Хеш-функция

Хеш-функция — это функция, преобразующая исходные битовые данные произвольной длины (прообраз) в битовую строку фиксированной длины (хеш-образ). Вычисляется за не более чем полиномиальное время.

Что можно хешировать?

- **Строки**
- **Фотографии**
- **Программное обеспечение** (например, чексумма для проверки корректности установки)

Области применения

- **Проверка целостности и подлинности** сообщений
- **Защита паролей** при аутентификации
- **Ускорение поиска** данных (хеш-таблицы)
- **Распознавание вредоносных файлов** (например, набор данных NSLR)

Криптографические хеш-функции

Криптографические хеш-функции — это подкласс хеш-функций, обладающий особыми свойствами, необходимыми для использования в криптографии. При разработке современного российского стандарта ГОСТ Р 34.11-2012 «Стрибог» к ним предъявляются следующие требования:

1. Стойкость к вычислению прообраза

> Если известно значение хеша h, должно быть **трудно** найти какое-либо сообщение m, такое что
>
> $$\mathrm{hash}(m) = h.$$

2. Стойкость ко второму прообразу

> Если известно сообщение m_1 и его хеш $\mathrm{hash}(m_1)$, должно быть **трудно** найти другое сообщение $m_2 \neq m_1$, такое что
>
> $$\mathrm{hash}(m_2) = \mathrm{hash}(m_1).$$

3. Стойкость к поиску коллизий

> Должно быть **трудно** найти **любую** пару различных сообщений $m \neq m'$, для которых
>
> $$H(m) = H(m').$$
>
> **Пример коллизии**
>
> Пусть длина прообраза — 6 бит, длина хеша — 4 бита. Тогда:
>
> - Всего возможных значений хеша: $2^4 = 16$
> - Всего возможных прообразов: $2^6 = 64$
>
> Так как $64 > 16$, по принципу Дирихле найдутся коллизии: некоторые хеш-значения будут соответствовать сразу нескольким (в нашем примере — четырём) прообразам.

4. Стойкость к удлинению прообраза

> Стойкость к удлинению прообраза: если злоумышленник не знает сообщение, но знает его длину и хеш-код от него, то ему должно быть сложно подобрать такое сообщение, которое, будучи дописанным к оригинальному, даст какую-нибудь известную хеш-функцию. Другими словами, не должно быть возможно злоумышленнику что-то менять путём дополнения в сообщении, получая известны выход. Это можно сформулировать по-другому: хеш-функция не должна быть хорошо «дополняема»

Поиск пароля по хэш-функции с использованием цепочек и функции редукции

Формальная постановка задачи

Пусть:

- H — хеш-функция, выдающая значения длины n

- P — конечное множество возможных паролей

Задача: создать структуру данных, которая для любого значения хеша h сможет:

- найти такой элемент $p \in P$, что $H(p) = h$, или
- определить, что такого элемента не существует

Проблема: прямой перебор всех $p \in P$ и вычисление $H(p)$ слишком трудоёмкий.

Функция редукции

Вводится функция редукции R:

$R : H \to P$, отображающая з В большинстве стандартных библиотек (C++, Rust, Go) используется открытая адресация линачения хеша обратно в пространство паролей.

Цепочка редукций и хешей

Для построения структуры:

- выбирается множество начальных паролей
- для каждого из них строится цепочка чередующихся операций хеширования и редукции:

$$aaaaaa \to^{H} 281DAF40 \to^{R} sgfnyd \to^{H} 920ECF10 \to^{R} kiebgt$$

Сохраняются:

- начальный пароль
- конечное значение цепочки

Пример поиска пароля

Имеем хеш: $920ECF10$

Находим цепочку, в конце которой получается $R(920ECF10) = kiebgt$.

