Лекция 3
Хеш-таблицы

Алгоритмы и структуры данных

Глушенков Д.А.

План лекции

- Хеш-функции.
- Хеш-таблицы.
  - Разрешение коллизий методом цепочек.
  - Разрешение коллизий методом открытой адресации.
    - Линейное пробирование
    - Квадратичное пробирование
    - Двойное хеширование
- Другие сценарии использования хеш-функций.

Быстрый контейнер. Постановка задачи.

Задача — хранить ключи в контейнере. Хотим:

- быстро добавлять (Add)
- быстро удалять (Delete)
- быстро проверять наличие (Has)

<u>Решение 1.</u> Неупорядоченный массив:

- быстрое добавление — $O(1)$
- медленное удаление — $O(n)$
- медленный поиск — $O(n)$

<u>Решение 2.</u> Упорядоченный массив:

- медленное добавление — $O(n)$
- медленное удаление — $O(n)$
- быстрый поиск — $O(\log(n))$

Быстрый контейнер. Постановка задачи.

Частное решение 3.

Пусть ключи — неотрицательные целые числа в диапазоне $[0,..,n-1]$. Будем хранить A – массив bool.

$A[i] = true \iff i$ содержится:

- мгновенное добавление — $O(1)$,
- мгновенное удаление — $O(1)$,
- мгновенный поиск — $O(1)$.

Быстрый контейнер. Хеш-таблица.

Хеш-таблица — массив ключей с особой логикой, состоящей из:

1. Вычисления хеш-функции, которая преобразует ключ поиска в индекс.
2. Разрешения конфликтов, так как два и более различных ключа могут преобразовываться в один и тот же индекс массива.

Отношение порядка над ключами не требуется.

Хеш-функции

Хеш-функция — преобразование по детерминированному алгоритму входного массива данных произвольной длины (один ключ) в выходную битовую строку фиксированной длины (значение).

Результат вычисления хеш-функции называют «хешем».

Коллизией хеш-функции H называется два различных входных блока данных X и Y таких, что $h(X) = h(Y)$.

Хеш-функции

Количество возможных значений хеш-функции не больше M и для любого ключа k:

$$0 \le h(k) < M$$

Важно! Хорошая хеш-функция должна:

1. Быстро вычисляться.
2. Минимизировать количество коллизий.

HASH = рубить, перемешивать.

Хеш-функции

Качество хеш-функции зависит от задачи и предметной области.

Пример плохой хеш-функции.

$$h(k) = [\text{последние 3 цифры } k] = k \ \% \ 1000$$

Такая хеш-функция порождает много коллизий, если множество ключей — цены.

Частые значения:

000, 500, 999, 998, 990, 900.

Хеш-функции. Метод деления.

$$h(n) = n \; mod \; M$$

M определяет размер диапазона значений: $[0, .., M - 1]$. Как выбрать M?

- Если $M = 2^K$, то значение хеш-функции не зависит от старших битов.
- Если $M = 2^8 - 1$, то значение хеш-функции не зависит от перестановки байт.

Хорошо в качестве M брать простое число, далекое от степеней двойки.

Хеш-функции. Метод деления.

Сумма Флетчера - это остаток от деления интерпретируемого как длинное число потока данных на 255.

Пусть G - длинное число потока данных, $B = 2^8 = 256$, $D = B - 1$

$$G \,\%\, D = (x_n * B^n + \ldots + x_1 * B + x_0) \,\%\, D =$$
$$= (x_n * (\ldots) * D + x_n + \cdots + x_1 * D + x_1 + x_0) \,\%\, D =$$
$$= \left((\ldots) * D \,\%\, D + (x_n + \cdots + x_1 + x_0\right) \,\%\, D) \,\%\, D =$$
$$= (x_n + \ldots + x_1 + x_0) \,\%\, D$$

- $(D+1)^n = D^n + \cdots + D + 1 = (\ldots) * D + 1$
- $(a+b) \,\%\, d = (a \,\%\, d + b \,\%\, d) \,\%\, d$

Хеш-функции. Метод умножения.

$$h(k) = [M \cdot \{k \cdot A\}],$$

где { } — дробная часть, [ ] — целая часть,
A – действительное число, $0 < A < 1$,
M определяет диапазон значений: $[0, .., M-1]$.

Кнут предложил в качестве A использовать число, обратное к золотому сечению:

$$A = \phi^{-1} = \left(\frac{\sqrt{5}-1}{2}\right) = 0.6180339887 \ldots$$

Такой выбор A дает хорошие результаты хеширования.

Хеш-функции. Метод умножения.

Хеш-функцию $h(k) = [M \cdot \{k \cdot A\}]$ вычисляют без использования операций с числами с плавающими точками.

Пусть M — степень двойки. $M = 2^p,\ p \le 32$.

Вместо действительного числа A берут близкое к нему $A = \frac{s}{2^{32}} = \frac{2654435769}{2^{32}}$. То есть, $s = 2654435769$.

Тогда $h(k) = \left[2^p \cdot \left\{k \cdot \frac{s}{2^{32}}\right\}\right] = \left[2^p \cdot \left\{\frac{r_1 2^{32} + r_0}{2^{32}}\right\}\right] = \left[2^p \cdot \frac{r_0}{2^{32}}\right] = \left[\frac{r_0}{2^{32-p}}\right] = \left[\frac{r_{01} 2^{32-p} + r_{00}}{2^{32-p}}\right] = r_{01} = \text{старшие } p \text{ бит } r_0.$

Итого, $\boldsymbol{h(k) = \left(k \cdot s \bmod 2^{32}\right) \gg (32 - p)}$.

Хеш-функции строки.

Строка $s = s_0, s_1, \dots, s_{n-1}$.

<u>Вариант 1</u>. $h_1(s) = (s_0 + s_1 a + s_2 a^2 + \cdots + s_{n-1} a^{n-1}) \, mod \; M$.

<u>Вариант 2</u>. $h_2(s) = (s_0 a^{n-1} + s_1 a^{n-2} + \cdots + s_{n-2} a + s_{n-1}) \, mod \; M$.

Число M – степень двойки.

Важно правильно выбрать константу a.

Хотим, чтобы при изменении одного символа, хеш-функция изменялась. То есть, чтобы все значения $s \cdot a \; mod \; M, 0 \le s \le M$ были различны.

Для этого достаточно, чтобы a и M были взаимно простыми.

Хеш-функции строки

$h_2(s)$ вычисляется эффективнее, если использовать метод Горнера:

$$h_2(s) = \Big(\big((s_0 a + s_1)a + s_2\big)a + \cdots + s_{n-2}\Big)a + s_{n-1}.$$

$h_1(s)$ можно вычислять аналогично, но начиная с конца строки.

Но в c-строках известен только указатель на начало строки, а размер строки не известен.

Поэтому удобнее вычислять $h_2(s)$.

Хеш-функция строки

```
// Хеш-функция строки.
int Hash( const char* str, int m )
{
    int hash = 0;
    for( ; *str != 0; ++str )
        hash = ( hash * a + *str ) % m;
    return hash;
}
```

Хеш-функции. Вероятность коллизии.

Парадокс дней рождений.

Сколько необходимо взять человек, чтобы вероятность совпадения дней рождения (число и месяц) хотя бы у двух людей превышала 50 %?

