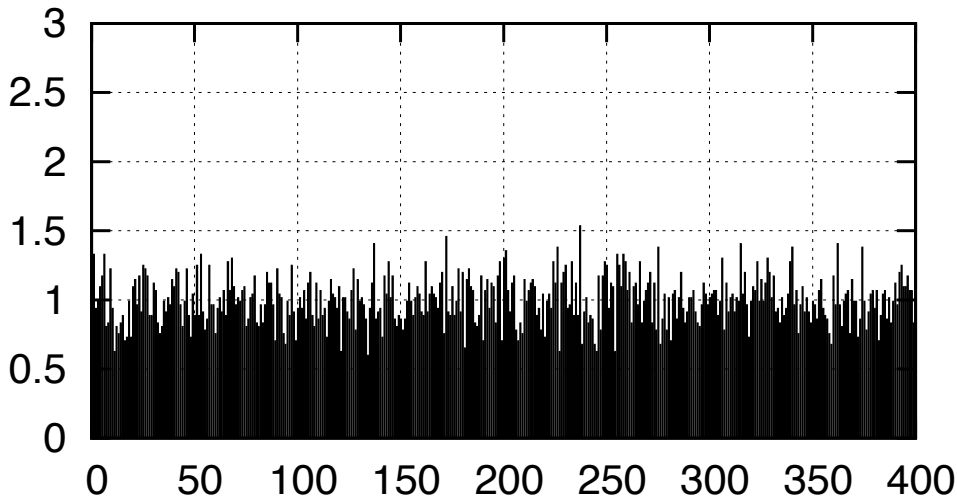
## Sedgwick hash, HASHSIZE=401



# Хеш-функции

Отличная функция.

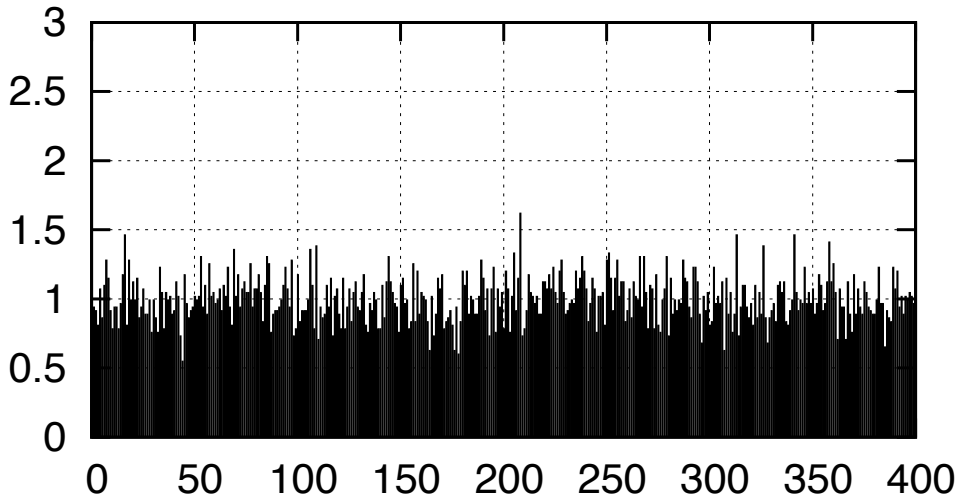
## CRC32 hash, HASHSIZE=400



# Хеш-функции

Отличная функция.

## CRC32 hash, HASHSIZE=401





## 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 по 4096 байт. Вероятность получения неверного ответа при их равенстве есть  $\frac{4096}{10^{20}} \approx 2.5 \cdot 10^{-16}$ .
- Вероятность совпадения значений хорошей 64-битной хеш-функции для двух блоков данных размером в 4096 байт есть  $\frac{4096}{2^{64}} = 2^{-52} \approx 10^{-17.1}$ , 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.

