Хеши

что это вообще такое и зачем они нужны

Универсальное семейство хеш-функций

Универсальное семейство хеш-функций

Пусть **H** - множество, отображающее ключи в диапазон **{0, 1, … , m - 1}**

Такое множество называется ***универсальным***, если для каждой пары ключей **k1 != k2** количество хеш-функций **h ∈ H**, для которых **h(k1) == h(k2)**, не превышает **|H| / m**

Универсальное семейство хеш-функций

Пусть **H** - множество, отображающее ключи в диапазон **{0, 1, … , m - 1}**

Такое множество называется ***универсальным***, если для каждой пары ключей **k1 != k2** количество хеш-функций **h ∈ H**, для которых **h(k1) == h(k2)**, не превышает **|H| / m**

Более грубо говоря, для каждой фиксированной пары ключей количество хеш-функций, в которых возникают коллизии для этих ключей, не должно превышать |H| / m

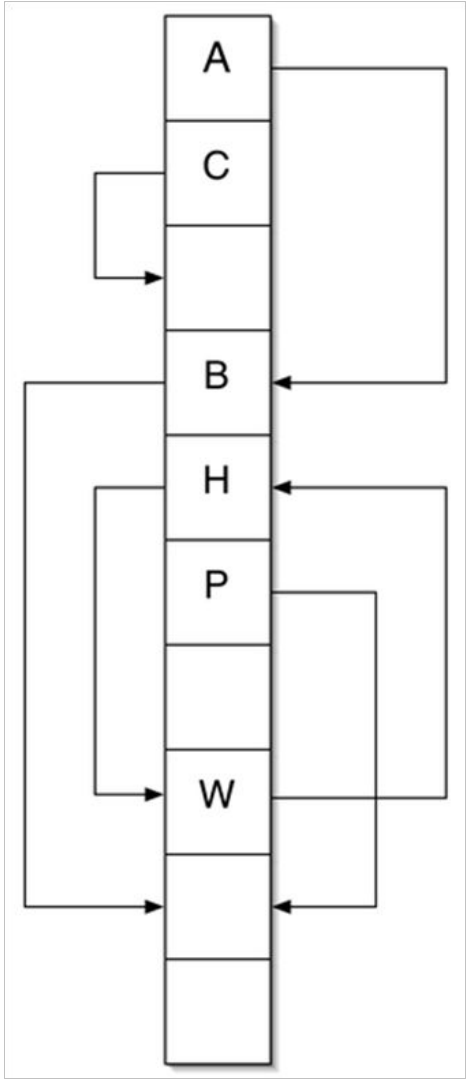
Метод КУКУШКИ

Хеширование кукушкой

Две вариации:

- Две независимые хеш функции
- Две таблицы для двух хеш – функций

Все хеш-функции берутся из универсального семейства хеш-функций!



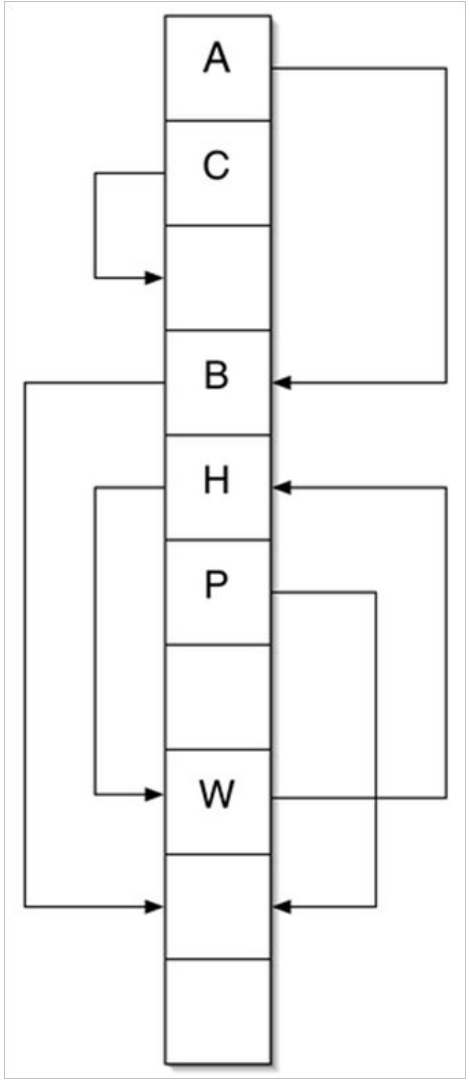
Хеширование кукушкой

Две вариации:

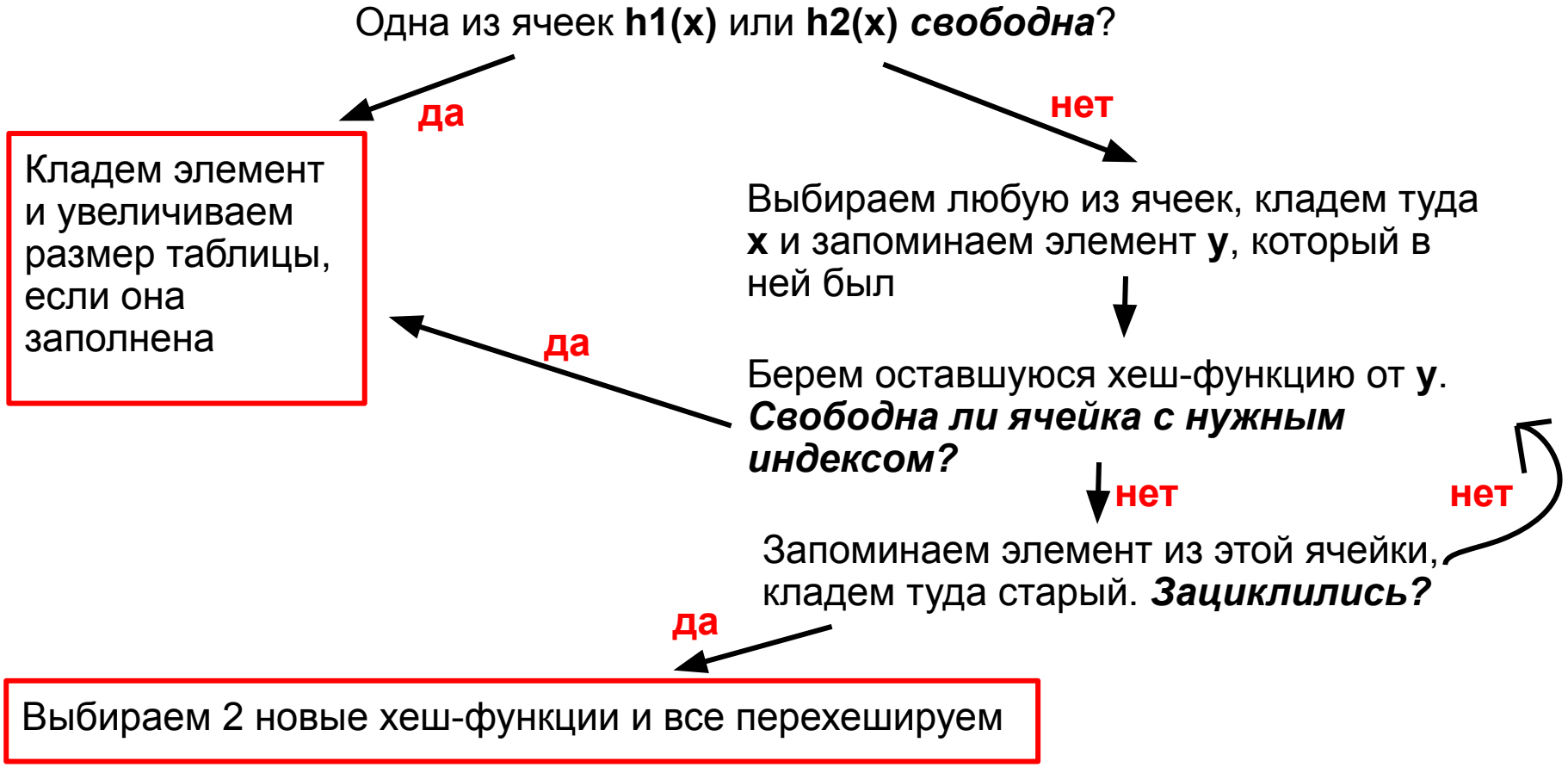
- Две независимые хеш функции
- Две таблицы для двух хеш – функций

Алгоритм **insert**:

- Если одна из ячеек с индексами **h1(x)** или **h2(x)** свободна, кладем в нее
- Иначе запоминаем элемент из занятой ячейки и помещаем туда новый
- Проверяем от сохраненного элемента вторую хеш – функцию
- Если занято, то сохраняем элемент, записываем ранее сохраненный и продолжаем дальше поиск



Хеширование кукушкой. **Insert**



Хеширование кукушкой

Что такое зацикливание?

Хеширование кукушкой

Что такое зацикливание?

Это явление, когда мы кладем куда-то элемент **x** и при дальнейших действиях в том же добавлении снова возвращаемся к ячейке, где лежит **x**, в попытке положить туда другой элемент

Хеширование кукушкой

Что такое зацикливание?

Это явление, когда мы кладем куда-то элемент **x** и при дальнейших действиях в том же добавлении снова возвращаемся к ячейке, где лежит **x**, в попытке положить туда другой элемент

Например, оно возникнет, если добавить в таблицу 3 элемента **x, y, z** такие, что

- **h1(x) = h1(y) = h1(z)**
- **h2(x) = h2(y) = h2(z)**

Хеширование кукушкой

Например, оно возникнет, если добавить в таблицу 3 элемента **x, y, z** такие, что

- **h1(x) = h1(y) = h1(z)**
- **h2(x) = h2(y) = h2(z)**

	h1(x)			
	x			

h1(x) свободна, вставляем в нее **x**

Хеширование кукушкой

Например, оно возникнет, если добавить в таблицу 3 элемента **x, y, z** такие, что

- **h1(x) = h1(y) = h1(z)**
- **h2(x) = h2(y) = h2(z)**

	h1(x)		**h2(y)**	
	x		**y**	

h1(y) занята, но **h2(y)** свободна, вставляем в нее **y**

Хеширование кукушкой

Например, оно возникнет, если добавить в таблицу 3 элемента **x, y, z** такие, что

- **h1(x) = h1(y) = h1(z)**
- **h2(x) = h2(y) = h2(z)**

	h1(x)		**h2(y)**	
	z		**y**	

h1(z) занята и **h2(z)** тоже. Выбираем одну из занятых ячеек, кладем туда **z** и запоминаем лежавший там элемент. Допустим, **x**

Хеширование кукушкой

Например, оно возникнет, если добавить в таблицу 3 элемента **x, y, z** такие, что

- **h1(x) = h1(y) = h1(z)**
- **h2(x) = h2(y) = h2(z)**

	h1(x)		**h2(y)**	
	z		**x**	

Из **h1(x)** мы **x** только что убрали, поэтому берем **h2(x)**
Эта ячейка занята, поэтому мы кладем туда **x** и запоминаем то, что там лежало до этого

Хеширование кукушкой

Например, оно возникнет, если добавить в таблицу 3 элемента **x, y, z** такие, что

- **h1(x) = h1(y) = h1(z)**
- **h2(x) = h2(y) = h2(z)**

	h1(x)		**h2(y)**	
	y		**x**	

Из **h2(y)** мы **y** только что убрали, поэтому берем **h1(y)**
Эта ячейка занята, поэтому мы кладем туда **y** и запоминаем то, что там лежало до этого

Хеширование кукушкой

Например, оно возникнет, если добавить в таблицу 3 элемента **x, y, z** такие, что

- **h1(x) = h1(y) = h1(z)**
- **h2(x) = h2(y) = h2(z)**

	h1(x)		**h2(y)**	
	y		**x**	

При попытке добавить **z** мы вернулись к той же ячейке, в которую изначально его положили. Значит если мы будем продолжать алгоритм, будут выполняться один и те же действия

-> зацикливание!

Хеширование кукушкой

Что такое зацикливание?