Хеш-функции

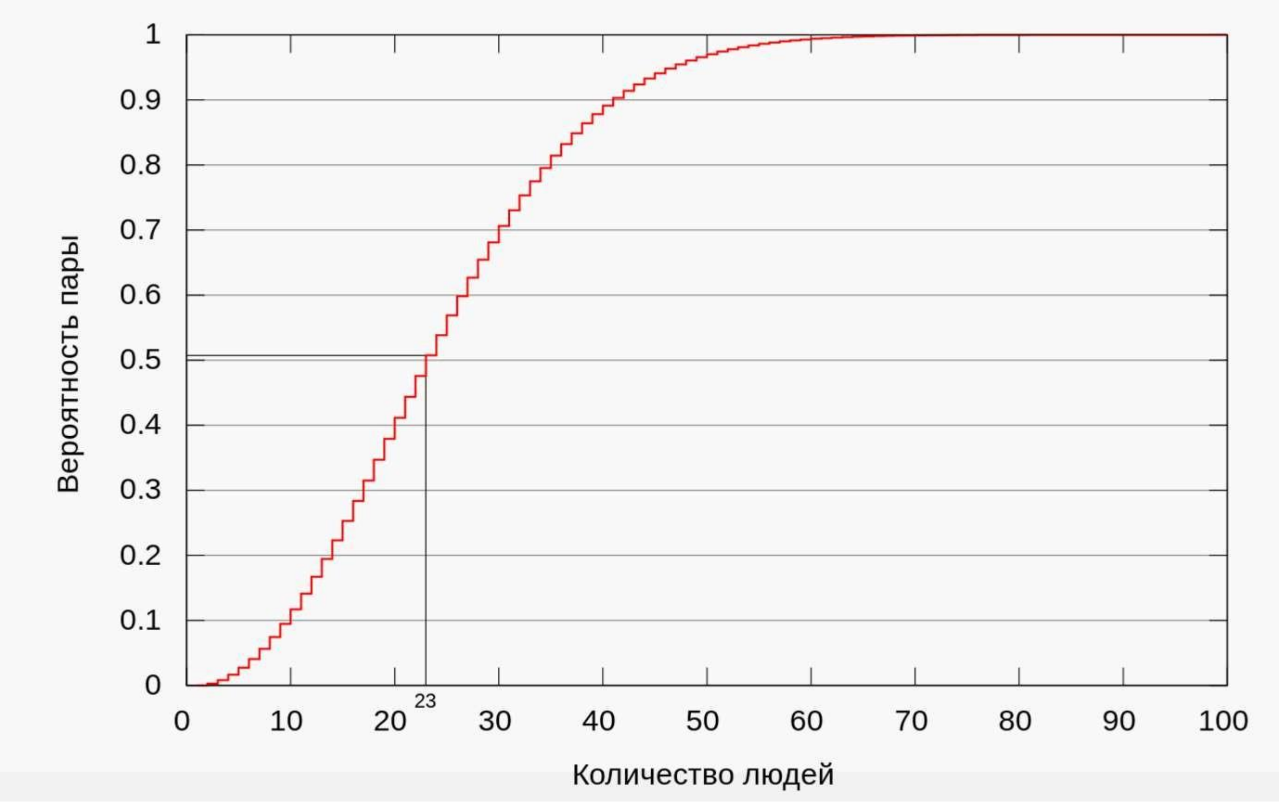
Вычислим вероятность того, что все дни рождения в группе будут различными:

$$\bar{p}(n) = 1 \cdot \left(1 - \frac{1}{365}\right) \cdot \left(1 - \frac{2}{365}\right) \cdot \dots \cdot \left(1 - \frac{n-1}{365}\right) = \frac{365 \cdot 364 \ldots (365-n+1)}{365^n} = \frac{365!}{365^n (365-n)!},$$

Тогда вероятность того, что хотя бы у двух человек из n дни рождения совпадут:

$$p(n) = 1 - \bar{p}(n)$$

Ответ: 23



Хеш-таблицы

При вставке в хеш-таблицу размером 365 ячеек всего лишь 23-х элементов вероятность коллизии уже превысит 50%, при вставке 50 элементов вероятность превысит 97% (если каждый элемент может равновероятно попасть в любую ячейку).

Хеш-таблицы различаются по методу разрешения коллизий.

Основные **методы** разрешения коллизий:

1. Метод цепочек.
2. Метод открытой адресации.

Хеш-таблицы

Хеш-таблица — структура данных, хранящая ключи в таблице. Индекс ключа вычисляется с помощью хеш-функции. Операции: добавление, удаление, поиск.

Пусть хеш-таблица имеет размер M, количество элементов в хеш-таблице — N.

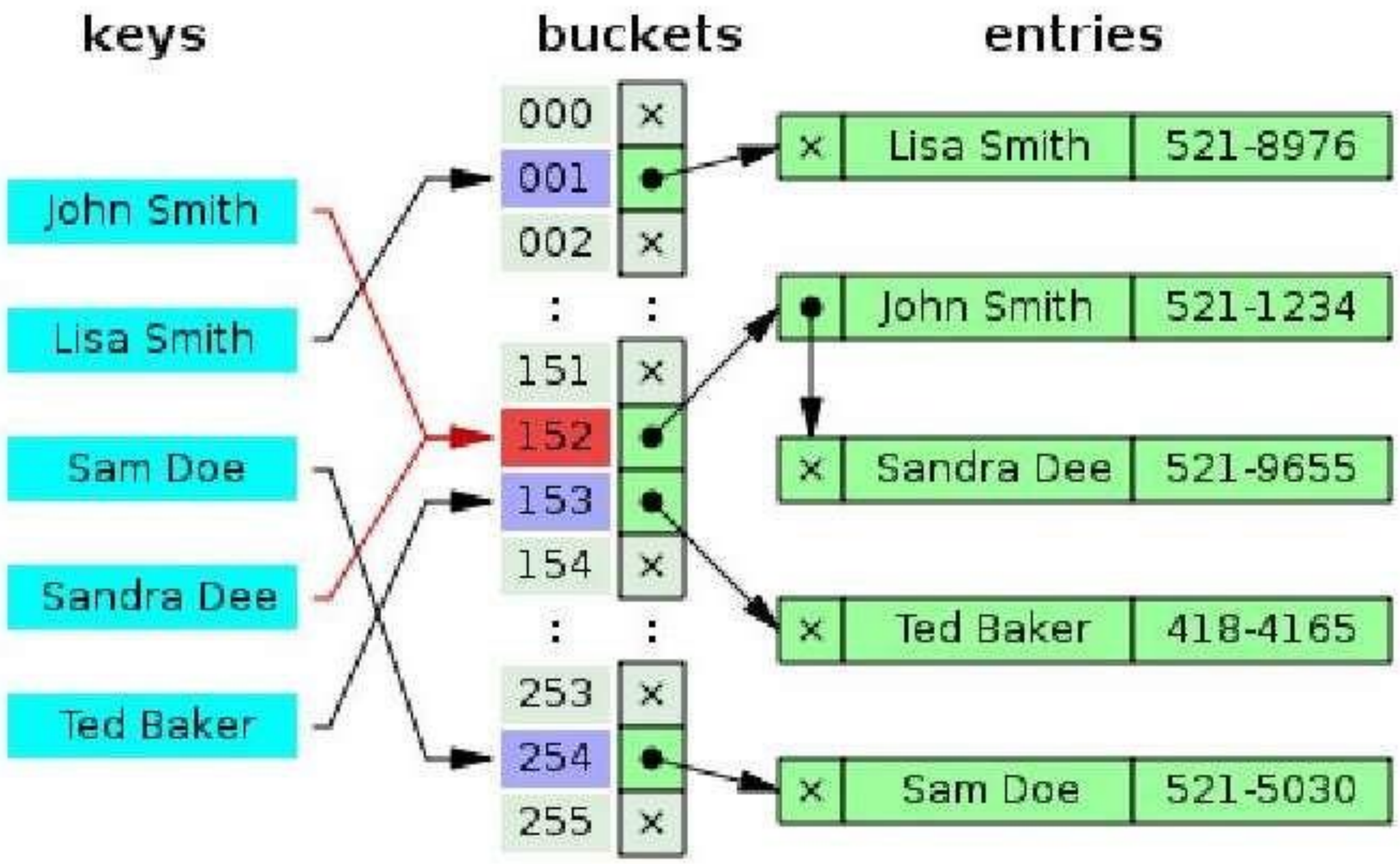
Число хранимых элементов, делённое на размер массива (число возможных значений хеш-функции), называется **коэффициентом заполнения хеш-таблицы** (load factor). Обозначим его $\alpha = \frac{N}{M}$.

Этот коэффициент является важным параметром, от которого зависит среднее время выполнения операций.

Хеш-таблицы. Метод цепочек.

Каждая ячейка массива является указателем на связный список (цепочку).

Коллизии приводят к тому, что появляются цепочки длиной более одного элемента.



Хеш-таблицы. Метод цепочек.

<u>Добавление ключа.</u>

1. Вычисляем значение хеш-функции добавляемого ключа — h.
2. Находим $A[h]$ — указатель на список ключей.
3. Вставляем в начало списка (в конец списка дольше). Если запрещено дублировать ключи, то придется просмотреть весь список.

Время работы:
В лучшем случае – $\boldsymbol{O(1)}$.
В худшем случае
– если не требуется проверять наличие дубля, то $\boldsymbol{O(1)}$,
– иначе – $\boldsymbol{O(N)}$.

Хеш-таблицы. Метод цепочек.

<u>Удаление ключа.</u>

1. Вычисляем значение хеш-функции удаляемого ключа — h.
2. Находим $A[h]$ — указатель на список ключей.
3. Ищем в списке удаляемый ключ и удаляем его.

Время работы:
В лучшем случае – $\boldsymbol{O(1)}$.
В худшем случае – $\boldsymbol{O(N)}$.

Хеш-таблицы. Метод цепочек.

<u>Поиск ключа.</u>

1. Вычисляем значение хеш-функции ключа $-$ h.
2. Находим $A[h]$ $-$ указатель на список ключей.
3. Ищем его в списке. Время работы:

Время работы:
В лучшем случае – $\boldsymbol{O(1)}$.
В худшем случае – $\boldsymbol{O(N)}$.

Хеш-таблицы. Метод цепочек.

Среднее время работы.