Находим цепочку, заканчивающуюся на `kiebgt`, и восстанавливаем её сначала:

$$aaaaaa \to H281DAF40 \to Rsgfnyd \to H920ECF10$$

Значит, искомый пароль — `sgfnyd`

Алгоритм поиска пароля

1. Построить цепочки из случайного набора начальных паролей

2. Для каждого заданного хеша `h`:

- Последовательно вычислять значения $R(H(R(h)))$, $R(H(R(H(R(h)))))$, и т.д.
- Если найденное значение совпадает с концом какой-либо цепочки:
- Берём начало этой цепочки и восстанавливаем её, пока не встретим $hash$
- Предшествующий элемент — искомый пароль

Проблемы

- **Подбор функции редукции R**:
- должна равномерно распределять значения в пространстве паролей
- должна покрывать достаточное подмножество паролей
- **Слияние цепочек**:
- при коллизиях H и R цепочки могут сливаться, теряя уникальность и ухудшая покрытие

Взлом с помощью радужных таблиц

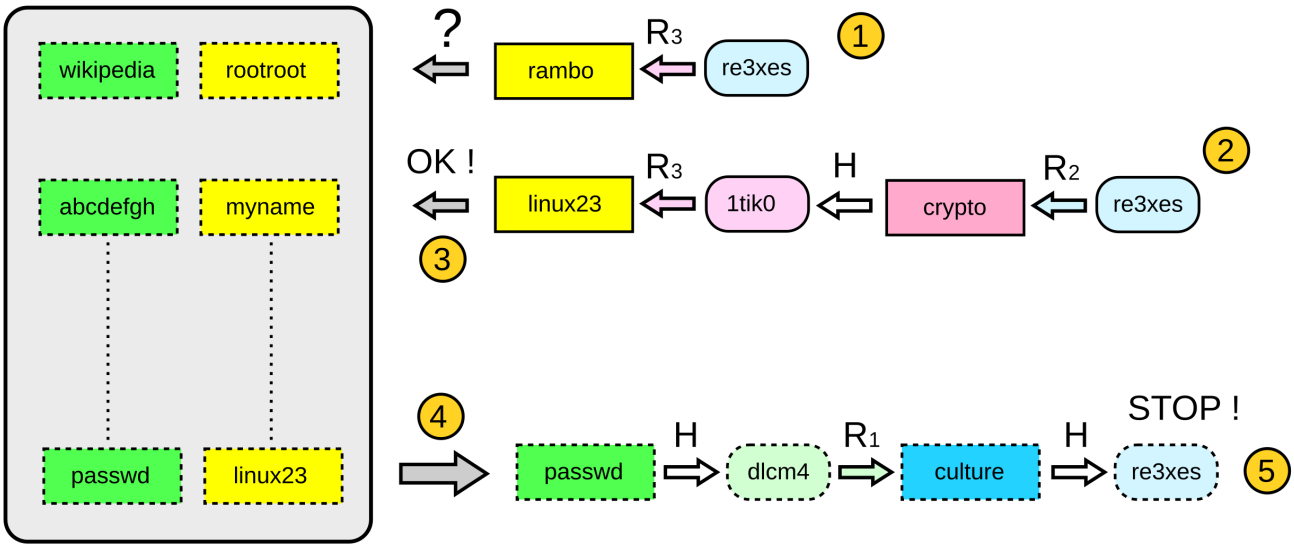
Предположим, что у нас имеется не одна функция редукции $R : H \to P$, а целый набор R_k. Для каждого пароля мы строим **цепочку** из чередующихся операций хеширования и редукции:

Шаг 1: $R_k(H)$

Шаг 2: $R_{k-1}(H(R_k(H(password))))$

Шаг k: $R_1(H(R_2(\ldots(R_k(H(password))))))$

Поиск исходного пароля в радужной таблице



Если на каком-то шаге результат применения функции редукции совпал с **хвостом** одной из цепочек в радужной таблице, мы переключаемся на *обратный* поиск: продолжаем вычислять пары «хеш → редукция», пока не получим исходный хеш. Формально, мы ищем индекс i, такой что

$$H(R_i(\dots H(R_1(H(password)))))=H_{\text{исходный}}$$

и берём пароль, предшествующий этому хешу (в примере выше — `culture`).

Криптографическая соль: главное средство защиты

Радужные таблицы становятся практически бесполезными, если вместе с паролем хешируется уникальная для каждого пользователя **соль**:

$$H = H(password\ +\ salt)$$

Из-за соли одно и то же слово порождает разные хеши, а предрассчитанную таблицу придётся строить отдельно для каждой соли, что экономически невыгодно.