Это явление, когда мы кладем куда-то элемент **x** и при дальнейших действиях в том же добавлении снова возвращаемся к ячейке, где лежит **x**, в попытке положить туда другой элемент

Например, оно возникнет, если добавить в таблицу 3 элемента **x, y, z** такие, что

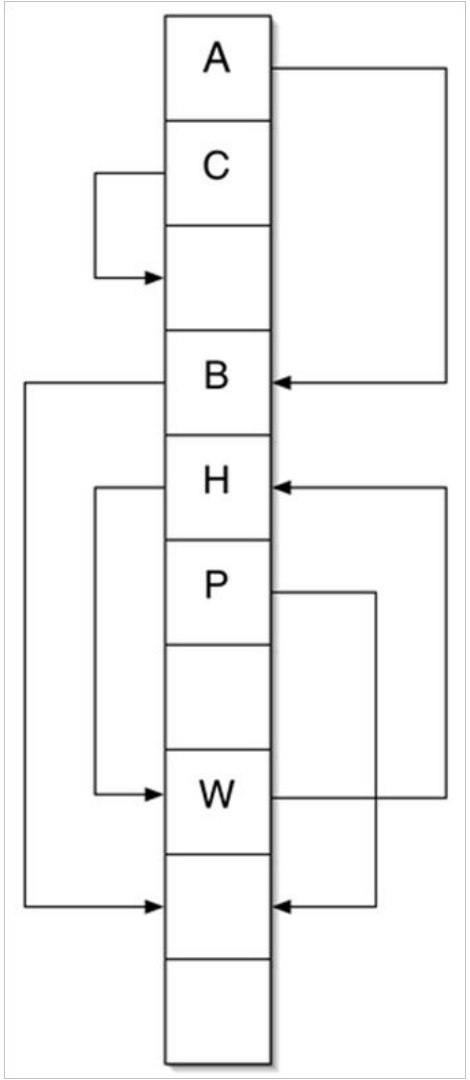
- **h1(x) = h1(y) = h1(z)**
- **h2(x) = h2(y) = h2(z)**

Одним из способов решения проблемы зацикливания является смена хеш-функции

Хеширование кукушкой

Пример хеширования кукушки:

- Стрелки показывают второе возможное место элементов.
- Если нам надо будет вставить новый элемент на место **A**, то мы поместим **A** в его вторую ячейку, занятую **B**, а **B** переместим в его вторую ячейку, которая сейчас свободна.
- А вот помещение нового элемента на место **H** не получится: так как **H** — часть цикла, добавленный элемент будет вытеснен после прохода по циклу.



Хеширование кукушкой

Пример хеширования кукушки:

h1(x) = x mod 8
h2(x) = [x / 8] mod 8

0	1	2	3	4	5	6	7

Хеширование кукушкой

Пример хеширования кукушки:

h1(x) = x mod 8
h2(x) = [x / 8] mod 8

Вставим в таблицу ключи **10, 22, 13**. Они вставляются по индексу своей первой хеш-функции

0	1	2	3	4	5	6	7
		10			13	22	

Хеширование кукушкой

Пример хеширования кукушки:

h1(x) = x mod 8
h2(x) = [x / 8] mod 8

Теперь попробуем вставить **5**

0	1	2	3	4	5	6	7
		10			13	22	

Хеширование кукушкой

Пример хеширования кукушки:

h1(x) = x mod 8
h2(x) = [x / 8] mod 8

Теперь попробуем вставить **5**
h1(5) = 5 mod 8 = 5, но эта ячейка занята

0	1	2	3	4	5	6	7
		10			13	22	

Хеширование кукушкой

Пример хеширования кукушки:

h1(x) = x mod 8
h2(x) = [x / 8] mod 8

Теперь попробуем вставить **5**
h1(5) = 5 mod 8 = 5, но эта ячейка занята
h2(5) = [5 / 8] mod 8 = 0, ячейка свобода - вставляем

0	1	2	3	4	5	6	7
5		10			13	22	

Хеширование кукушкой

Пример хеширования кукушки:

h1(x) = x mod 8
h2(x) = [x / 8] mod 8

Попробуем вставить **42**

0	1	2	3	4	5	6	7
5		10			13	22	

Хеширование кукушкой

Пример хеширования кукушки:

h1(x) = x mod 8
h2(x) = [x / 8] mod 8

Попробуем вставить **42**
h1(42) = 42 mod 8 = 2, но эта ячейка занята

0	1	2	3	4	5	6	7
5		10			13	22	

Хеширование кукушкой

Пример хеширования кукушки:

h1(x) = x mod 8
h2(x) = [x / 8] mod 8

Попробуем вставить **42**
h1(42) = 42 mod 8 = 2, но эта ячейка занята
h2(42) = [42 / 8] mod 8 = 5, эта ячейка тоже занята

0	1	2	3	4	5	6	7
5		10			13	22	

Хеширование кукушкой

Пример хеширования кукушки:

h1(x) = x mod 8
h2(x) = [x / 8] mod 8

Тогда поставим **42** в ячейку **2** и запомним, что до этого там было значение **10**

0	1	2	3	4	5	6	7
5		42			13	22	

Хеширование кукушкой

Пример хеширования кукушки:

h1(x) = x mod 8
h2(x) = [x / 8] mod 8

Теперь нужно подумать, куда переместить **10**.

0	1	2	3	4	5	6	7
5		42			13	22	

Хеширование кукушкой

Пример хеширования кукушки:

h1(x) = x mod 8
h2(x) = [x / 8] mod 8

Теперь нужно подумать, куда переместить **10**. Раньше **10** стояла по индексу **h1(10) = 2**. Попытаемся поставить ее по индексу **h2(10) = [10 / 8] mod 8 = 1**

0	1	2	3	4	5	6	7
5		42			13	22	

Хеширование кукушкой

Пример хеширования кукушки:

h1(x) = x mod 8
h2(x) = [x / 8] mod 8

Теперь нужно подумать, куда переместить **10**. Раньше **10** стояла по индексу **h1(10) = 2**. Попытаемся поставить ее по индексу **h2(10) = [10 / 8] mod 8 = 1**
Ячейка с индексом **1** свободна -> ставим в нее **10**

0	1	2	3	4	5	6	7
5	10	42			13	22	

Хеширование кукушкой

Пример хеширования кукушки:

h1(x) = x mod 8
h2(x) = [x / 8] mod 8

Все оставшиеся элементы ставятся точно так же. Либо в какой-то момент находится пустая ячейка, в которую можно поставить элемент, либо происходит зацикливание, и вся таблица перехешируется.

0	1	2	3	4	5	6	7
5	10	42			13	22	

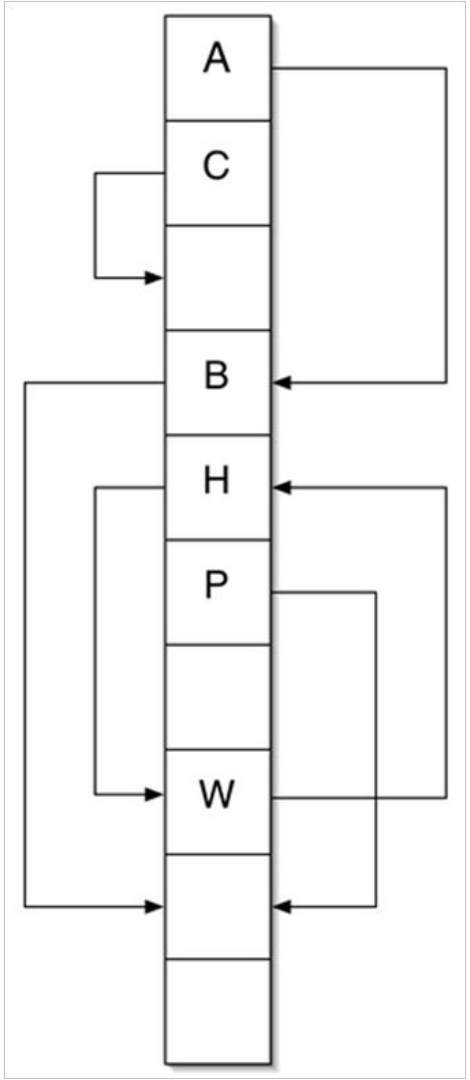
Хеширование кукушкой

Алгоритм **remove:**

- Смотрим на ячейку с индексами **h1(x)** и **h2(x)**
- Если в одной из них есть нужный элемент, помечаем как свободную

Алгоритм **contains:**

- Смотрим на ячейку с индексами **h1(x)** и **h2(x)**
- Если в одной из них есть нужный элемент, возвращаем **true**
- Иначе возвращаем **false**



Расширение темы: применение

Ассоциативный массив

Массив


Массив представляет собой совокупность объектов, имеющих одинаковые размер и тип. Каждый объект в массиве называется элементом массива.
Существует два типа массивов, различающиеся по способу идентификации элементов:
По размерности массивы делятся на одномерные и многомерные.
в массивах первого типа элемент определяется индексом в последовательности,
массивы второго типа имеют ассоциативную природу, и для обращения к элементам используются ключи, логически связанные со значениями.

Одномерный индексируемый массив

При обращении к элементам **одномерных индексируемых массивов** используется целочисленный индекс, определяющий позицию заданного элемента.

Обобщенный синтаксис элементов одномерного массива: **$имя [индекс 1];**

Одномерные массивы создаются следующим образом:

```
$meat[0] = "chicken";   $meat[l] = "steak";   $meat[2] = "turkey";
```

При выполнении следующей команды: print $meat[1]; в браузере выводится строка steak.

Одномерный ассоциативный массив

Ассоциативные массивы используются в ситуациях, когда элементы массива удобнее связывать со словами, а не с числами. Ассоциативный массив заметно экономит время и объем программного кода, необходимого для вывода определенных элементов массива.

Предположим, нужно сохранить в массиве лучшие сочетания напитков и блюд. Проще всего было бы хранить в массиве пары «ключ/значение» – например, присвоить напиток названию блюда. Самым разумным решением будет использование ассоциативного массива:

```
$pairings["zinfandel"] = "Broiled Veal Chops";
$pairings["merlot"] = "Baked Ham";
$pairings["sauvignon"] = "Prime Rib";
$pairings["sauternes"] = "Roasted Salmon";
```

Если нужно узнать, с каким блюдом лучше всего идет напиток «merlot», необходимо использовать ссылку на элемент массива:

```
$pairings: print $pairings["merlot"]; // Выводится строка "Baked Ham"
```

Ассоциативный массив

— абстрактный тип данных (интерфейс к хранилищу данных), позволяющий хранить пары вида «(ключ, значение)» и поддерживающий операции добавления пары, а также поиска и удаления пары по ключу:

- `INSERT(ключ, значение)`
- `FIND(ключ)`
- `REMOVE(ключ)`

Предполагается, что ассоциативный массив не может хранить две пары с одинаковыми ключами.

Ассоциативный массив

Пример

```
$employee = [
  'name' => 'John',
  'email' => 'john@example.com',
  'phone' => '1234567890',
  'hobbies' => ['Football', 'Tennis'],
  'profiles' => ['facebook' => 'johnfb', 'twitter' => 'johntw']
];
```

ключ => значение

ключ => множество значений

ключ => в качестве значения ассоциативный массив

Смешанное индексирование

В многомерных массивах допускается **смешанное индексирование** (числовое и ассоциативное).

Допустим, нужно расширить модель одномерного ассоциативного массива для хранения информации об игроках первого и второго состава футбольной команды.

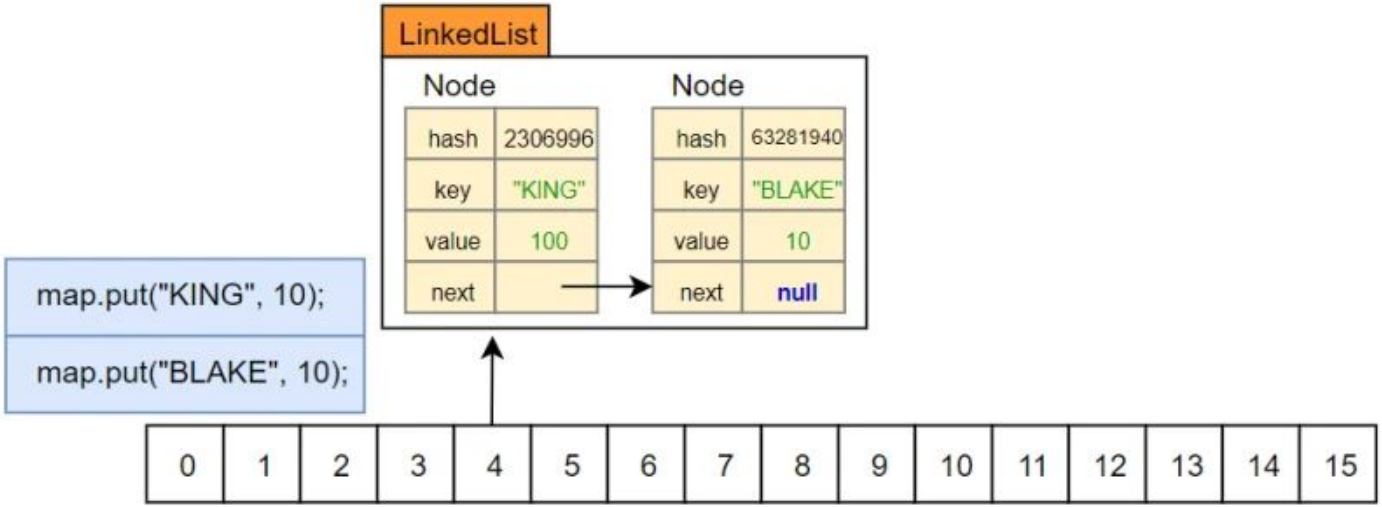
Решение может выглядеть следующим образом:

```
$Buckeyes["quarterback"] [1] = "Bellisari";
$Buckeyes["quarterback"] [2] = "Moherman":
$Buckeyes["quarterback"] [3] = "Wiley";
```

Какими средствами можно реализовать ассоциативный массив?