Теорема. Среднее время работы операций поиска, вставки (с проверкой на дубликаты) и удаления в хеш-таблице, реализованной методом цепочек — $O(1+\alpha)$, где α — коэффициент заполнения таблицы.

Доказательство. Среднее время работы — математическое ожидание времени работы в зависимости от исходного ключа.

Время работы для обработки одного ключа $T(k)$ зависит от длины цепочки и равно $O(1+N_{h(k)})$,где N_i — длина i-й цепочки. Предполагаем, что хеш-функция равномерна, а ключи равновероятны.

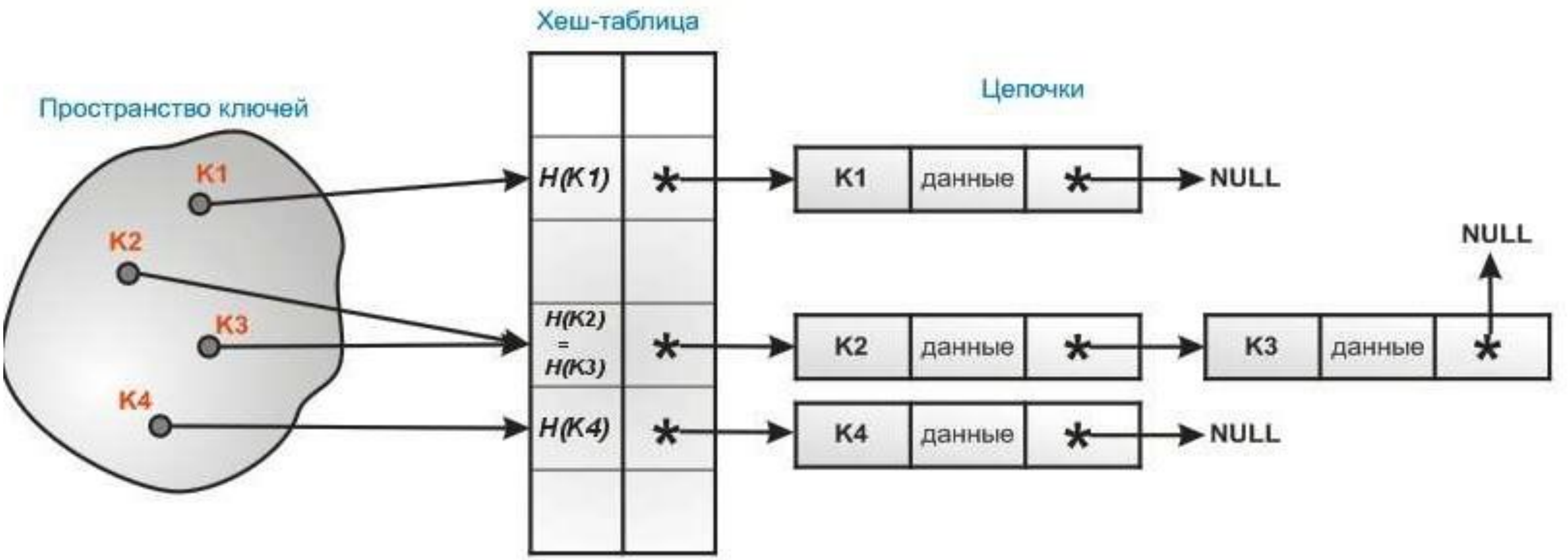
Среднее время работы

$$T_{\text{ср}}(M,N) = M\big(T(k)\big) = \sum_{i=0}^{M-1} \frac{1}{M}(1+N_i) = \frac{1}{M}\sum_{i=0}^{M-1}(1+N_i) = \frac{M+N}{M} = 1+\alpha$$

Хеш-таблицы. Метод цепочек.

```
// Элемент цепочки в хеш-таблице.
template<class T>
struct HashTableNode {
    T Data;
    HashTableNode<T>* Next;
};

// Хеш-таблица.
template<class T, class H>
class HashTable {
public:
    HashTable( int initialSize );
    bool Has( const T& key ) const;
    void Add( const T& key );
    bool Delete( const T& key );
private:
    vector<HashTableNode<T>*> table;
    H hasher;
};
```

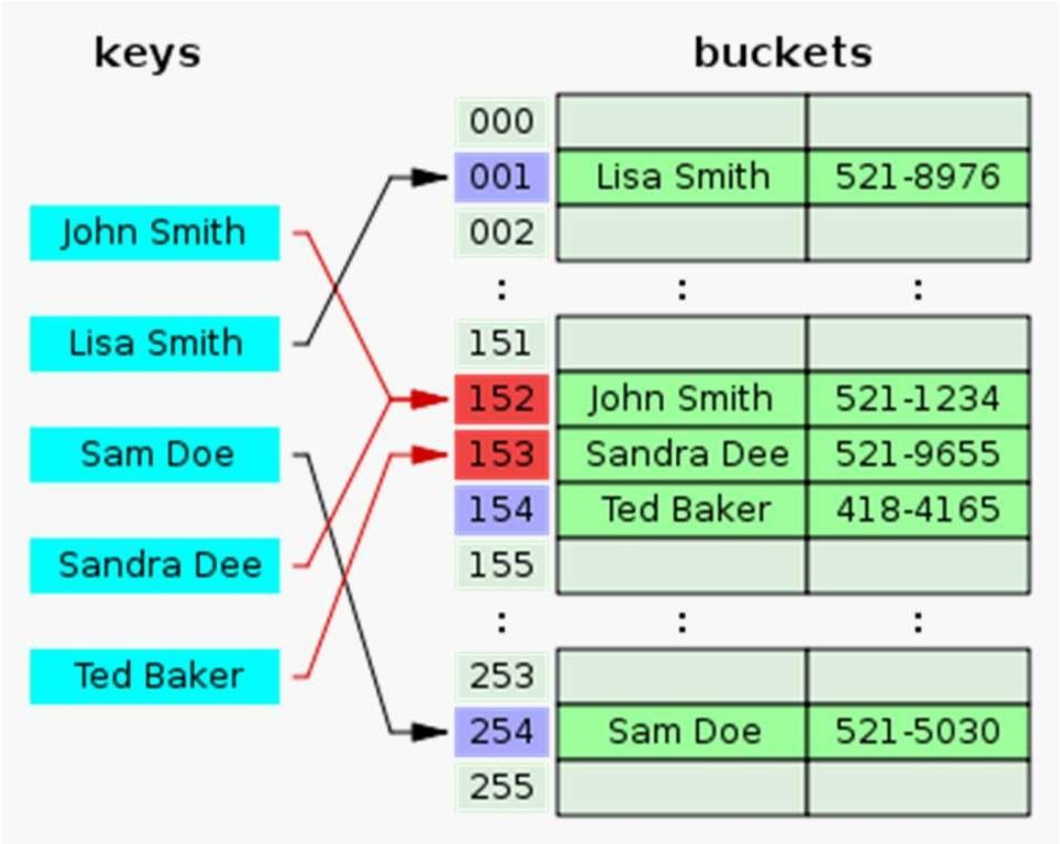
Хеш-таблицы. Метод цепочек.



Хеш-таблицы. Открытая адресация.

Все элементы хранятся непосредственно в массиве. Каждая запись в массиве содержит либо элемент, либо NIL.

При поиске элемента систематически проверяем ячейки до тех пор, пока не найдем искомый элемент или не убедимся в его отсутствии.



Хеш-таблицы. Открытая адресация.

Вставка ключа.

1. Вычисляем значение хеш-функции ключа — h.
2. Систематически проверяем ячейки, начиная от $A[h]$, до тех пор, пока не находим пустую ячейку.
3. Помещаем вставляемый ключ в найденную ячейку.

В п.2 поиск пустой ячейки выполняется в некоторой последовательности. Такая последовательность называется **«последовательностью проб»**.

Хеш-таблицы. Открытая адресация.

Последовательность проб зависит от вставляемого в таблицу ключа. Для определения исследуемых ячеек расширим хеш-функцию, включив в нее номер пробы (от 0).

$$h: U \times \{0, 1, \dots, M-1\} \to \{0, 1, \dots, M-1\}.$$

Важно, чтобы для каждого ключа k последовательность проб

$$\langle h(k, 0), h(k, 1), \dots, h(k, M-1) \rangle$$

представляла собой перестановку множества $\langle 0,1, \dots, M-1 \rangle$, чтобы могли быть просмотрены все ячейки таблицы.

Хеш-таблицы. Открытая адресация.

```
// Вставка ключа в хеш-таблицу (если разрешаем дубли).
void HashTable::Add( const T& k )
{
    for( int i = 0; i < tableSize; ++i ) {
        int j = h( k, i );
        if( IsNil( table[j] ) ) {
            table[j] = k;
            return;
        }
    }
    throw HashTableException( “Overflow” );
}
```

Хеш-таблицы. Открытая адресация.

Поиск ключа.