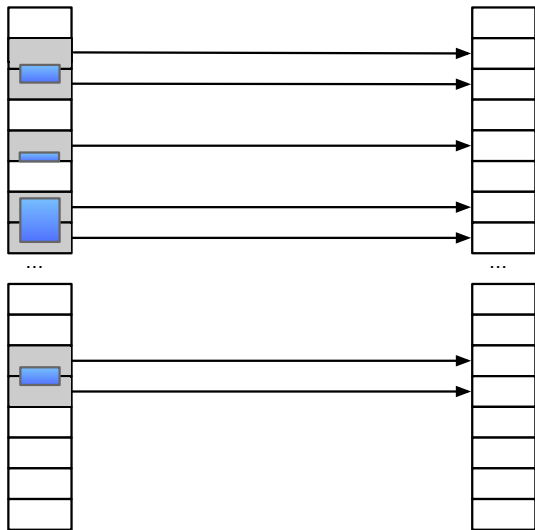
# Синхронизация больших объектов: алгоритм

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 несовпадающим хешем
- 5 клиент запрашивает блоки с несовпадающим хешем

# Синхронизация больших объектов



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

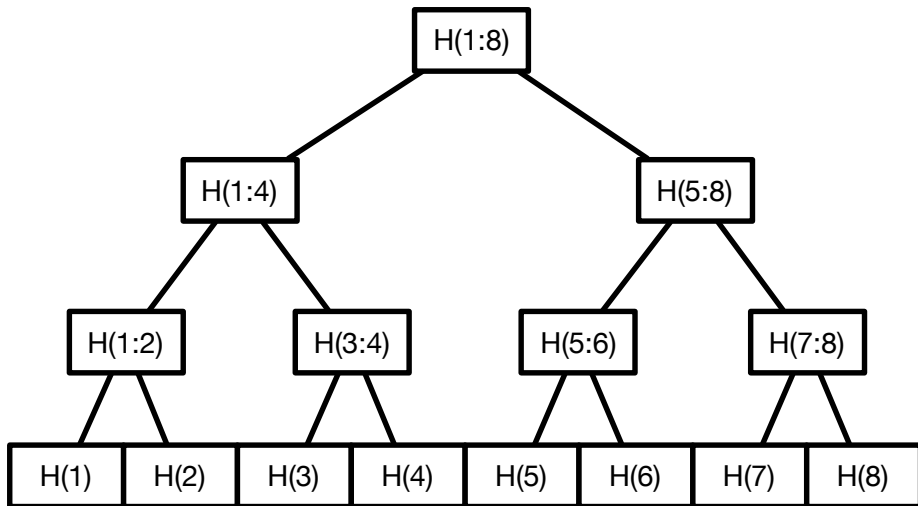
# Синхронизация больших объектов: продвинутый алгоритм

- 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

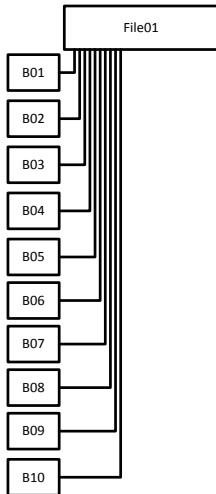
- Пусть  $H(U:V)$  — значение аддитивной хеш-функции от множества блоков от  $U$  до  $V$  включительно.
- Структура данных — дерево, на вершине которого находится узел, содержащий значение хеш-функции от всего объекта.
- Уровнем ниже — два узла, содержащие значения хеш-функции от половины объекта и 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 деревья с обеих 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

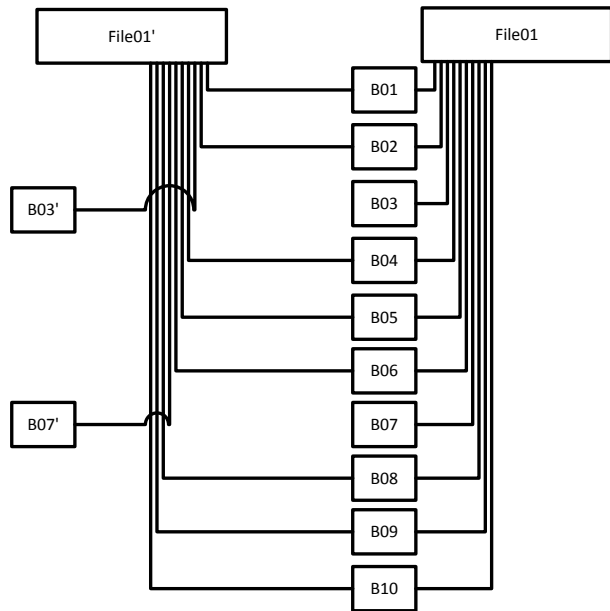
- Имеется блочное хранилище.
- В хранилище копия файла File01, 10 блоков, от B1 до B10.