Что ещё работает по принципу пара: ключ => значение?

Реализация ассоциативного массива через хеш-таблицу с закрытой адресацией




```
public V put(K key, V value)
        hash(Object key)
        index = (n - 1) & hash
```

hash("BLAKE") = 63281940

index = 4

n = 16

P.S. Ассоциативный массив можно реализовать и на другой структуре данных, например на дереве

MultiMap

Многомерный индексируемый массив

Многомерные индексируемые массивы работают практически так же, как и их одномерные прототипы, однако элементы в них определяются несколькими индексами вместо одного.

Теоретически размерность индексируемого массива не ограничивается, хотя в большинстве приложений практически не встречаются массивы с размерностью выше 3.

Обобщенный синтаксис элементов многомерного массива:

```
$имя[индекс 1][индекс 2],...,[индекс N];
```

Пример ссылки на элемент двухмерного индексируемого массива:

```
$position = $chess_board[5][4]; // выводит позицию шахматной фигуры
```

Многомерный ассоциативный массив

Допустим, в массиве `$pairings` из примера приведенного ранее должна храниться информация не только о сорте, но и о производителе напитка.

Присваивание значений элементам можно сделать следующим образом:

```
$pairings["Martinelli"]["zinfandel"] = "Broiled Veal Chops";
$pairings["Beringer"]["merlot"] = "Baked Ham";
$pairings["Jarvis"]["sauvignon"] = "Prime Rib";
$pairings["Climens"]["sauternes"] = "Roasted Salmon";
```

ключ, по которому в качестве значения получили ассоциативный массив

Вариации реализации MultiMap

Implementation	Keys behave like...	Values behave like...
ArrayListMultimap	HashMap	ArrayList
HashMultimap	HashMap	HashSet
LinkedListMultimap	LinkedHashMap	LinkedList
LinkedHashMultimap	LinkedHashMap	LinkedHashSet
TreeMultimap	TreeMap	TreeSet
ImmutableListMultimap	ImmutableMap	ImmutableList
ImmutableSetMultimap	ImmutableMap	ImmutableSet

Различая словарей

Associative Container	Keys sorted	Value available	Identical keys possible
`std::set`	Yes	No	No
`std::multiset`	Yes	No	Yes
`std::unordered_set`	No	No	No
`std::unordered_multiset`	No	No	Yes
`std::map`	Yes	Yes	No
`std::multimap`	Yes	Yes	Yes
`std::unordered_map`	No	Yes	No
`std::unordered_multimap`	No	Yes	Yes

КриптоХеш

Криптография

— это метод защиты информации путем ее преобразования в нечитаемый формат.

Криптография

— это метод защиты информации путем ее преобразования в нечитаемый формат.

КриптоХеш функция

— Разработаны для защиты от атак.
Они детерминированы, т.е. на одном и том же входе всегда будет один и тот же выход, и они производят выход фиксированного размера независимо от размера входа.

КриптоХеш функция

Криптографической функцией хеширования (хеш-функцией) называется алгоритм, который отображает сообщение произвольного размера в относительно короткий (скажем, 256 бит) массив битов фиксированного размера.

Этот массив называется **хеш-значением сообщения**, или **криптографическим хешем**.

КриптоХеш функция

Криптографической функцией хеширования (хеш-функцией) называется алгоритм, который отображает сообщение произвольного размера в относительно короткий (скажем, 256 бит) массив битов фиксированного размера.

Этот массив называется **хеш-значением сообщения**, или **криптографическим хешем**.

!!!

Можно сказать, что хеш сообщения – это **криптографически стойкая контрольная сумма**.

КриптоХеш функция: СВОЙСТВА

- **Детерминирована**, т.е. обработка одного и того же сообщения всегда дает одно и то же хеш-значение;
- **Необратима**, т.е. невозможно или очень трудно восстановить исходное сообщение, зная его хеш. Единственным способом обратить хеш должен быть *полный перебор*, и его выполнение должно быть вычислительно неосуществимым;
- **Устойчивость к коллизиям** - должно быть вычислительно неосуществимо найти два разных сообщения с одинаковым хеш-значением.
- **Эффект лавины** - любое изменение сообщения, большое или малое, должно приводить к значительному изменению хеш-значения – настолько значитель ному, что два хеш-значения невозможно связать друг с другом.

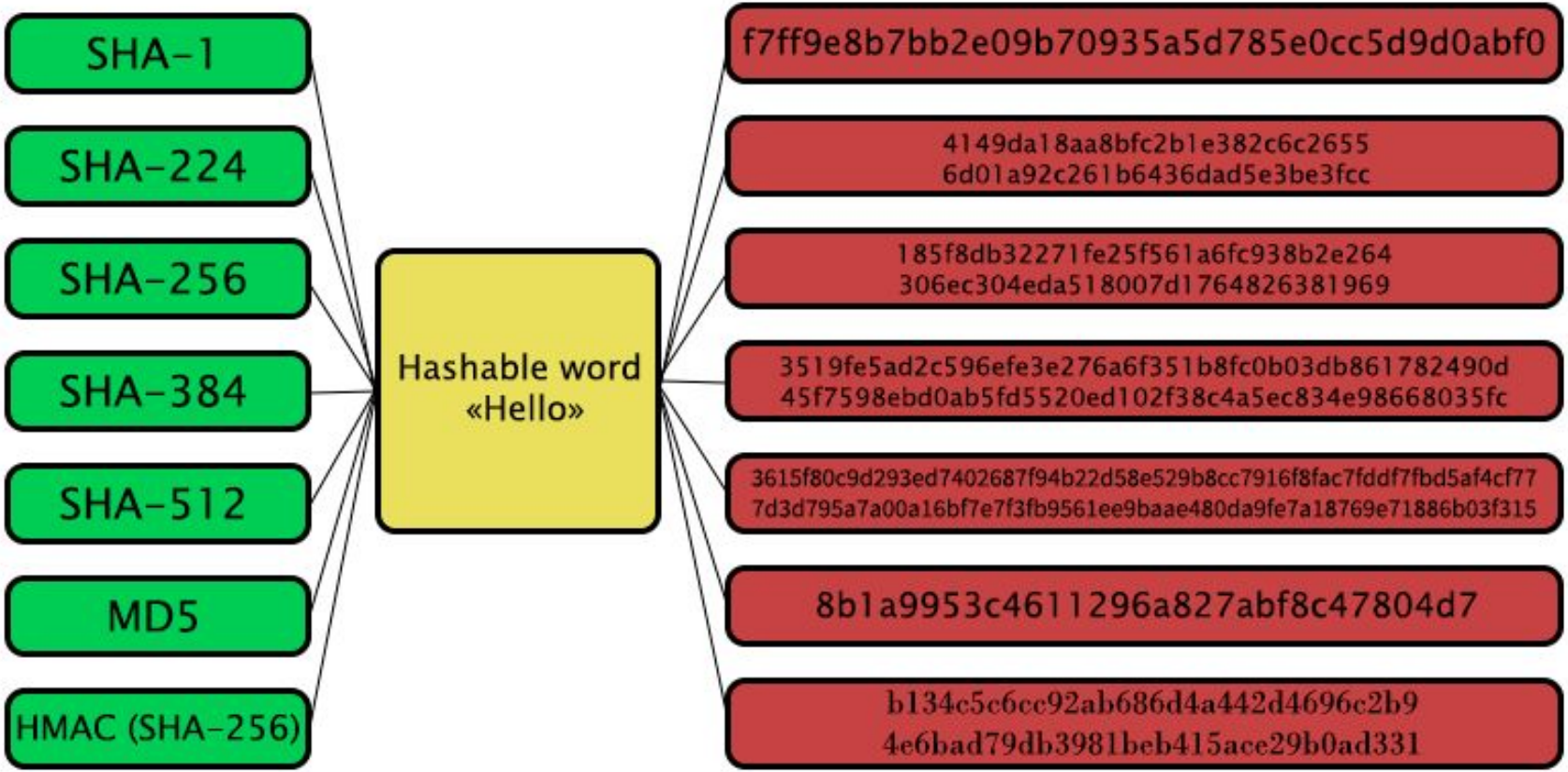
Контрольная сумма

- Простая форма хэш-функции, используемая для обнаружения ошибок в данных.
- Они вычисляют значение на основе данных и отправляют его вместе с данными.
- Получатель может затем пересчитать контрольную сумму и сравнить ее с полученной контрольной суммой для проверки на наличие ошибок.

ПРИМЕНИМОСТЬ

- Целостность данных
- Цифровая подпись
- Цифровые сертификаты
- Хранение паролей
- Идентификатор содержимого
- Блокчейн и криптовалюты
  - Верификация транзакций
  - Блочная увязка

Алгоритмы


SHA-1
SHA-224
SHA-256
SHA-384
SHA-512
MD5
HMAC (SHA-256)
Hashable word «Hello»
f7ff9e8b7bb2e09b70935a5d785e0cc5d9d0abf0
4149da18aa8bfc2b1e382c6c2655 6d01a92c261b6436dad5e3be3fcc
185f8db32271fe25f561a6fc938b2e264 306ec304eda518007d1764826381969
3519fe5ad2c596efe3e276a6f351b8fc0b03db861782490d 45f7598ebd0ab5fd5520ed102f38c4a5ec834e98668035fc
3615f80c9d293ed7402687f94b22d58e529b8cc7916f8fac7fddf7fbd5af4cf77 7d3d795a7a00a16bf7e7f3fb9561ee9baae480da9fe7a18769e71886b03f315
8b1a9953c4611296a827abf8c47804d7
b134c5c6cc92ab686d4a442d4696c2b9 4e6bad79db3981beb415ace29b0ad331

Концепция работы криптохеша!