Исследуется та же последовательность, что и в алгоритме вставки ключа.

Если при поиске встречается пустая ячейка, поиск завершается неуспешно, поскольку искомый ключ должен был бы быть вставлен в эту ячейку в последовательности проб, и никак не позже нее.

Хеш-таблицы. Открытая адресация.

<u>Удаление ключа.</u>

Алгоритм удаления достаточен сложен: нельзя при удалении ключа из ячейки i просто пометить ее значением NIL. Иначе в последовательности проб для некоторого ключа (или некоторых) возникнет пустая ячейка, что приведет к неправильной работе алгоритма поиска.

<u>Решение.</u> Помечать удаляемые ячейки специальным значением «Deleted».

Нужно изменить методы поиска и вставки.

- В методе вставки проверять «Deleted», вставлять на его место, если можем.
- В методе поиска продолжать пробирование при обнаружении «Deleted».

Хеш-таблицы. Открытая адресация.

Вычисление последовательности проб.

Желательно, чтобы для различных ключей k последовательность проб $\langle h(k,0), h(k,1), \dots, h(k,M-1)\rangle$

давала большое количество последовательностей-перестановок множества $\langle 0,1,\dots,M-1\rangle$.

Обычно используются три метода построения $h(k,i)$:

1. Линейное пробирование.
2. Квадратичное пробирование.
3. Двойное хеширование.

Хеш-таблицы. Открытая адресация.

Линейное пробирование.

$$h(k,i) = (h'(k,i) + c\ i)\ mod\ M$$

Основная проблема – кластеризация.

Последовательность подряд идущих занятых элементов таблицы быстро увеличивается, образуя кластер.

Попадание в элемент кластера при добавлении гарантирует «одинаковую прогулку» для различных ключей и проб. Новый элемент будет добавлен в конец кластера, увеличивая его.

Если $h(k_1,i) = h(k_2,j)$, то $h(k_1,i+r) = h(k_2,j+r)$ для всех r.

Хеш-таблицы. Открытая адресация.

<u>Теорема.</u> Если c и M не являются взаимно простыми, то

$$\{c \cdot i \mod M, 0 \le i \le M\} \ne \{0, \dots, M-1\}.$$

<u>Доказательство.</u>

Пусть c и M не являются взаимно простыми. Тогда c и M имеют общий делитель d:

$$c = d \cdot x, M = d \cdot y$$

Рассмотрим результат деления $c \cdot i$ на M (k – частное, r – остаток):

$$c \cdot i = M \cdot k + r,$$

Для любого i остаток от деления также делится на d (k – частное при делении $c \cdot i$ на M, r – остаток от деления):

$$r = c \cdot i - M \cdot k = d \cdot x \cdot i - d \cdot y \cdot k = d(xi - yk)$$

То есть $\{c \cdot i \mod M, 0 \le i \le M\}$ содержит только элементы, кратные d.

Хеш-таблицы. Открытая адресация.

<u>**Теорема.**</u> Если c и M взаимно просты, то

$$\{c \cdot i \mod M, 0 \le i \le M\} = \{0, \dots, M-1\}.$$

<u>Доказательство.</u>

От противного.

Пусть множество $\{c \cdot i \mod M, 0 \le i \le M\}$ имеет меньше M различных элементов. Тогда существуют некоторые x и y, что $cx \equiv cy \pmod M), x < y < M$ (то есть в какие-то 2 различные итерации попадем в одну и ту же ячейку таблицы).

Следовательно, $c \cdot (y - x) = M \cdot u$ (между итерациями y и x полностью обернулись u раз вокруг таблицы M). Из этого следует, что $y - x$ делится на M, так как c и M взаимно простые. Но $0 < y - x < M$. Противоречие.

Хеш-таблицы. Открытая адресация.

<u>Квадратичное пробирование.</u>

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \; mod \; M.$$

Требуется, чтобы последовательность проб содержала все индексы $0, .., M-1$.
Требуется подбирать c_1 и c_2.

При $c_1 = c_2 = 1/2$, проба вычисляется рекуррентно:

$$h(k, i+1) = h(k,i) + i + 1 \; mod \; M.$$

Возникает **<u>вторичная кластеризация.</u>** Проявляется на ключах с одинаковым хеш-значением $h'(\cdot)$.
Если $h(k_1, 0) = h(k_2, 0)$, то $h(k_1, i) = h(k_2, i)$ для всех i.

Соответствует цепочкам в методе цепочек. Разница лишь в том, что в методе открытой адресации эти цепочки могут еще пересекаться.

Хеш-таблицы. Открытая адресация.

Квадратичное пробирование.

Утверждение. Если $c_1 = c_2 = \frac{1}{2}$, а $M = 2^p$, то квадратичное пробирование дает перестановку $\{0, 1, 2, 3, \dots, M-1\}$.

Доказательство. От противного. Пусть существуют i и j, $0 \le i,\ j \le M-1$, для которых

$$\frac{1}{2}i^2 + \frac{1}{2}i \equiv \frac{1}{2}j^2 + \frac{1}{2}j \ (mod\ 2^p).$$

Тогда

$$\frac{1}{2}i^2 + \frac{1}{2}i - \frac{1}{2}j^2 + \frac{1}{2}j = 2^p \cdot D$$
$$i^2 + i - j^2 - j = 2^{p+1} \cdot D,$$
$$(i-j)(i+j+1) = 2^{p+1} \cdot D,$$

Если i и j одинаковой четности, то $i + j + 1$ нечетна, но $i - j$ не может делиться на 2^{p+1}.
Если i и j разной четности, то $i - j$ нечетна, но $i + j + 1$ не может делиться на 2^{p+1}, так как $0 < i + j + 1 < 2^{p+1}$. Противоречие.

Хеш-таблицы. Открытая адресация.

Двойное хеширование.

$$h(k, i) = \left(h_1(k) + ih_2(k)\right) \, mod \; M.$$

Требуется, чтобы последовательность проб содержала все индексы $0, .., M-1$. Для этого все значения $h_2(k)$ должны быть взаимно простыми с M.

- M может быть степенью двойки, а $h_2(k)$ всегда возвращать нечетные числа.
- M простое, а $h_2(k)$ меньше M.

Общее количество последовательностей проб $= O(M^2)$.

Хеш-таблицы. Открытая адресация.

<u>Анализ хеш-таблиц с открытой адресацией.</u>

<u>Теорема.</u> Математическое ожидание количества проб при неуспешном поиске в хеш-таблице с открытой адресацией и коэффициентом заполнения $\alpha = \frac{n}{m} < 1$ в предположении равномерного хеширования не превышает $\frac{1}{1-\alpha}$.
равномерного хеширования не превышает

<u>Без доказательства.</u>

Время работы методов поиска, добавления и удаления:
В лучшем случае — $O(1)$.
В худшем случае — $O(N)$.
В среднем — $O\left(\frac{1}{1-\alpha}\right)$.

Хеш-таблицы. Открытая адресация.

Плюсы:

+ Основное преимущество метода открытой адресации — не тратится память на хранение указателей списка.
+ Нет элементов, хранящихся вне таблицы.

Минусы:

– Хеш-таблица может оказаться заполненной. Коэффициент заполнения α не может быть больше 1.
– При приближении коэффициента заполнения α к 1 среднее время работы поиска, добавления и удаления стремится к N.
– Сложное удаление.

Динамическая хеш-таблица.

Изначально может быть неизвестно количество хранимых ключей.
Коэффициент заполнения может приближаться к 1, а в реализации методом цепочек может быть больше 1.

Среднее время работы для метода цепочек: $O(1+\alpha)$,

для открытой адресации $O\left(\frac{1}{1-\alpha}\right)$.

Требуется динамически увеличивать размер таблицы. Аналогично динамическому массиву.
Процесс увеличения размера хеш-таблицы называется «**перехешированием**».

Динамическая хеш-таблица.

Перехеширование.

1. Создать новую пустую таблицу. Размер новой таблицы $\widetilde{M}$ может быть равен $2 \cdot M$, где M — размер старой таблицы. Если размер таблицы должен быть простым, то следует использовать простое число, близкое к $2 \cdot M$.

2. Проитерировать старую таблицу. Каждый ключ старой таблицы перенести в новую. Для добавления в новую таблицу надо использовать другую хеш-функцию, возвращающую значения от 0 до $\widetilde{M} - 1$.

Динамическая хеш-таблица.

Когда выполнять перехеширование?

Для разных хеш-таблиц следует использовать разные стратегии.

<u>Для хеш-таблиц, реализованных методом цепочек:</u>

Например, когда коэффициент заполнения α достиг $2 - 3$.