# Дедупликация

- Изменились блоки В3 и В7.
- Вариант 1: создать копию нового файла, содержащую все 10 блоков.
- 8 блоков — В1, В2 ... будут совпадать.
- Если есть возможность определить, что изменились именно блоки В3 и В7, то в хранилище достаточно передать именно эти блоки и заменить ими старые блоки В3 и В7.
- Старые блоки В3 и В7 сохраняются.
- Наличие новые блоков В3' и В7' 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*.

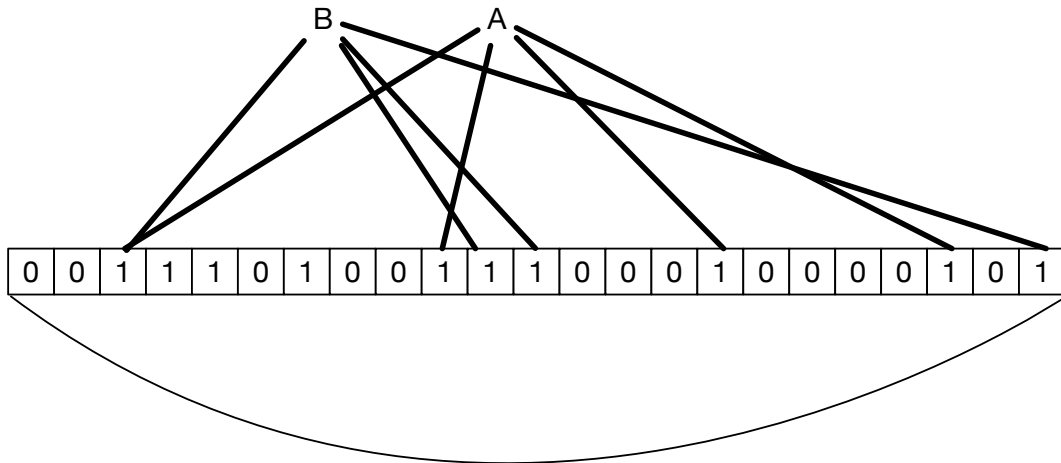
# Вероятностные множества

- 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.

$n=4$

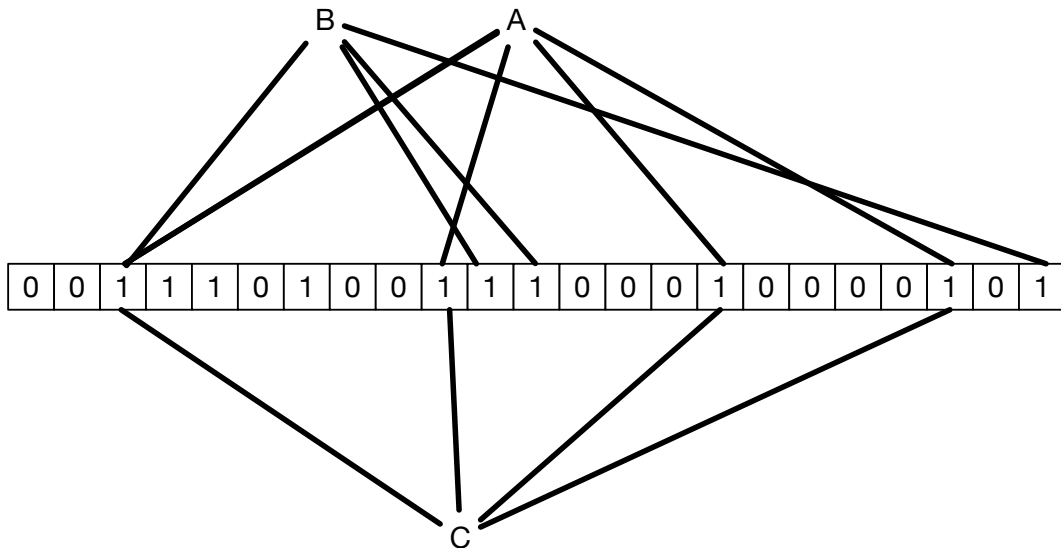


# Фильтр Блума

- **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).