1 - Итеративная последовательная схема

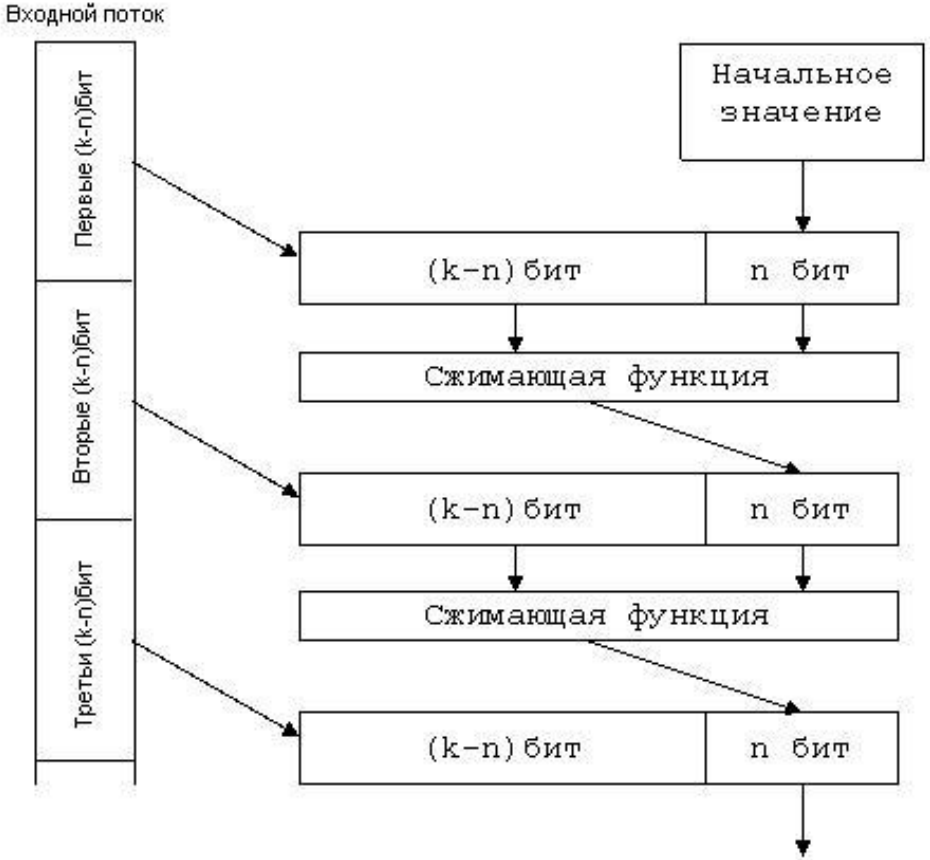
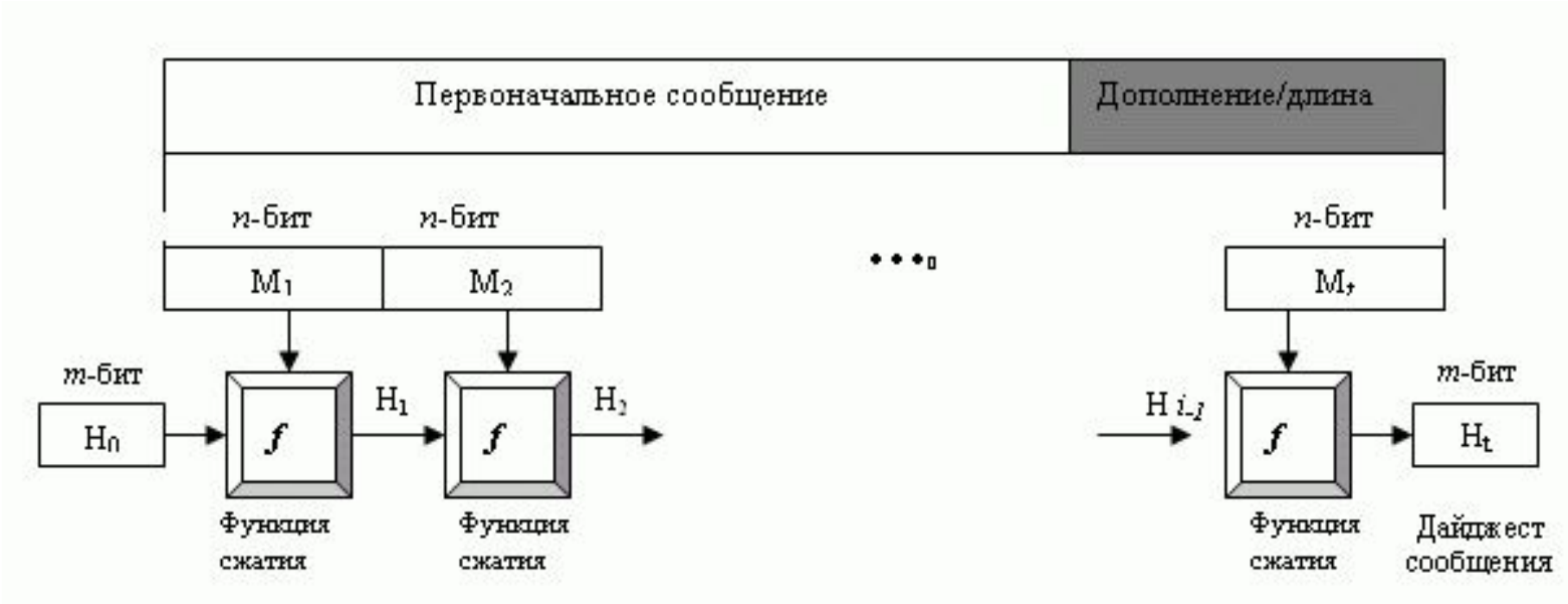
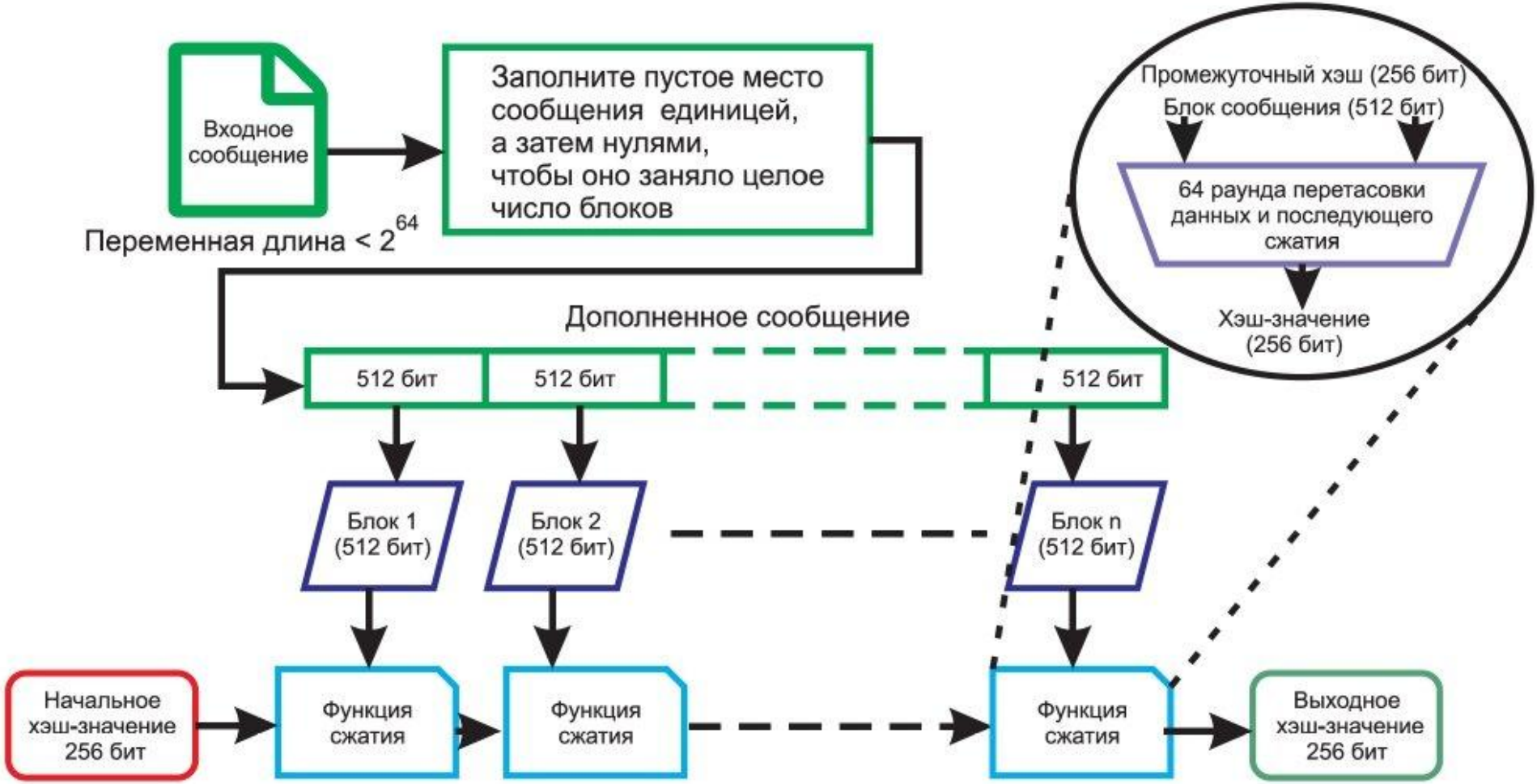


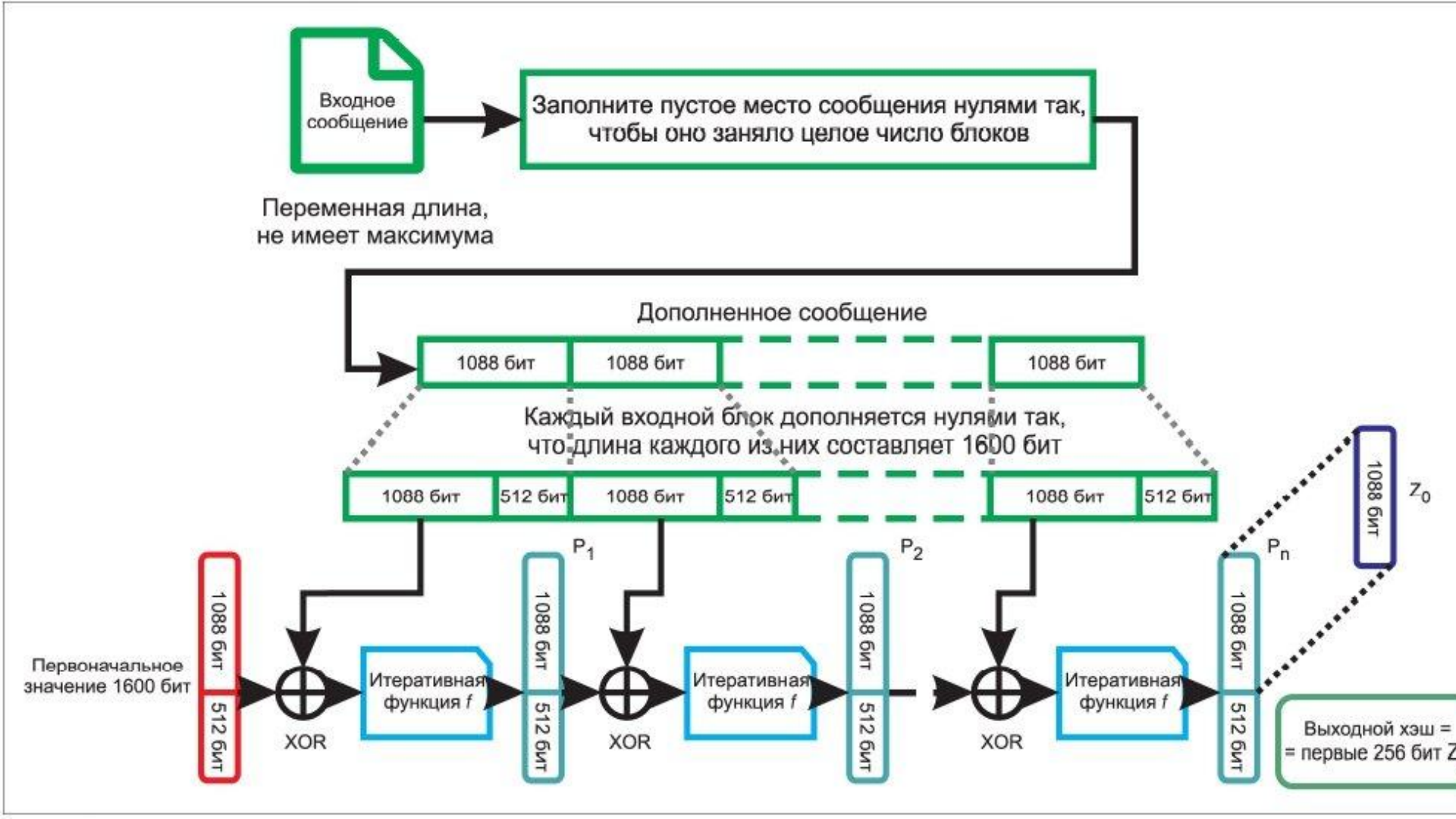
Рис 1.Итеративная последовательная схема