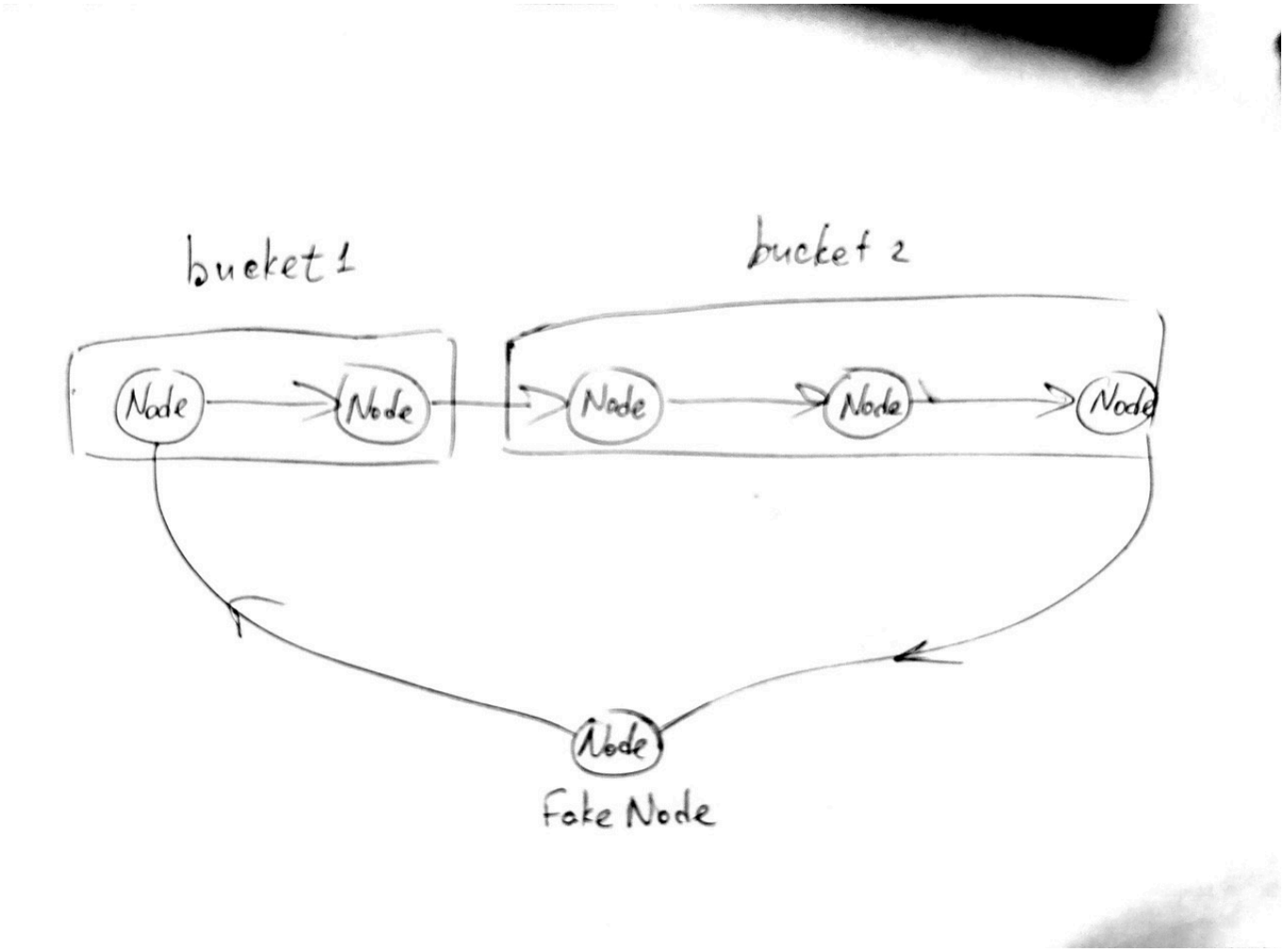
Преимущества радужных таблиц

- По времени генерация длиннее, чем простая таблица «хеш → пароль», но **памяти** требуется существенно меньше: с $O(N)$ до $O(N^{2/3})$

Недостатки

- Таблица применима **только** к тому алгоритму хеширования, для которого была создана.
- Эффективность быстро падает, если в пароле используется соль или pepper.