# Фильтр Блума: применение

- Google Chrome: По имени сайта быстрая проверка, не вредоносный ли он.
- Google BigTable: По заданной строке или столбцу базы данных определяет их наличие в таблице. Многократно уменьшает количество запросов к жёсткому диску за данными.
- Распределённые системы хранения: быстрое определение отсутствия требуемых данных.

# Алгоритм Карпа-Рабина



# Поиск подстрок в строке: алгоритм Карпа-Рабина

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

Упростим задачу.

Пусть строки состоят из символов A, B, C, D.  
Отообразим 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  $[k \dots r]$ :

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

# Алгоритм Карпа-Рабина

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

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

$$H(src_{[0,3]}) = 291$$

$$H(src_{[1,4]}) = 183$$

$$H(src_{[2,5]}) = 161$$

$$H(src_{[3,6]}) = 407$$

$$H(src_{[4,7]}) = 206$$

$$H(src_{[5,8]}) = 291$$

$$H(src_{[6,9]}) = 183$$

$$H(src_{[7,10]}) = 411$$

$$H(src_{[8,11]}) = 207$$

$$H(src_{[9,12]}) = 166$$

$$H(src_{[10,13]}) = 283$$

$$H(src_{[11,14]}) = 431$$

# Алгоритм Карпа-Рабина

Хеш-функция для наших строк:

```
unsigned hash(string s, unsigned l, unsigned r,  
              vector<unsigned> const& ptab) {  
    unsigned sum = 0;  
    for (unsigned i = l; i < r; i++) {  
        sum += (s[i]-'A'+1) * ptab[i-l];  
    }  
    return sum;  
}
```

# Алгоритм Карпа-Рабина

Поиск подстроки:

```
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) {
        bool ok = true;
        for (unsigned j = 0; ok && j < s1.size(); j++) {
            if (s1[j] != s2[i+j]) {
                ok = false;
            }
        }
        if (ok) {
            printf("match at: %u\n", i);
        }
    }
}
```

# Алгоритм Карпа-Рабина

Какова сложность кода при условии, что  $N = \text{src.size}()$ ,  $M = \text{pat.size}()$ ?

# Алгоритм Карпа-Рабина

Какова сложность кода при условии, что  $N = \text{src.size}()$ ,  $M = \text{pat.size}()$ ?

$O(N \cdot M)$

Что не так?



# Алгоритм Карпа-Рабина

Какова сложность кода при условии, что  $N = \text{src.size}()$ ,  $M = \text{pat.size}()$ ?

$O(N \cdot M)$

Что не так?

Мы делали много лишних вычислений.

# Алгоритм Карпа-Рабина

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

Попробуем применить индукцию и вычислить, скажем,  $H(s_{[1,4]})$ .

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

Умножим на  $p^1$ :

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

Сравним с

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

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

# Алгоритм Карпа-Рабина

Достаточно вычислить хеш от всех подстрок строки `src`.

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

...

# Алгоритм Карпа-Рабина

```
int karp_rabin(string const &s1, string const &s2, vector<unsigned> const &ptab) {
    unsigned hs1 = hash(s1, 0, s1.size(), ptab);
    vector<unsigned> htab(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) {
            bool ok = true;
            for (unsigned j = 0; j < s1.size(); j++)
                if (s1[j] != s2[i+j])
                    ok = false;
            if (ok)
                return i;
        }
        hs1 *= 5;
    }
    return -1;
}
```

# Алгоритм Карпа-Рабина

Применённая здесь 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);  
    }  
}
```