1 - Итеративная последовательная схема

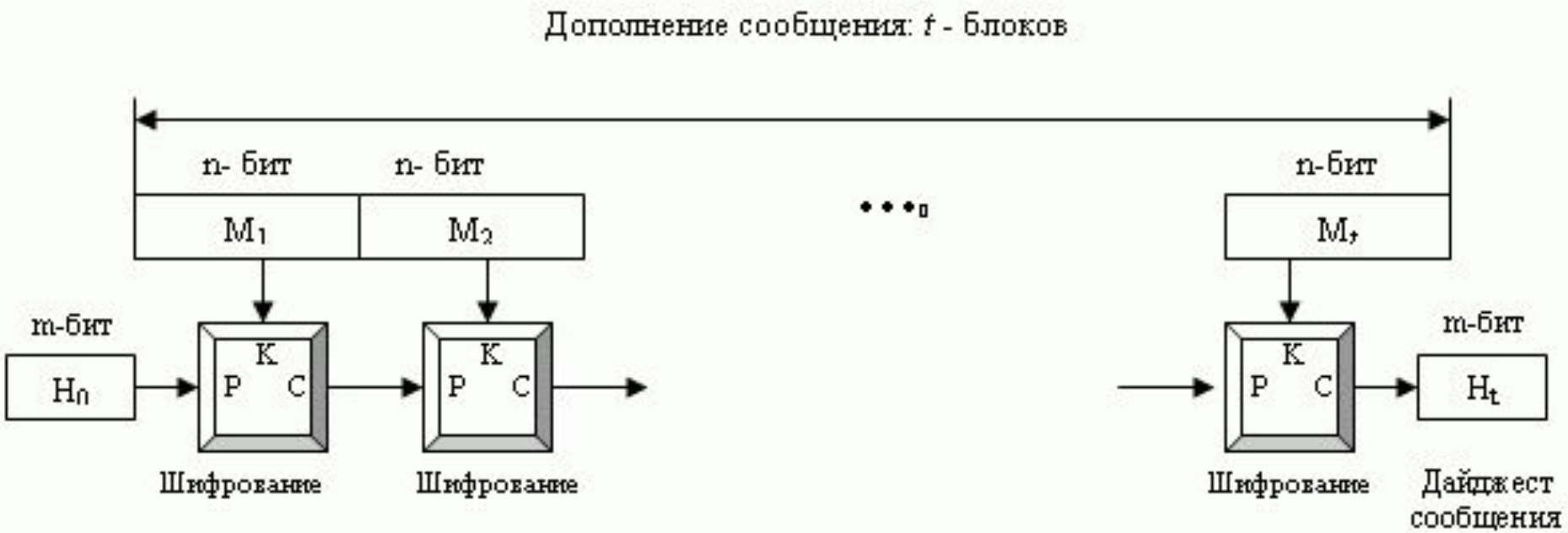


Безопасная генерация хэша – функция SHA-256




Входное сообщение
Заполните пустое место сообщения нулями так, чтобы оно заняло целое число блоков
Переменная длина, не имеет максимума
Дополненное сообщение
1088 бит
1088 бит
1088 бит
Каждый входной блок дополняется нулями так, что длина каждого из них составляет 1600 бит
1088 бит
512 бит
1088 бит
512 бит
1088 бит
512 бит
P1
P2
Pn
Z0
1088 бит
Первоначальное значение 1600 бит
1088 бит
512 бит
XOR
Итеративная функция f
1088 бит
512 бит
XOR
Итеративная функция f
1088 бит
512 бит
XOR
Итеративная функция f
1088 бит
512 бит
Выходной хэш =
= первые 256 бит Z

2 - На основе симметричного блочного алгоритма



В основе лежит алгоритм блочного шифрования!

2 - На основе симметричного блочного алгоритма

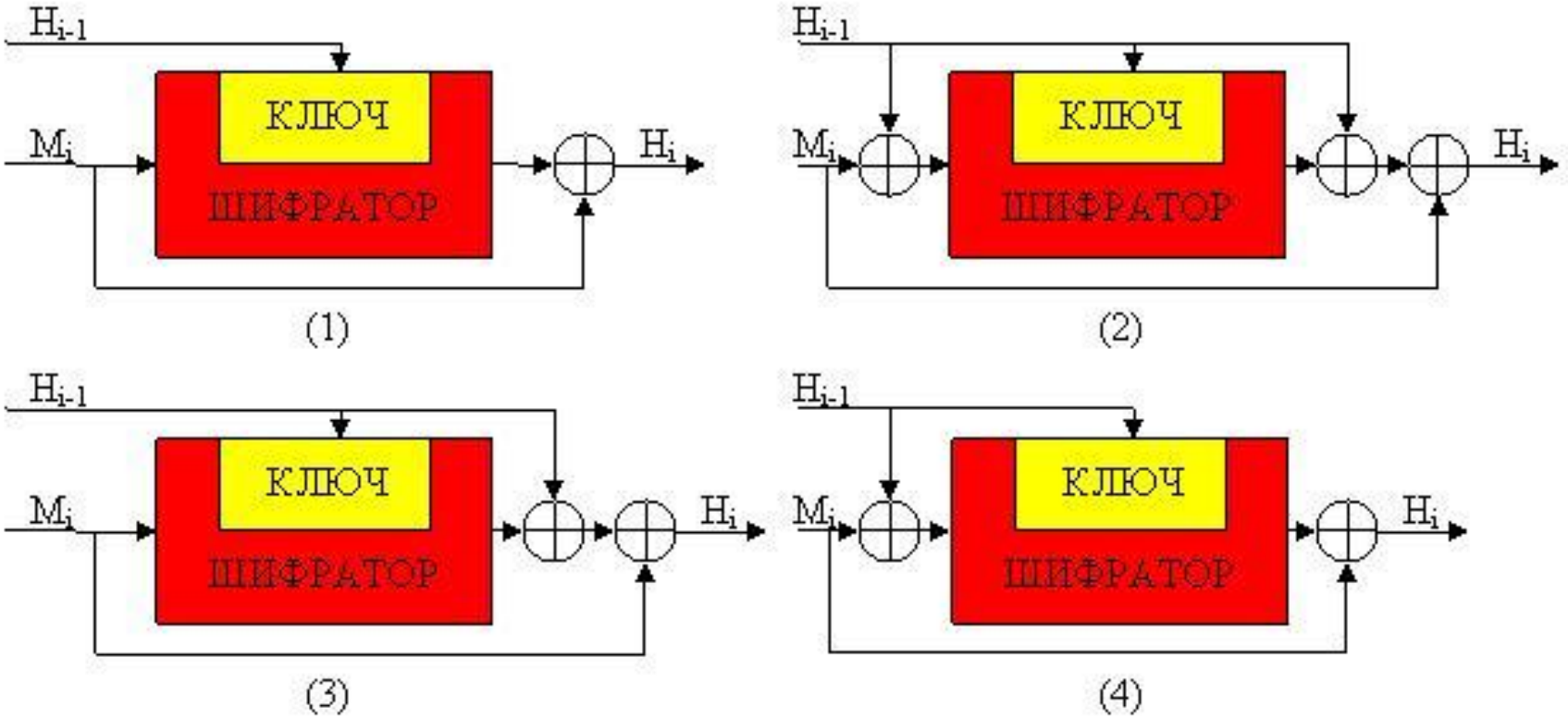
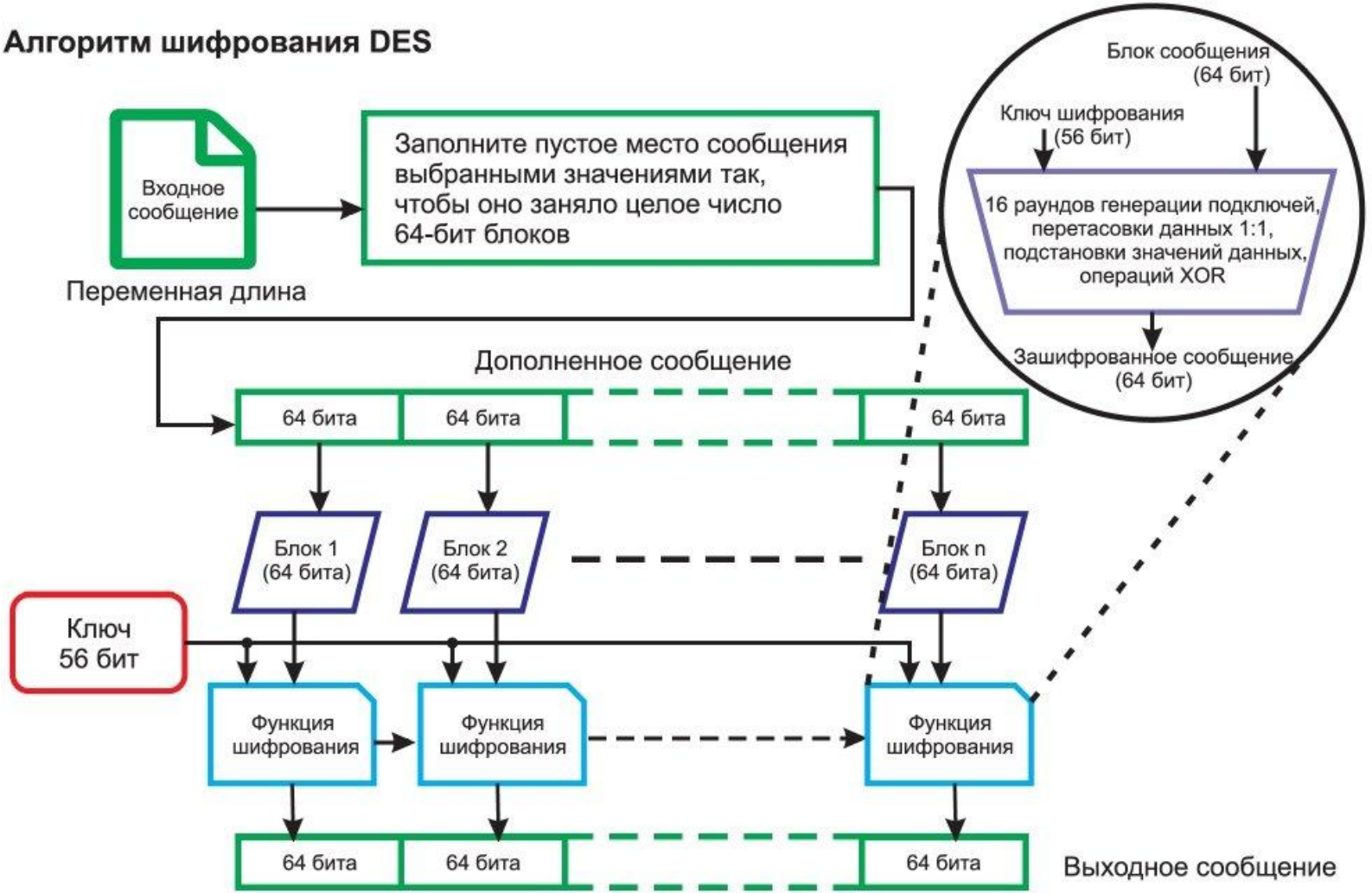
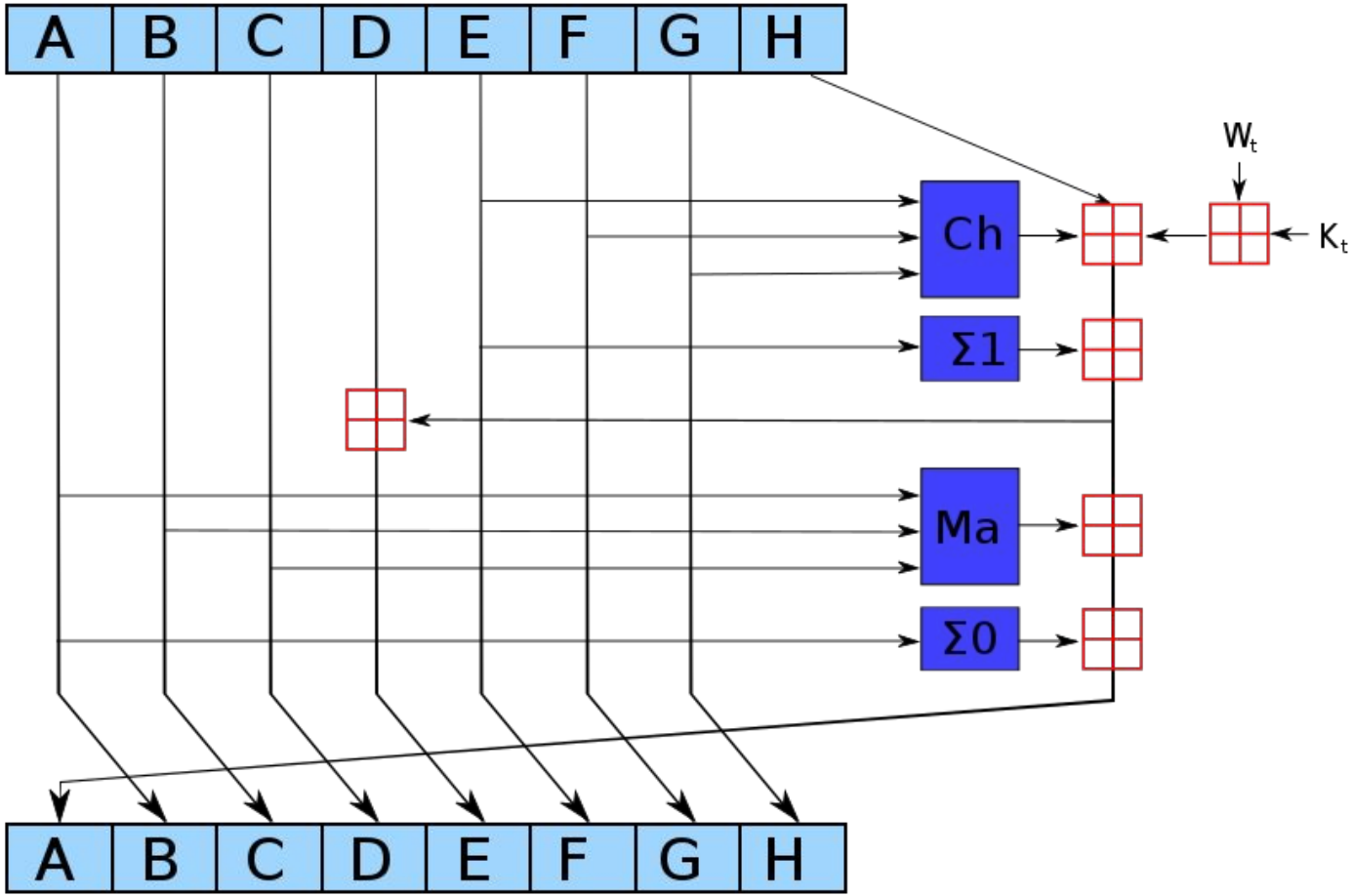


Рис.3. Четыре схемы безопасного хэширования

Алгоритм шифрования DES



Иттерация SHA - 2



Зачем надо солить хеш ?