Хэш-таблица



Описание реализации (метод цепочек)

В данной реализации хеш-таблица организована с использованием массива **бакетов**, где каждый бакет представляет собой односвязный список (`std::forward_list`). Элементы с одинаковым индексом бакета (в результате коллизий) хранятся в соответствующей цепочке.

Структура узла

```
template <typename Key, typename Value> // шаблонные параметры для
ключа+значения
struct Node {
    std::pair<const Key, Value> kv;  // ключ и значение
    std::size_t hash;                // заранее вычисленный хеш ключа
};
```

Хранение хеша внутри узла позволяет при обработке коллизий быстрее находить нужный элемент без повторного вычисления хеш-функции.

Вставка пары `(key, value)`

Алгоритм вставки работает следующим образом:

1. **Вычисляется хеш ключа** с помощью заданной хеш-функции:

```
std::size_t hash = hasher(key);
```

2. **Определяется индекс бакета, куда должен быть помещён элемент:**

```
std::size_t index = hash % N;  // N — общее число бакетов
```

3. **Выбирается соответствующий бакет**, который является цепочкой (односвязным списком).
4. **Если в цепочке уже есть элементы (коллизия)**, то:
   - Новый элемент просто добавляется в цепочку (чаще всего в начало — это эффективнее по времени для `forward_list`, но возможно и в конец).
   - Для ускорения добавления в конец цепочки можно дополнительно хранить указатель на последний элемент каждой цепочки.

Удаление

По хешу попадаем в нужный бакет, находим элемент, *перешиваем* ссылки и при необходимости обновляем указатель на хвост цепочки.

Перестройка таблицы

Когда коэффициент заполнения `load_factor` достигает `max_load_factor`, таблица **перехешируется** — размер массива бакетов увеличивается, и для всех ключей пересчитываются индексы.

> **Сложность операций**: средняя $O(1)$, худшая $O(n)$ при длинной цепочке.

Открытая адресация

Линейное, квадратичное и двойное хеширование

Вместо цепочек все элементы хранятся **внутри массива**. При коллизии выполняется последовательность проб:

$h_0(x),\ h_1(x),\ \ldots,\ h_n(x)$

- **Линейное пробирование**: $h_i(x) = (h(x) + i) \bmod N$
- **Квадратичное**: $h_i(x) = (h(x) + c \cdot i + c \cdot i^2) \bmod N$
- **Двойное хеширование**: $h_i(x) = (h(x) + i \cdot h_{new}(x)) \bmod N$

Удаление (tombstone)

Чтобы не разрывать цепочку проб, удалённая ячейка помечается как *занятая-ранее* (tombstone).

- При **поиске** такие ячейки считаются занятыми.
- При **вставке** первая томб-ячейка на маршруте используется под новый элемент.

Плюсы и минусы двух методов разрешения коллизий

Критерий	Метод цепочек	Открытая адресация
Память	Требуется указатель на следующую запись; массив + список	Всё в одном массиве, без указателей
Производительность при низком `load_factor`	$O(1)$ вставка/поиск	$O(1)$ вставка/поиск
Производительность при высоком `load_factor`	Цепочки растут $\rightarrow O(n)$	Увеличивается длина проб $\rightarrow O(n)$
Удаление	Простейшее, за $O(1)$	Требуются tombstones или перестройка
Поддержка итераторов	Легко и стабильно	Сложнее, перестройка может инвалидировать
Подходит для внешней памяти	Да (списки можно хранить на диске)	Обычно нет
Реорганизация (rehash)	Необязательна до разумного `load_factor`	Критична: поиски резко деградируют

Выбор метода зависит от требований к памяти, частоты вставок/удалений и предельного `load_factor`. В большинстве стандартных библиотек (C++, Rust, Go) используется

открытая адресация либо hybid-подход с короткими цепочками. Для неизменяемых структур или работы с внешней памятьью удобнее классические цепочки.