<u>Для хеш-таблиц, реализованных методом открытой адресации:</u>

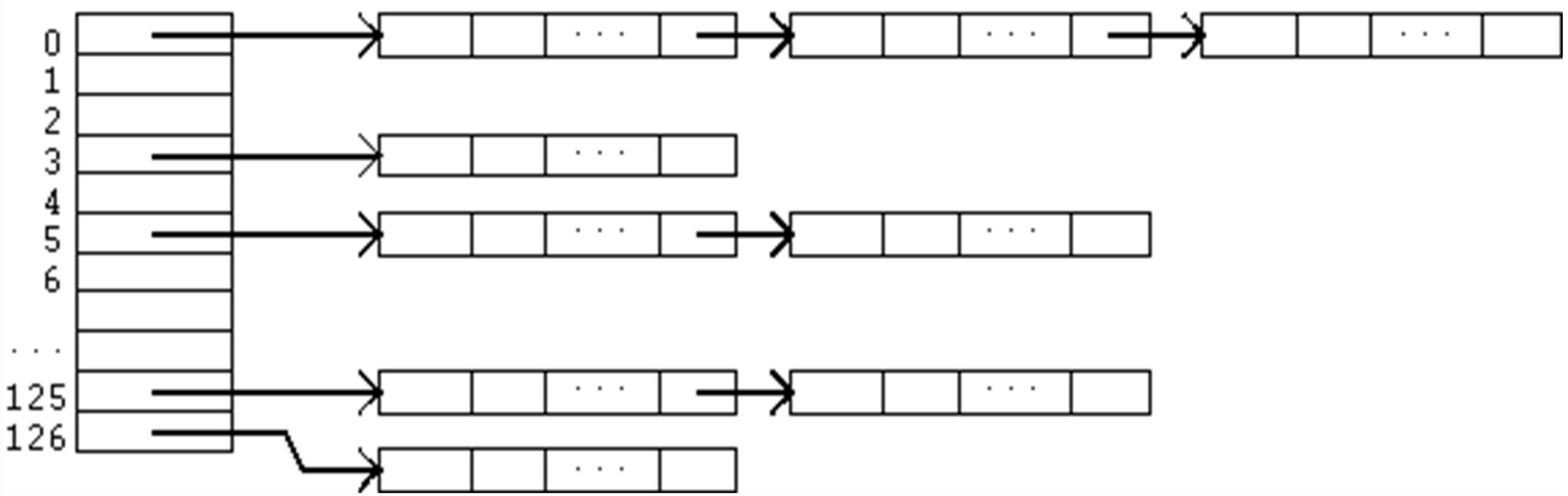
Например, когда α достиг значения $2/3$ или $3/4$.

Хеш-таблицы. Время работы.

	Лучший случай	В среднем. Метод цепочек.	В среднем. Метод открытой адресации.	Худший случай
Поиск	$O(1)$	$O(1+\alpha)$	$O\left(\frac{1}{1-\alpha}\right)$	$O(N)$
Вставка	$O(1)$	$O(1+\alpha)$	$O\left(\frac{1}{1-\alpha}\right)$	$O(N)$
Удаление	$O(1)$	$O(1+\alpha)$	$O\left(\frac{1}{1-\alpha}\right)$	$O(N)$

Хеш-таблицы. Более экзотические варианты.

- Рандомное пробирование:
  $h(k, i) = h(k) + r_i$, где r_i — элемент псевдо-рандомной последовательности, сгенерированной для заданного seed.
- Цепочка bucket-ов.



- https://habr.com/ru/companies/vk/articles/323242/

Контрольная сумма (checksum).

Функции вычисления контрольной суммы также являются хеш-функциями.

Контрольная сумма:

- Предназначена для проверки целостности данных. Считаем контрольную сумму полученных данных, сравниваем с ожидаемым значением.
- Не предназначена для защиты данных от вмешательства злоумышленников (легко взломать).
- Используется в протоколах передачи данных (TCP/IP), системах хранения и резервирования данных (RAID).

Простейший вариант — суммирование значений байт.

Сообщение: $12, 34, 56$

Контрольная сумма: $12 + 34 + 56 = 102$

Checksum. Как устроен штрихкод.

UPC (Universal Product Code) — американский стандарт штрихкодов. Представляет собой 30 вертикальных черточек различной ширины, разделенных пропусками различной ширины, и набор цифр под ними.

У черточек 4 варианта толщины – x1, x2, x3, x4 относительно самой тонкой. Точное значение толщины не регламентируется, только пропорции. Те же самые варианты и у пропусков между черточками. Это позволяет наносить штрихкоды разного размера.

Checksum. Как устроен штрихкод.

Что же видит сканер штрихкодов?

10100011010110001001100100011010001101000110101010111001011001101101100100111011001101000100101

Черточке, в зависимости от толщины, сопоставляется последовательность единичных битов — 1, 11, 111 или 1111.
Пропускам сопоставляются нулевые биты: 0, 00, 000, 0000.

Любой UPC штрихкод всегда начинается и заканчивается одним и тем же набором из 3-х бит: 101 (две тонкие линии по краям штрихкода). По ним калибруется сканер штрихкодов – определяет единичную толщину черточки и пропуск

Checksum. Как устроен штрихкод.

UPC штрихкод — последовательность из 95 бит.

Биты	Назначение
101	Левый защитный код
0001101 0110001 0011001 0001101 0001101 0001101	Левый блок цифр
01010	Центральный защитный код
1110010 1100110 1101100 1001110 1100110 1000100	Правый блок цифр
101	Правый защитный код

Checksum. Как устроен штрихкод.

Но и это еще не все: левый и правый блоки цифр кодируются по-разному.

- Коды цифр слева всегда начинаются с 0 и заканчиваются на 1.
- Коды цифр справа всегда начинаются с 1 и заканчиваются на 0.
- Коды цифр слева и справа получаются друг из друга инвертированием битов.
- Оба кода визуализируются двумя черточками и двумя пропусками.
- В цифрах слева число 1 всегда нечетное, в цифрах справа — всегда четное.

Коды цифр слева

0001101 = 0	0110001 = 5
0011001 = 1	0101111 = 6
0010011 = 2	0111011 = 7
0111101 = 3	0110111 = 8
0100011 = 4	0001011 = 9

Коды цифр справа

1110010 = 0	1001110 = 5
1100110 = 1	1010000 = 6
1101100 = 2	1000100 = 7
1000010 = 3	1001000 = 8
1011100 = 4	1110100 = 9

Checksum. Как устроен штрихкод.

Цифры под штрихкодом дублируют те, что закодированы в штрихкоде.

- Первая цифра — префикс, описывающий тип штрихкода. 0 — обычный, 2 — товар с варьирующимся весом, 5 — купон и т.п.
- Следующие 5 цифр — код производителя.
- Следующие 5 цифр — код товара у данного производителя, имеет смысл только в связке с кодом производителя.
- Последняя цифра нужна для проверки контрольной суммы.

0 51000 01251 7

Checksum. Как устроен штрихкод.

Как посчитать контрольную сумму.

- Обозначим первые 11 цифр штрихкода буквами
  A BCDEF GHIJK.
- Вычислим значение 3 x (A + C + E + G + I + K) + (B + D + F + H + J) и вычтем его из ближайшего бóльшего числа, кратного 10. Результат должен совпасть с 12-й цифрой штрихкода.

$$3 \times (0 + 1 + 0 + 0 + 2 + 1) + (5 + 0 + 0 + 1 + 5) = 3 \times 4 + 11 = 23$$

$$30 - 23 = 7$$

0 51000 01251 7

Checksum. Как устроен штрихкод.

Штрихкоды можно читать вверх ногами. Если в первой считанной цифре число единичных битов четное, значит читаем код вверх ногами.

В таком случае будем декодировать по реверсированным таблицам кодов. Ни один из этих реверсированных кодов не совпадает с обычными кодами. Никакой двусмысленности.

Коды цифр справа, наоборот

0100111 = 0	0111001 = 5
0110011 = 1	0000101 = 6
0011011 = 2	0010001 = 7
0100001 = 3	0001001 = 8
0011101 = 4	0010111 = 9

Коды цифр слева, наоборот

1011000 = 0	1000110 = 5
1001100 = 1	1111010 = 6
1100100 = 2	1101110 = 7
1011110 = 3	1110110 = 8
1100010 = 4	1101000 = 9

CRC.

CRC (Cyclic redundancy check) — циклически избыточный код,.