“Соль” в криптохешировании

- Радужные таблицы
- Посолим хеш
  - Статическая соль
  - Уникальная динамическая соль

Радужные таблицы

- Радужные таблицы состоят из хэшей наиболее часто употребляемых паролей — имен, дат рождения, названий животных и т.п.
- Эти таблицы могут включать миллионы, миллиарды значений, но работа с ними относительно быстра, и проверить хэш на соответствие одному из значений не составляет никакого труда.

Радужные таблицы

- Радужные таблицы состоят из хэшей наиболее часто употребляемых паролей — имен, дат рождения, названий животных и т.п.
- Эти таблицы могут включать миллионы, миллиарды значений, но работа с ними относительно быстра, и проверить хэш на соответствие одному из значений не составляет никакого труда.

Рассмотрим ситуацию: У Вас есть хешированный пароль пользователя, и хранится он в таблице базы данных. Даже если злоумышленник получает доступ к нашей базе данных он не сможет определить исходный пароль. Но что если он сравнивает все хэшированные пароли друг с другом, и находит некоторые из них как быть?

Радужные таблицы

- Радужные таблицы состоят из хэшей наиболее часто употребляемых паролей — имен, дат рождения, названий животных и т.п.
- Эти таблицы могут включать миллионы, миллиарды значений, но работа с ними относительно быстра, и проверить хэш на соответствие одному из значений не составляет никакого труда.

Рассмотрим ситуацию: У Вас есть хешированный пароль пользователя, и хранится он в таблице базы данных. Даже если злоумышленник получает доступ к нашей базе данных он не сможет определить исходный пароль. Но что если он сравнивает все хэшированные пароли друг с другом, и находит некоторые из них как быть?

Мы уже знаем две строки могут иметь одинаковый хеш, только если они обе равны(без учёта коллизии). Значит, если атакующий видит хеши он может сделать вывод о том что пароли для этих учетных записей одинаковые. Если атакующий, знает хотя бы один пароль к аккаунту, он может его использовать для получения доступа ко всем аккаунтам с этим паролем.

Решение проблемы: посолить!

Одним из решением проблемы радужных таблиц является использование случайного числа при генерации хеша, так называемая ***соль***.

```
$salt = "f?*-Q@t03#6_z";
```

```
$hash = sha1($salt  + "password");
```

Решение проблемы: посолить!

Одним из решением проблемы радужных таблиц является использование случайного числа при генерации хеша, так называемая ***соль***.

```
$salt = "f?*-Q@t03#6_z";
```

```
$hash = sha1($salt  + "password");
```

Статическая соль может служить достаточно хорошо, пока структура конструкции и соль хранятся в тайне.

Если же злоумышленник узнает секрет хэширования - он с легкостью сможет модифицировать под него свою радужную таблицу.

Уникальная соль!

Решение: можно сгенерировать уникальную ***соль*** для каждого пользователя.

```
$salt = "f?*-Q@t03#6_z";
```

```
$hash = sha1($salt  + "password");
```

Уникальная соль!

Решение: можно сгенерировать уникальную ***соль*** для каждого пользователя.

```
$salt[i]= "f?*-Q@t03#6_z";
```

```
$hash = sha1($salt[i]  + "password");
```

USER	PASSWORD	SALT
user	6aa747272c7d077d0fd1672ae…	}{@kE$
prof	a3910b3c32a672ea9ee70866…	#_абв78
pety	08ad24df81e43e013c5cf8564…	~~$~~

Фильтр БЛУМА

Вероятностное множество

Вероятностное множество - это структура данных, позволяющая добавлять элементы в множество и выполнять запросы проверки принадлежности элемента множеству.

Вероятностное множество

Вероятностное множество - это структура данных, позволяющая добавлять элементы в множество и выполнять запросы проверки принадлежности элемента множеству.

Структура позволяет понять, что элемент совершенно точно **НЕ** принадлежит множеству! При запросе на то, принадлежит ли элемент, возможны ложноположительные срабатывания

Вероятностное множество

Вероятностное множество - это структура данных, позволяющая добавлять элементы в множество и выполнять запросы проверки принадлежности элемента множеству.

Структура позволяет понять, что элемент совершенно точно **НЕ** принадлежит множеству! При запросе на то, принадлежит ли элемент, возможны ложноположительные срабатывания

Другими словами, при запросе **“элемент x НЕ принадлежит множеству?”** при ответе:

- **ДА**. Элемент точно не принадлежит множеству
- **НЕТ**. Элемент возможно есть в множестве, но его может и не быть

Фильтр Блума

Фильтр Блума — это реализация вероятностного множества, позволяющая компактно хранить элементы и проверять принадлежность заданного элемента к множеству. При этом существует возможность получить ложноположительное срабатывание (элемента в множестве нет, но структура данных сообщает, что он есть), но не ложноотрицательное.

Фильтр Блума

Фильтр Блума — это реализация вероятностного множества, позволяющая компактно хранить элементы и проверять принадлежность заданного элемента к множеству. При этом существует возможность получить ложноположительное срабатывание (элемента в множестве нет, но структура данных сообщает, что он есть), но не ложноотрицательное.

- Фильтр Блума может использовать любой объём памяти, заранее заданный пользователем, причем чем он больше, тем меньше вероятность ложного срабатывания.

Фильтр Блума

Фильтр Блума — это реализация вероятностного множества, позволяющая компактно хранить элементы и проверять принадлежность заданного элемента к множеству. При этом существует возможность получить ложноположительное срабатывание (элемента в множестве нет, но структура данных сообщает, что он есть), но не ложноотрицательное.

- Фильтр Блума может использовать любой объём памяти, заранее заданный пользователем, причем чем он больше, тем меньше вероятность ложного срабатывания.
- Поддерживается операция добавления новых элементов в множество, но не удаления существующих

Фильтр Блума

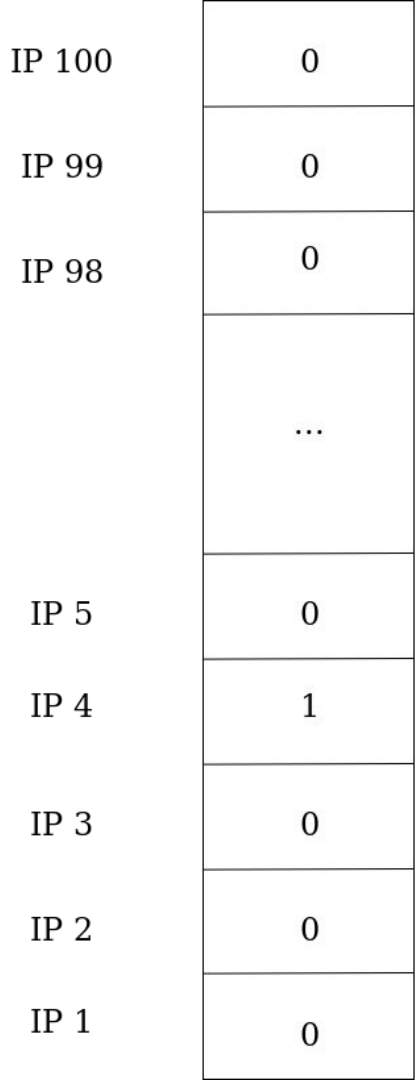
Фильтр Блума — это реализация вероятностного множества, позволяющая компактно хранить элементы и проверять принадлежность заданного элемента к множеству. При этом существует возможность получить ложноположительное срабатывание (элемента в множестве нет, но структура данных сообщает, что он есть), но не ложноотрицательное.

- Фильтр Блума может использовать любой объём памяти, заранее заданный пользователем, причем чем он больше, тем меньше вероятность ложного срабатывания.
- Поддерживается операция добавления новых элементов в множество, но не удаления существующих
- С увеличением размера хранимого множества повышается вероятность ложного срабатывания.

Применимость фильтра Блума

Представим, что у нас есть черный список из 100 IP-адресов, и нам нужно уметь определять, есть ли адрес в черном списке.

Самый простой способ - создать битсет из 100 битов и поставить каждому IP-адресу в соответствие 1 бит. Если адрес есть в списке, то на его месте стоит **1**, иначе **0**

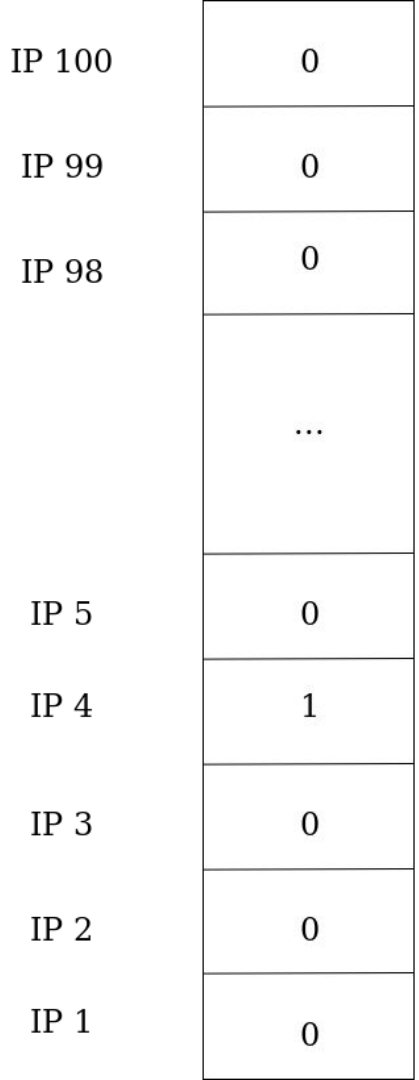


Применимость фильтра Блума

Представим, что у нас есть черный список из 100 IP-адресов, и нам нужно уметь определять, есть ли адрес в черном списке.