CRC = сообщение % полином

Возьмём исходное сообщение: $K(x) = k_0 + k_1 x + \ldots + k_n x^n$,
И порождающий многочлен: $P(x) = p_0 + p_1 x + \ldots + p_m x^m$,

<u>Поделим исходное сообщение на многочлен с остатком:</u>

$K(x) = A(x) \cdot P(x) + R(x)$,

$A(x)$ — частное, $R(x)$ — остаток, $degR < degP = m$.

$$R(x) = r_0 + r_1 x + \ldots + r_{m-1} x^{m-1}$$

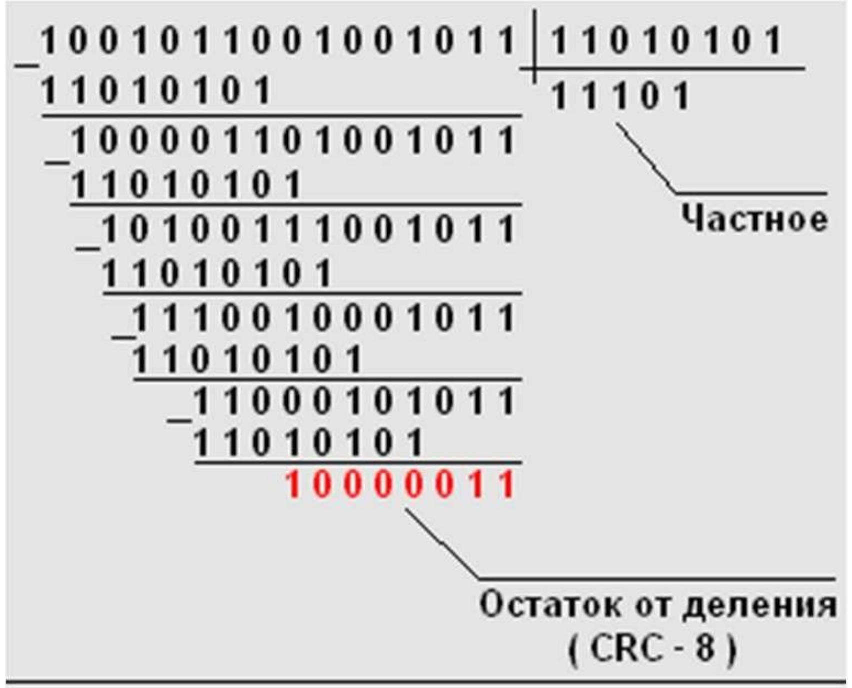
Все коэффициенты в поле Z_2.

CRC.

Пример расчёта CRC-8:

Исходный массив данных: 1001 0110 0100 1011.

Порождающий многочлен: 1101 0101.



Пример расчета контрольной суммы CRC - 8

CRC.

Обычно при вычислении CRC исходное сообщение умножается на x^m:

$$H_p(K)(x) = K(x) \cdot x^m \; mod \; P(x)$$

Для разных стандартов CRC используются многочлены разных степеней, с разными коэффициентами.

CRC стандарт	Многочлен
CRC-1	$x+1$
CRC-5-USB	x^5+x^2+1
CRC-8	$x^8+x^7+x^6+x^4+x^2+1$
CRC-16(-IBM)	$x^{16}+x^{15}+x^2+1$
CRC-32-IEEE 802.3	$x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x+1$
CRC-64-ISO	$x^{64}+x^4+x^3+x+1$

CRC.

Типичная эффективная реализация (CRC-32):

```
crc = CRCINIT
while( bufflen-- )
      crc = t[(crc ^ *buff++) & 0xFF] ^ (crc >> 8);
```

CRC-32 используется в:

- TCP/IP
- Zip/RAR

CRC.

Алгоритмы семейства CRC:

- Эффективно вычисляются в реальном времени (используют только двоичные сдвиги и XOR), что находит широкое применение в микроконтроллерах и встраиваемых системах.
- Отлично ловят типовые ошибки в битах, привносимые в процессе передачи или при хранении данных: измененные значения отдельных битов
- Плохо справляется с ошибками, связанными с измененным порядком битов.
- Не предназначены для защиты от злоумышленника

Криптографические хеш-функции.

Хеш-функции удовлетворяющие требованиям:

1.Стойкость к поиску первого прообраза — отсутствие эффективного полиномиального алгоритма вычисления обратной функции, т.е. нельзя восстановить текст m по известному хеш-значению $H(m)$ за реальное время (необратимость).

2.Стойкость к поиску второго прообраза — вычислительно невозможно, зная сообщение m и его хеш-значение $H(m)$, найти такое другое сообщение $m' \neq m$, чтобы $H(m) = H(m')$.

3.Стойкость к коллизиям — нет эффективного полиномиального алгоритма, позволяющего найти два разных сообщения с одинаковыми хеш-значениями.

Криптографические хеш-функции.

MD1, MD2, MD3, MD4, MD5, MD6, SHA-1, SHA-2 , SHA-3 — известные криптографические хеш-функции/семейство хеш-функций.

MD = Message Digest. Один из самых популярных – MD5 – 128-битный алгоритм хеширования. Разработан Рональдом Л. Ривестом в 1991 г. Использует битовые операции с блоками длины 128.

SHA = Secure Hash Code. Один из самых популярных – SHA-256.

Важная особенность криптографической хеш-функции – лавинный эффект. Замена одного символа приводит к полному изменению значения хеша:

MD5("md5") = 1BC29B36F623BA82AAF6724FD3B16718.

MD5("md4") = C93D3BF7A7C4AFE94B64E30C2CE39F4F

Криптографические хеш-функции.

Практическое использование:

- Проверка целостности
- Поиск дублей
- Проверка парольной фразы
- Цифровая подпись
- Блокчейн

MD5

Шаг 1. Выравнивание потока

Сначала к концу данных дописывают единичный бит. Затем добавляют некоторое число нулевых байт, чтобы длина данных стала сравнима с 448 по модулю 512

Шаг 2. Добавление длины сообщения

В конец данных дописывают 64-битное представление длины данных

Шаг 3. Инициализация буфера

Для вычислений инициализируются 4 переменных размером по 32 бита:

```
A = 01 23 45 67; // 67452301h
B = 89 AB CD EF; // EFCDAB89h
C = FE DC BA 98; // 98BADCFEh
D = 76 54 32 10. // 10325476h
```

MD5

Шаг 4. Вычисление в цикле

Определяем 4 вспомогательные функции:

- $F(x, y, z) = (x \,\&\, y) \mid (\sim x \,\&\, z)$
- $G(x, y, z) = (x \,\&\, z) \mid (y \,\&\, \sim z)$
- $H(x, y, z) = x \wedge y \wedge z$
- $I(x, y, z) = y \wedge (x \mid \sim z)$

Инициализируем таблицу констант $T[1..64]$:

$$T[i] = int(4294967296 * abs(\sin(i)))$$

Каждый 512-битный блок проходит 4 этапа вычислений по 16 раундов. Для этого блок представляется в виде массива X из 16 слов по 32 бита.

MD5

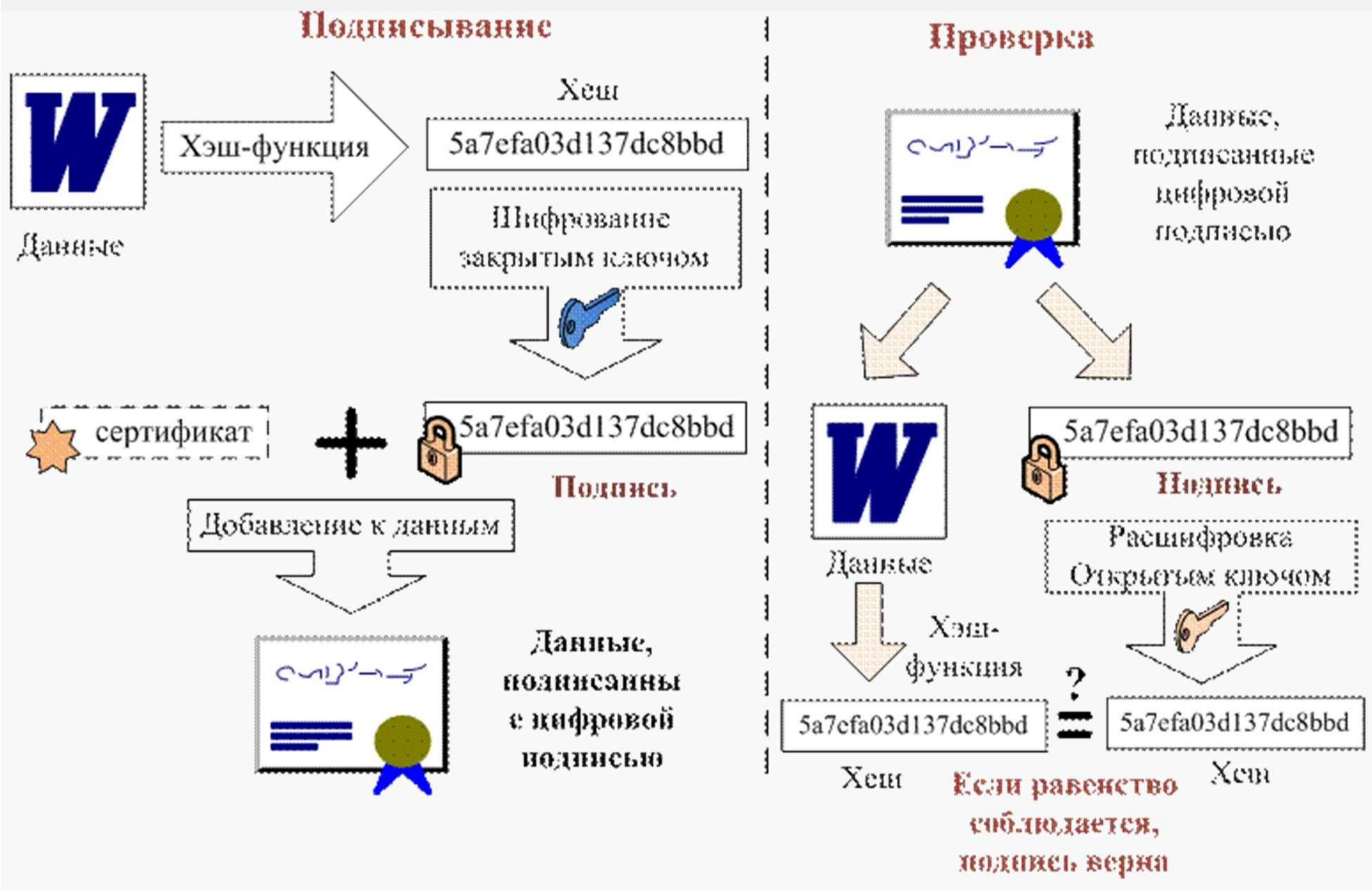
```
tempA = A
tempB = B
tempC = C
tempD = D

/* [abcd k s i] a = b + ((a + F(b,c,d) + X[k] + T[i]) << s). */

[ABCD 0 7 1][DABC 1 12  2][CDAB 2 17  3][BCDA  3 22 4]
[ABCD 4 7 5][DABC 5 12  6][CDAB 6 17  7][BCDA  7 22 8]
[ABCD  8 7  9][DABC  9 12 10][CDAB 10 17 11][BCDA 11 22 12]
[ABCD 12 7 13][DABC 13 12 14][CDAB 14 17 15][BCDA 15 22 16]

/* ещё 3 аналогичных этапа с вызовом функций G, H, I */

A += tempA
B += tempB
C += tempC
D += tempD
```