Самый простой способ - создать битсет из 100 битов и поставить каждому IP-адресу в соответствие 1 бит. Если адрес есть в списке, то на его месте стоит **1**, иначе **0**

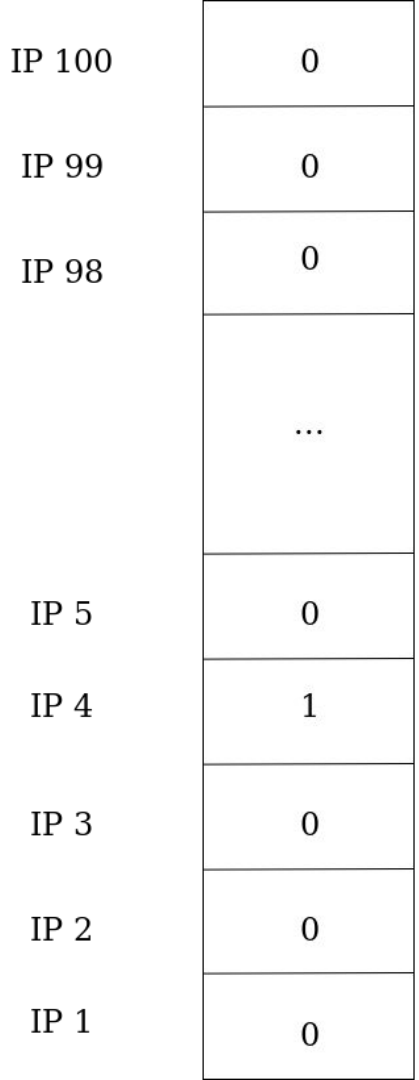
В данном случае адрес **4** есть в списке, а все остальные нет



Применимость фильтра Блума

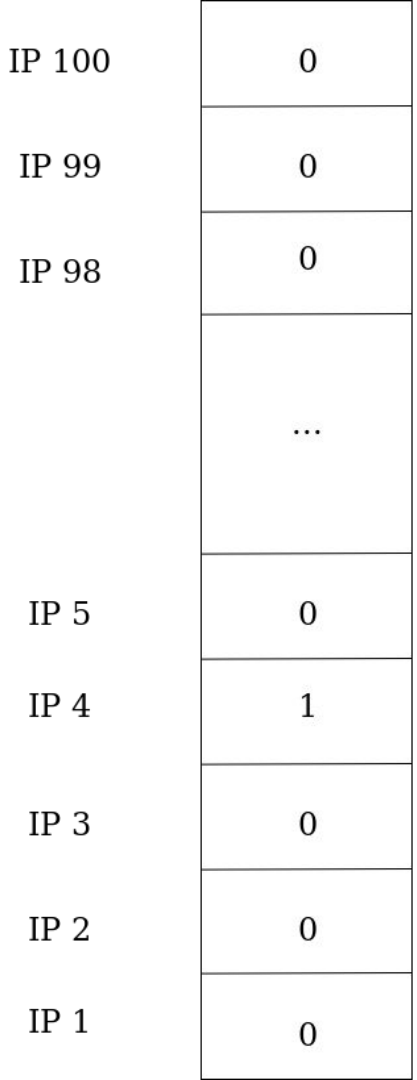
Однако в реальной жизни каждый IPv4-адрес имеет 32 бита, что означает, что существует **4 294 967 296** (2^32) возможных адресов. И количество IP-адресов в черном списке, вероятно, не превысит нескольких сотен в самом крайнем случае.

Мы не можем позволить себе составлять такой большой список для хранения малого количества записей



Применимость фильтра Блума

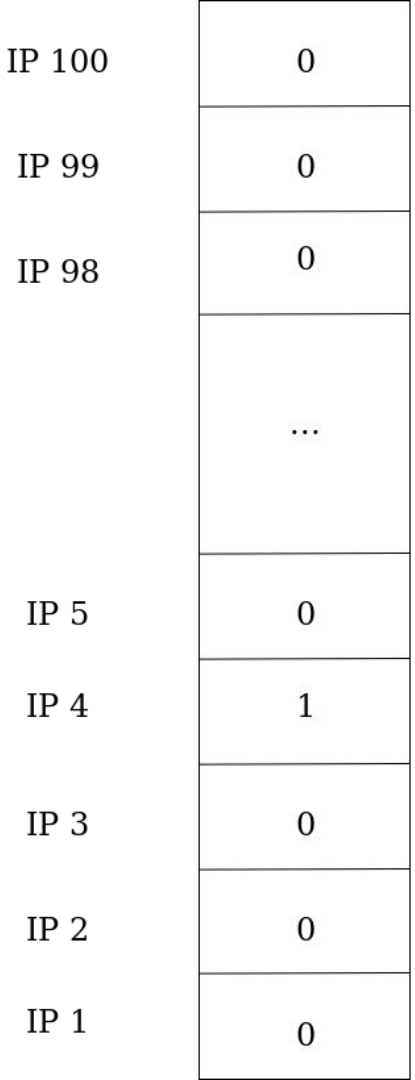
Как найти более оптимальный способ сопоставления IP-адреса и записи в списке?



Применимость фильтра Блума

Как найти более оптимальный способ сопоставления IP-адреса и записи в списке?

Ответ: использовать хеш-функции!



Структура фильтра Блума

Фильтр Блума как структура состоит из:

- битсета из **m** бит
- **k** хеш-функций, каждая из которых отображает элементы исходного множества во множество **{0, … , m - 1}**

Структура фильтра Блума

Фильтр Блума как структура состоит из:

- битсета из **m** бит
- **k** хеш-функций, каждая из которых отображает элементы исходного множества во множество **{0, … , m - 1}**

Изначально весь битсет заполнен нулями. Для добавления элемента **x** в множество нужно поставить единицы на каждую из позиций $\mathbf{h_1(x), \dots , h_k(x)}$

Структура фильтра Блума

Пусть есть элемент **5**, и есть хеш функции:

- $h_1(x) = x \% 7$
- $h_2(x) = (x * 5) \% 7$
- $h_3(x) = ([x / 2] * 4) \% 7$

Изначально битсет - пустое множество

0	1	2	3	4	5	6
0	0	0	0	0	0	0

Структура фильтра Блума

Пусть есть элемент **5**, и есть хеш функции:

- $h_1(x) = x \% 7$
- $h_2(x) = (x * 5) \% 7$
- $h_3(x) = ([x / 2] * 4) \% 7$

Изначально битсет - пустое множество

$h_1(5) = 5 \% 7 = 5$

0	1	2	3	4	5	6
0	0	0	0	0	0	0

Структура фильтра Блума

Пусть есть элемент **5**, и есть хеш функции:

- $h_1(x) = x \% 7$
- $h_2(x) = (x * 5) \% 7$
- $h_3(x) = ([x / 2] * 4) \% 7$

Изначально битсет - пустое множество

$h_1(5) = 5 \% 7 = 5$  ставим 1 на позицию $h_1(5)$

0	1	2	3	4	5	6
0	0	0	0	0	1	0

Структура фильтра Блума

Пусть есть элемент **5**, и есть хеш функции:

- $h_1(x) = x \% 7$
- $h_2(x) = (x * 5) \% 7$
- $h_3(x) = ([x / 2] * 4) \% 7$

Изначально битсет - пустое множество

$h_2(5) = (5 * 5) \% 7 = 25 \% 7 = 4$

0	1	2	3	4	5	6
0	0	0	0	0	1	0

Структура фильтра Блума

Пусть есть элемент **5**, и есть хеш функции:

- $h_1(x) = x \% 7$
- $h_2(x) = (x * 5) \% 7$
- $h_3(x) = ([x / 2] * 4) \% 7$

Изначально битсет - пустое множество

$h_2(5) = (5 * 5) \% 7 = 25 \% 7 = 4$  ставим 1 на позицию $h_2(5)$

0	1	2	3	4	5	6
0	0	0	0	1	1	0

Структура фильтра Блума

Пусть есть элемент **5**, и есть хеш функции:

- $h_1(x) = x \% 7$
- $h_2(x) = (x * 5) \% 7$
- $h_3(x) = ([x / 2] * 4) \% 7$

Изначально битсет - пустое множество

$h_3(5) = ([5 / 2] * 4) \% 7 = 8 \% 7 = 1$

0	1	2	3	4	5	6
0	0	0	0	1	1	0

Структура фильтра Блума

Пусть есть элемент **5**, и есть хеш функции:

- $h_1(x) = x \% 7$
- $h_2(x) = (x * 5) \% 7$
- $h_3(x) = ([x / 2] * 4) \% 7$

Изначально битсет - пустое множество

$h_3(5) = ([5 / 2] * 4) \% 7 = 8 \% 7 = 1$

ставим 1 на позицию $h_3(5)$

0	1	2	3	4	5	6
0	1	0	0	1	1	0

Структура фильтра Блума

Пусть есть элемент **5**, и есть хеш функции:

- $h_1(x) = x \% 7$
- $h_2(x) = (x * 5) \% 7$
- $h_3(x) = ([x / 2] * 4) \% 7$

Изначально битсет - пустое множество

Теперь наше множество состоит из одного элемента **5**

0	1	2	3	4	5	6
0	1	0	0	1	1	0

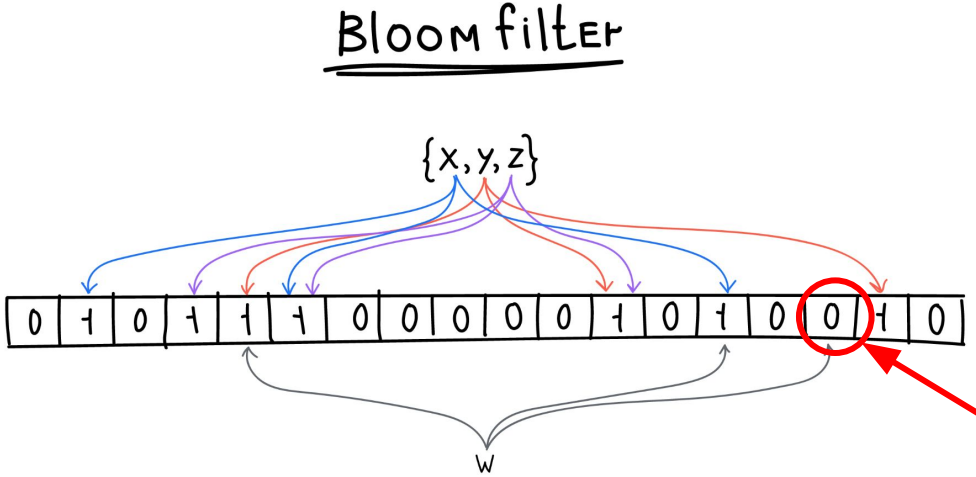
Структура фильтра Блума

Теперь для того, чтобы понять, есть ли элемент **x** в множестве, необходимо посмотреть на состояние всех битов на позициях $\mathbf{h_1(x), \dots , h_k(x)}$

Структура фильтра Блума

Теперь для того, чтобы понять, есть ли элемент **x** в множестве, необходимо посмотреть на состояние всех битов на позициях $\mathbf{h_1(x)}$, … , $\mathbf{h_k(x)}$

- если хотя бы один из проверяемых битов равен **0**, то элемента **точно нет**

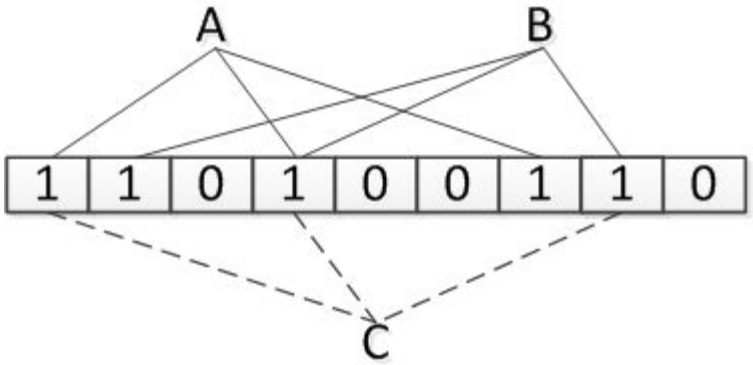


бит на месте одной из хеш-функций для **w** равен **0**, поэтому элемента **w** в множестве **точно нет**

Структура фильтра Блума

Теперь для того, чтобы понять, есть ли элемент **x** в множестве, необходимо посмотреть на состояние всех битов на позициях $\mathbf{h_1(x), \dots, h_k(x)}$

- если хотя бы один из проверяемых битов равен **0**, то элемента **точно нет**
- если все проверяемые биты равны **1**, то структура выдаст ответ **“да, элемент есть”**, но это будет **неточной информацией**



здесь все рассматриваемые биты равны **1**, поэтому программа считает, что элемент **C** в множестве есть, **хотя это не так**