MD5

История взлома MD5:

- В 1996 году Ганс Доббертин нашёл псевдоколлизии в MD5, используя определённый инициализирующий буффер (ABCD).
- В 2004 году китайские исследователи Ван Сяоюнь, Фен Дэнгуо, Лай Сюэцзя и Юй Хунбо объявили об обнаруженной ими уязвимости в алгоритме, позволяющей за небольшое время (1 час на кластере IBM p690) находить коллизии.
- В 2005 году Ван Сяоюнь и Юй Хунбо из университета Шаньдуна в Китае опубликовали алгоритм, который может найти две различные последовательности в 128 байт, которые дают одинаковый MD5-хеш.
- В 2006 году чешский исследователь Властимил Клима опубликовал алгоритм, позволяющий находить коллизии на обычном компьютере с любым начальным вектором (A, B, C, D) при помощи метода, названного им «туннелирование».

Цифровая подпись DSA.



Цифровая подпись DSA.

Как генерируются ключи:

- Берем два огромных простых числа – p и q.
- Из диапазона $[1, q - 1]$, случайно выбираем число x – приватный ключ.
- Вычисляем публичный ключ $y = g^x \, mod \, p$, где g – предопределенная константа, называемая «генератором».
- Публичный ключ y и параметры p, q, g – в открытом доступе, приватный ключ x ни в коем случае нельзя раскрывать.

Дерево Меркла.

Древовидная структура данных, позволяющая эффективно проверять целостность больших наборов данных.

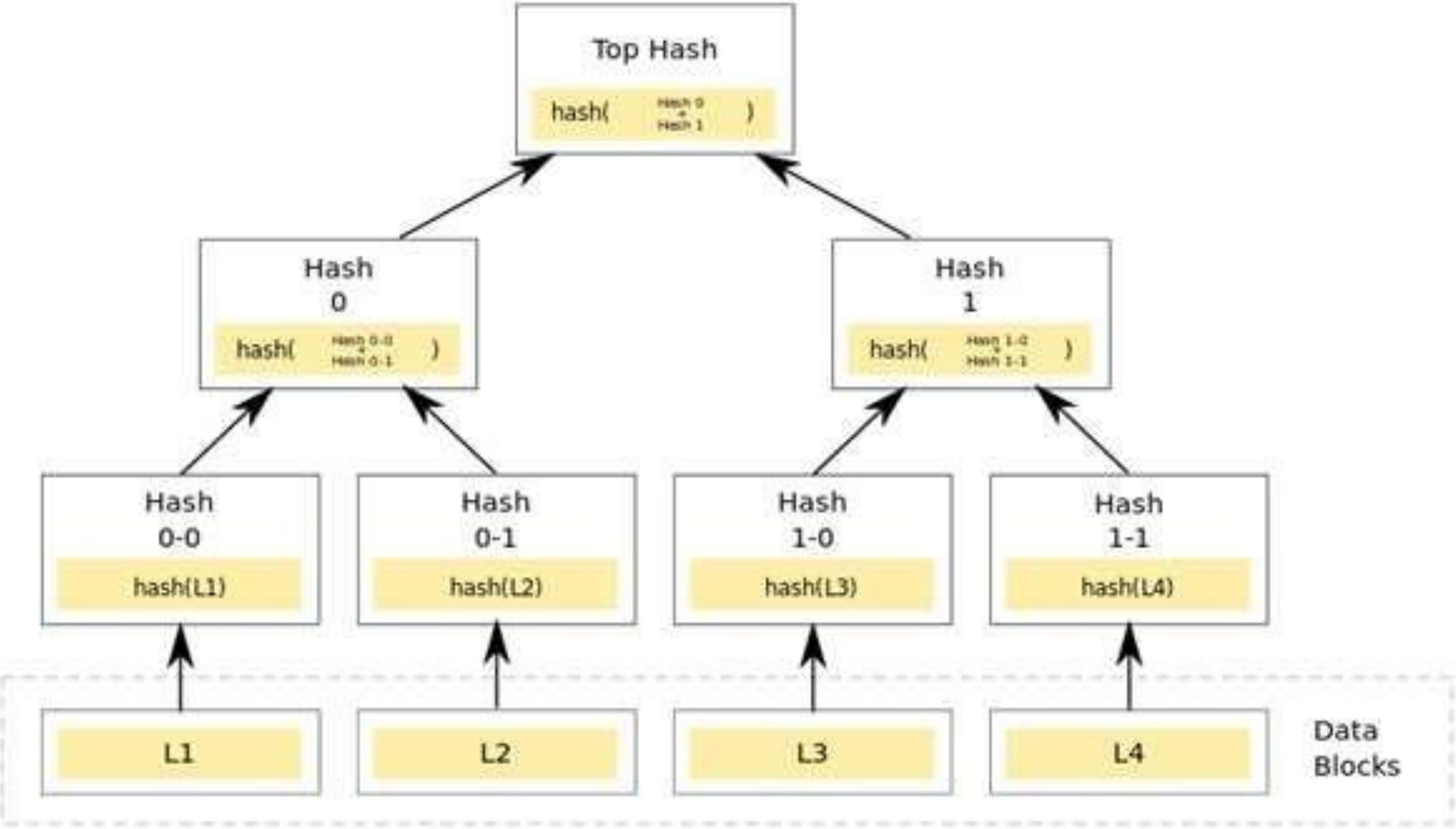
- Набор данных разбиваем на блоки. Например, 1 файл – 1 блок. В случае Bitcoin 1 транзакция – 1 блок.
- Каждый блок хэшируем криптографической хэш-функцией (например, SHA-256). Это порождает листья дерева Меркла.
- Берем попарно соседние листья, считаем хэш от их хэшей, создаем общий родительский узел.
- Повторяем процесс для получившихся родительских узлов, пока не останется единственный узел – корень дерева Меркла.

Корень дерева Меркла хранит «отпечаток» всего набора данных. Малейшее изменение в них приведет к изменению хэшей по всей цепочке до корня. Дерево Меркла позволяет за логарифмическое время найти виновный блок.

Дерево Меркла.

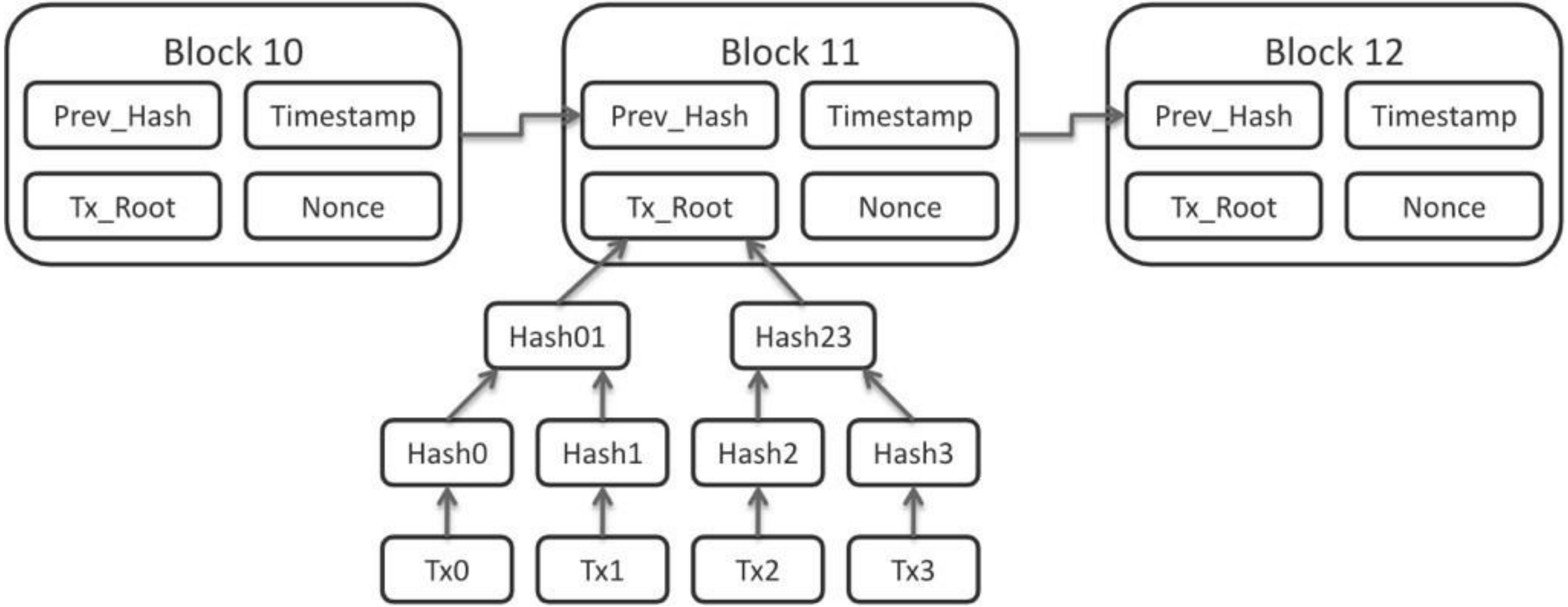
Деревья Меркла используются в:

- Файловых системах для проверки целостности файлов
- Распределённых БД для быстрой синхронизации копий
- Блокчейнах для упрощенной верификации платежей в «легких клиентах»



Блокчейн.

Блокчейн — цепочка криптографически связанных блоков.



Блокчейн. Bitcoin.

Упрощенное представление блока в Bitcoin:

```
struct Transaction {
    std::string sender;
    std::string receiver;
    double amount;
};

struct Block {
    int index;
    std::string previousHash;
    std::string hash;
    time_t timestamp;
    std::vector<Transaction> transactions;
    int nonce;
    std::string merkleRoot;
};
```

Блокчейн.

Пример современного хеша блока BTC – **0000000000000000000**526273c1abbdb82cc3f7964b5c287193eeaf0f86d14b3

Хешируются с помощью SHA-256 данные:

- Хеш списка добавленных транзакций до 1Мб
- Хеш предыдущего блока
- Timestamp
- Nonce (соль, подбирается)

Если значение меньше порогового значения, то блок может быть добавлен в цепочку.

Порог = Максимум / Сложность.

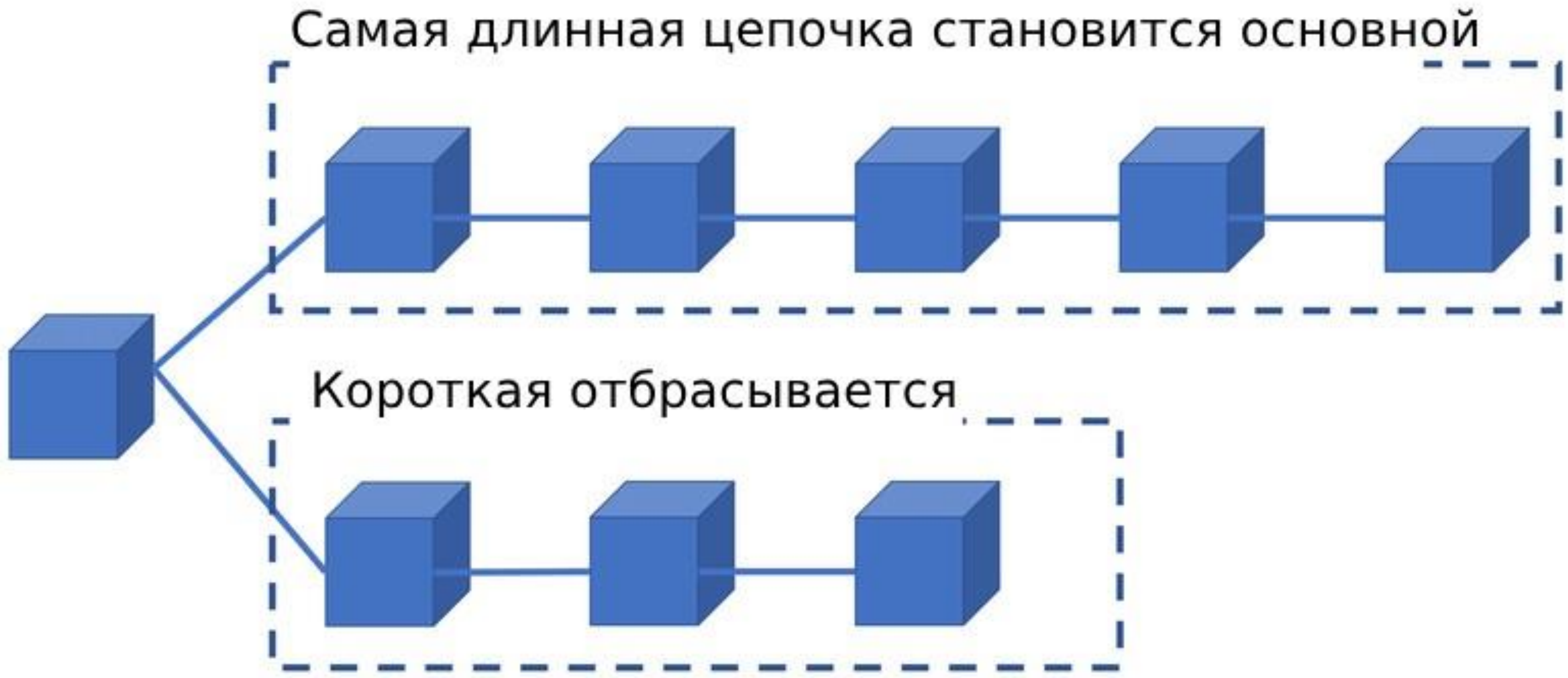
Вычисляется на основе истории так, чтобы очередные блоки находились в среднем раз в 10 минут.

Блокчейн. Bitcoin. Как продлевается цепочка блоков.

- Пользователь инициирует транзакцию и оповещает об этом узлы сети Bitcoin
- Узлы, получив новую транзакцию, верифицируют ее. Если она корректна, то добавляется в пул памяти (mempool) данного узла. Там хранятся неподтвержденные транзакции.
- Транзакции не равноценны: они приоритезируются по дате создания, размеру, а также величине комиссии для майнеров.
- Майнеры (специализированные узлы сети) ходят в mempool и набирают транзакции для включения в новый блок.
- Создание нового блока – очень вычислительноемкая операция, занимающая время. Майнеры стремятся первыми вычислить новый блок. Если им это удается, они оповещают об этом узлы сети, отправляют блок на верификацию.
- Если остальные узлы сети подтвердят валидность блока, локальная копия блокчейна продлевается с помощью этого блока.

Блокчейн. Bitcoin. Как продлевается цепочка блоков.

Что если в сети Bitcoin появится несколько вариантов продолжения цепочки блоков? Это допустимая ситуация, ведь информация о новом блоке может распространяться с задержкой. Работа по вычислению блоков не останавливается, каждый продлевает ту цепочку, которую считает актуальной. Но рано или поздно обнаруживается конфликт и:



Блокчейн биткоина.

В блокчейне биткоина по состоянию на 27.10.23:

- находится 813 981 блок
- средний размер блока 3309 транзакций
- общий размер блокчейна 491.50 Гб.
- за последние сутки сгенерировано 150 блоков (~1 блок в 9.5 минут)
- за последние сутки выполнено 431 304 транзакции

https://bitinfocharts.com/ru/bitcoin/

Спасибо за внимание!

Дмитрий Глушенков