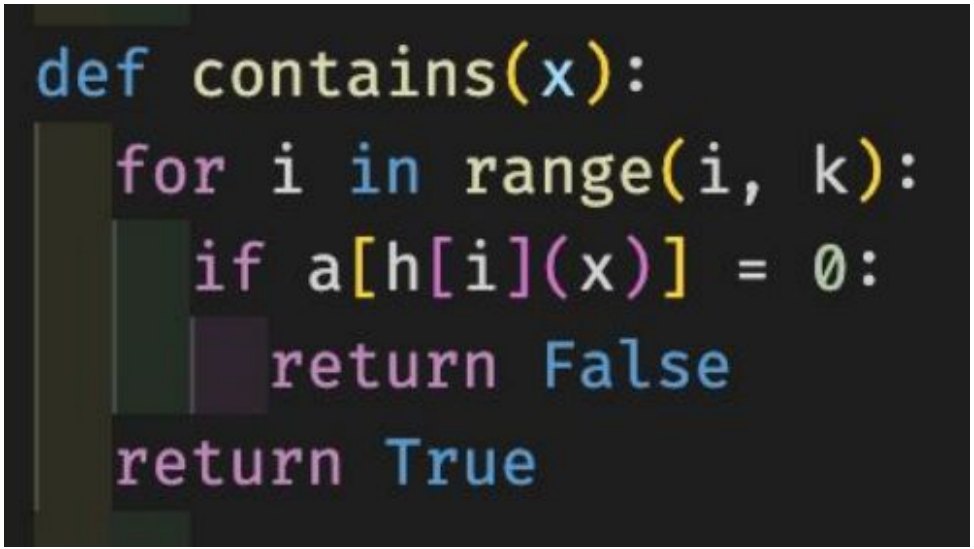
Фильтр Блума. Код

При добавлении устанавливаем **1** во все ячейки с индексом значения очередной хеш-функции

```
def insert(x)
    for i in range(1, k)
        a[H[i](x)] = 1
```

Фильтр Блума. Код

При проверке смотрим, есть ли хотя бы в какой-то позиции **0**. Если нет, то возвращаем **true**, но помним про ложноположительные срабатывания



Снижение вероятности коллизий

Способ 1: увеличение размера массива

При увеличении размера массива (и, соответственно, новой нормализации хеш-функций) вероятность коллизий уменьшается

Снижение вероятности коллизий

Способ 1: увеличение размера массива

При увеличении размера массива (и, соответственно, новой нормализации хеш-функций) вероятность коллизий уменьшается

Вероятность ложного срабатывания (фильтр Блума возвращает **1**, когда элемент отсутствует в наборе) составляет **(1 - e^(m / n))**, где **m** — количество элементов, которые предполагается внести в фильтр, а **n** размер фильтра.

Снижение вероятности коллизий

Способ 1: увеличение размера массива

При увеличении размера массива (и, соответственно, новой нормализации хеш-функций) вероятность коллизий уменьшается

Вероятность ложного срабатывания (фильтр Блума возвращает **1**, когда элемент отсутствует в наборе) составляет **(1 - e^(-m / n))**, где **m** — количество элементов, которые предполагается внести в фильтр, а **n** размер фильтра.

При увеличении **n, e^(-m / n)** также увеличивается, а значит, **вероятность уменьшается**

Снижение вероятности коллизий

Способ 2: увеличение количества хеш-функций

Вероятность ложного срабатывания при **k** хеш-функциях составляет

(1 - e^(-mk / n)) ^ k

Снижение вероятности коллизий

Способ 2: увеличение количества хеш-функций

Вероятность ложного срабатывания при **k** хеш-функциях составляет

(1 - e^(-mk / n)) ^ k

Для того, чтобы понять, какое число хеш-функций является оптимальным, необходимо найти производную формулы выше, приравнять ее к **0** и найти корни

$$\left(1-e^{-(k\,m)/n}\right)^k\left(\frac{k\,m\,e^{-(k\,m)/n}}{n\left(1-e^{-(k\,m)/n}\right)}+\log\left(1-e^{-(k\,m)/n}\right)\right)=0$$

Снижение вероятности коллизий

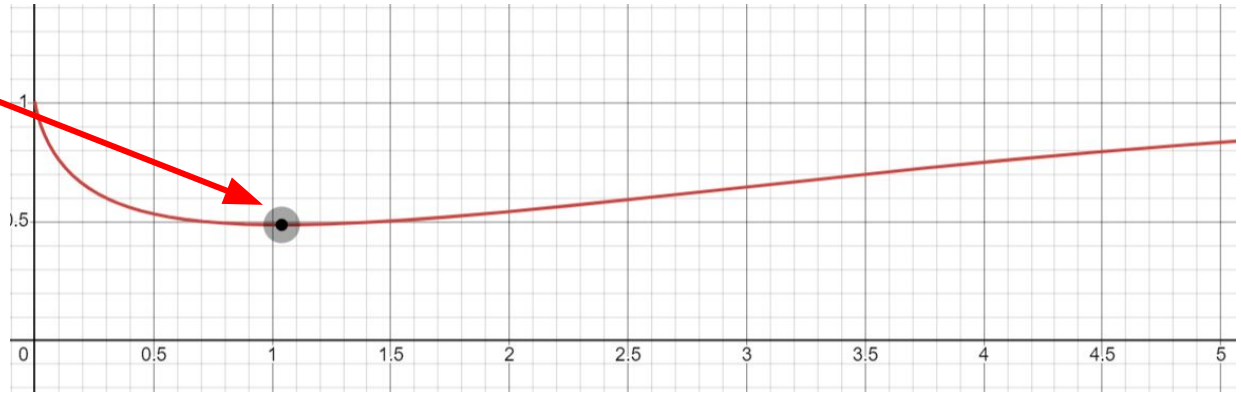
Способ 2: увеличение количества хеш-функций

Вероятность ложного срабатывания при **k** хеш-функциях составляет

(1 - e^(-mk / n)) ^ k

Это означает, что оптимальное количество хеш-функций составляет

(n / m) * ln(2)



Снижение вероятности коллизий

При использовании только увеличения числа хеш-функций:

При каждом запросе на отсутствие элемента в множестве мы теперь смотрим на большее количество ячеек, поэтому вероятность хоть в какой-нибудь найти **0** повышается

Снижение вероятности коллизий

При использовании только увеличения числа хеш-функций:

При каждом запросе на отсутствие элемента в множестве мы теперь смотрим на большее количество ячеек, поэтому вероятность хоть в какой-нибудь найти **0** повышается

Однако

Эти же самые хеш-функции мы используем и для добавления элементов в множество, поэтому при добавлении мы ставим большее количество единиц в разные ячейки таблицы, и поэтому увеличивается вероятность того, что при запросе все просматриваемые ячейки будут **1** из-за того, что они были покрыты другими элементами при добавлении

Снижение вероятности коллизий

Вывод: использовать только увеличение числа хеш-функций нельзя

Можно ли использовать только увеличение размера массива?

Снижение вероятности коллизий

Вывод: использовать только увеличение числа хеш-функций нельзя

Можно ли использовать только увеличение размера массива?

На самом деле, можно, однако это не является самым оптимальным методом по 2 причинам:

- если индекс ячейки был достаточно маленьким до взятия по модулю, то он не изменится и при большем модуле, соответственно, коллизия в нем все еще произойдет

Снижение вероятности коллизий

Вывод: использовать только увеличение числа хеш-функций нельзя

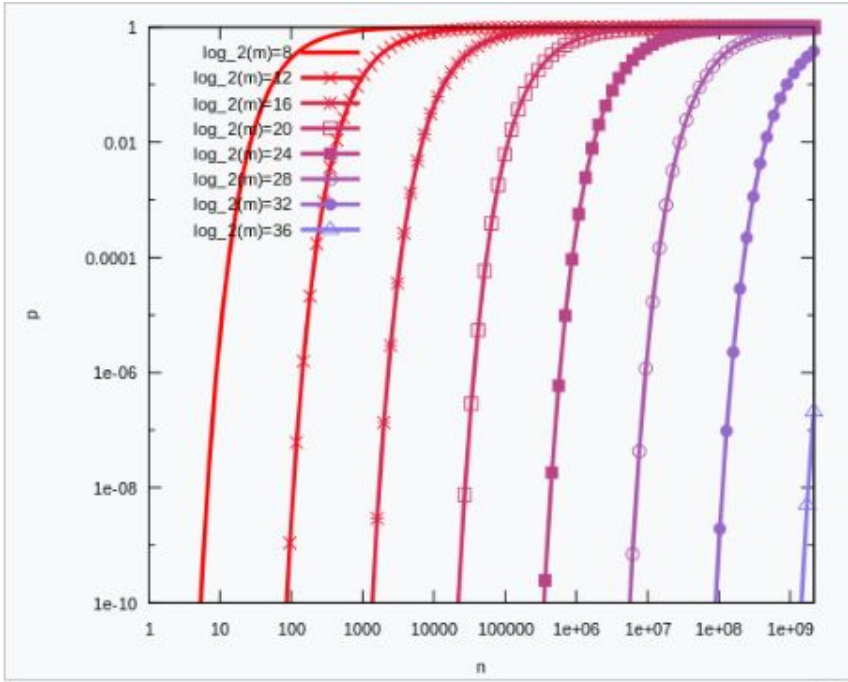
Можно ли использовать только увеличение размера массива?

На самом деле, можно, однако это не является самым оптимальным методом по 2 причинам:

- если индекс ячейки был достаточно маленьким до взятия по модулю, то он не изменится и при большем модуле, соответственно, коллизия в нем все еще произойдет
- при фиксированном размере массива график зависимости числа коллизий от числа хеш-функций примерно представляет собой функцию, которая сначала убывает, а затем возрастает. Соответственно при количестве хеш-функций меньшем или большем, чем оптимальное, мы получаем бОльшую вероятность коллизии

Снижение вероятности коллизий

Вывод: для оптимальной работы фильтра Блума необходимо комбинировать оба метода



Оценка вероятности ложного срабатывания p как функция от числа вставленных элементов n и размера битового массива m, при оптимальном числе хеш-функций $k = (m/n) \ln 2$.

Где применяется фильтр Блума?

- **Поиск в интернете.** Здесь фильтр позволяет поисковой машине не сканировать ресурсы, на которых точно нет информации из поискового запроса. Это экономит время пользователя.

Где применяется фильтр Блума?

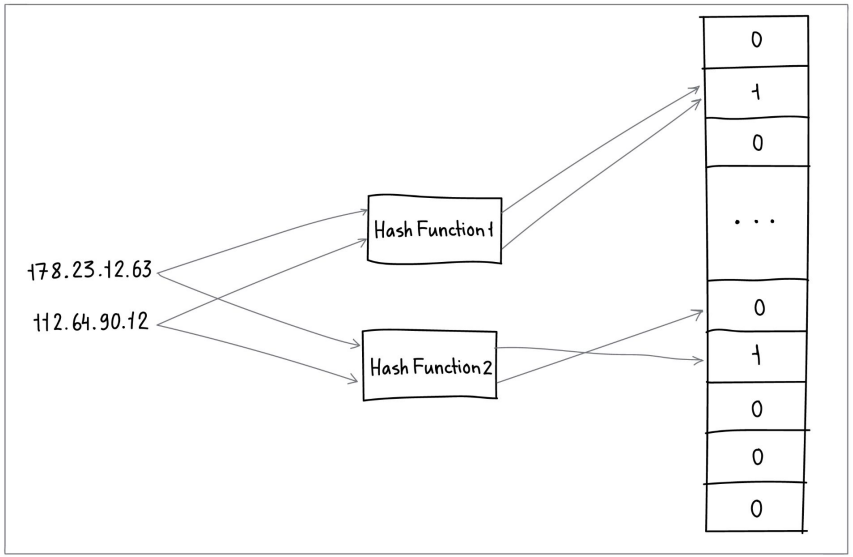
- **Поиск в интернете.** Здесь фильтр позволяет поисковой машине не сканировать ресурсы, на которых точно нет информации из поискового запроса. Это экономит время пользователя.
- **Системы проверки орфографии.** В этом случае компьютер может выделить слово, которого нет в словаре. Так не придется нагружать устройство после каждого нажатия клавиши.

Где применяется фильтр Блума?

- **Поиск в интернете.** Здесь фильтр позволяет поисковой машине не сканировать ресурсы, на которых точно нет информации из поискового запроса. Это экономит время пользователя.
- **Системы проверки орфографии.** В этом случае компьютер может выделить слово, которого нет в словаре. Так не придется нагружать устройство после каждого нажатия клавиши.
- **Криптокошельки.** Фильтр Блума используется, чтобы искать транзакции. В результате с некоторой вероятностью можно утверждать, что транзакция в наличии, или ее нет.

Где применяется фильтр Блума?

- **Таблицы маршрутизации.** Например, черные списки IP-адресов, куда нельзя перенаправлять запросы. Эту информацию нужно получать максимально быстро. Ниже на рисунке пример проверки двух IP-адресов в черном списке:



Где применяется фильтр Блума?

- **В биоинформатике.** Фильтр используется, чтобы искать фрагменты в последовательностях ДНК. Ниже на рисунке видно, как в фильтр Блума добавили цепочку ДНК **ACCTAG** и искали фрагмент **CGTAT**: