

Основы программирования

Лекция 5 Хеширование

Введение

Процесс поиска данных в больших объемах информации сопряжен с временными затратами, которые обусловлены необходимостью просмотра и сравнения с ключом поиска значительного числа элементов. Сокращение поиска возможно осуществить путем локализации области просмотра.

Например, отсортировать данные по ключу поиска, разбить на непересекающиеся блоки по некоторому групповому признаку или поставить в соответствие реальным данным некий код, который упростит процедуру поиска.

В настоящее время используется широко распространенный метод обеспечения быстрого доступа к информации, хранящейся во внешней памяти, который называется - хеширование.

Хеширование используется для ускорения поиска и добавления данных в структуру данных, в работе программного кэша, индексов баз данных, в языковых процессорах (например в компиляторах). Оно позволяет быстро найти элемент по его хеш-коду, что особенно полезно, когда количество элементов в структуре данных велико.

Возможно Вам знакома такая штука как *ассоциативный массив* и, в зависимости от языка программирования он может быть Вам также известен под названием *map*, *отображение* или *словарь*. Но как бы он не назывался – в своей основе он обычно также использует хеш-таблицы.

Хеш-таблицы

Хеш-таблица – это структура данных, реализующая интерфейс **ассоциативного массива** (ещё названия **map**, **hashmap**, **словарь**).

Хеш-таблица – это **обычный массив с необычной адресацией**, задаваемой хеш-функцией.

Хеш-таблица – это **массив, формируемый в определенном порядке хеш-функцией**.

Она позволяет хранить пары вида **“ключ - значение”** и выполнять операции (в среднем за время $O(1)$):

- добавление новой пары;
- поиск;
- удаление пары по ключу.

Хеш-таблица является массивом, формируемым хеш-функцией в определённом порядке.

- Доступ к элементам осуществляется по ключу (key)
- Ключи могут быть строками, числами, указателями

При создании таблицы очень важно, чтобы таблица не меняла свой размер (иначе придётся пересчитывать хэши)

Применение хеш-

таблиц
Хеш-таблицы имеют очень большое практическое применение:

- Базы данных
- Языковые процессоры (компиляторы, ассемблеры) – повышение скорости обработки таблицы идентификаторов
- Распределение книг в библиотеке по тематическим каталогам
- Упорядочение слов в словарях
- Шифрование специальностей в вузах, паролей
- Для поиска информации о водителе лишь по его номеру в водительском удостоверении.
- Таблица символов компилятора.
- При программировании шахмат тоже используется хеширование
- Для баз данных телефонных номеров.
- Для хранения паролей пользователей.

Хеш-таблицы

Основные правила использования хеш-таблиц

1. Выполнение операции в хеш-таблице начинается с вычисления хеш-функции от ключа. Получающееся хеш-значение является индексом в массиве.

2. Количество хранимых элементов массива, деленное на число возможных значений хеш-функции, называется коэффициентом заполнения хеш-таблицы (load factor) и является важным параметром, от которого зависит среднее время выполнения операций.

3. Операции поиска, вставки и удаления должны выполняться в среднем за время $O(1)$. При такой оценке не учитываются возможные аппаратные затраты на перестройку индекса хеш-таблицы, связанную с увеличением размера массива и добавлением в хеш-таблицу новой пары.

4. Механизм разрешения коллизий является важной составляющей любой хеш-таблицы.

Хеширование

Хеширование (или хэширование, англ. hashing, hash – крошить, перемешивать, рубить на куски) – это преобразование входного массива данных определенного типа и произвольной длины в выходную битовую строку фиксированной длины.

Это процесс получения индекса (хеш-адреса) элемента массива непосредственно в результате операций производимых над ключом, который хранится вместе с элементом.

Хеширование используется для ускорения поиска и добавления данных в структуру данных. Оно позволяет быстро найти элемент по его хеш-коду, что особенно полезно, когда количество элементов в структуре данных велико.

Такие преобразования также называют хеш-функциями (или функций свертки), а их результаты называют **хеш**, **хеш-код** или **хеш-таблицей** (или дайджестом сообщения (англ. message digest (digest – краткое изложение, справочник))).

Хеширование применяется для сравнения данных:

- если у двух массивов хеш-коды разные, то массивы гарантированно различаются;
- если у двух массивов хеш-коды одинаковые, то массивы, **скорее всего**, одинаковы.

Хеширование

Основные понятия

Пространством ключей называется множество всех теоретически возможных значений ключей записи.

Пространством записей называется множество тех ячеек памяти (адресов), которые выделяются для хранения таблицы.

Таблицы прямого доступа применимы только для таких задач, в которых размер пространства записей равен размеру пространства ключей.

В большинстве реальных задач, **размер пространства записей** много **меньше**, чем размер пространства ключей.

Пример. Если в качестве ключа используется фамилия, то, даже ограничив длину ключа 10 символами кириллицы, получаем 33^{10} возможных значений ключей.

Значительная часть пространства записей в примере будет заполнена пустыми записями, так как в каждом конкретном заполнении таблицы **фактическое множество ключей будет меньше пространства ключей.**

Из соображений экономии памяти целесообразно назначать **размер пространства записей равным размеру фактического множества записей или превосходящим его незначительно.**

Введение

Представим что у нас есть таблица в которой записаны имена людей и их номера телефонов. Там может содержаться также и любая другая информация, но для простоты оставим только номер. И в этой таблице мы хотим совершать поиск по некоторому **ключу**, а ключом у нас будет являться **имя**. Проще говоря, мы хотим находить по имени человека его номер телефона.

Как мы это можем сделать? Например, мы можем действовать простым перебором, то есть перебирать все записи по очереди и проверять в каждой из них имя. И делать так, до тех пор пока, не найдем нужную запись.

Как Вы понимаете это очень медленный способ, и, к примеру, если у нас миллионы записей, то нам придется перебирать все эти миллионы при каждом поиске.

А вот если бы мы знали идентификатор (обычно пишется - id) искомой записи, то мы находили бы нужные данные гораздо быстрее, а если бы знали адрес ячейки памяти, где лежат эти данные – то нашли бы практически мгновенно. Причём эта скорость не зависела от количества данных, то есть

ИМЯ	Номер
Петя	32-32-25
Вася	44-11-56
Маша	11-11-45
Саша	11-33-23
Коля	78-96-11
Степа н	88-88-01

id	ИМЯ	Номер
0	Петя	32-32-25
1	Вася	44-11-56
2	Маша	11-11-45
3	Саша	11-33-23
4	Коля	78-96-11
5	Степа н	88-88-01

Введение

Однако мы можем схитрить. Мы можем заполнить таблицу заново таким образом, *чтобы индекс каждого элемента вычислялся с помощью его значения.*

Допустим у нас есть пустая таблица и элемент который мы хотим в нее сохранить, но на этот раз мы сохраним запись в ней не в первую попавшуюся ячейку, а в ячейку с тем индексом который мы вычислим из нашего ключа (имени). В будущем, зная это **ИМЯ** или же **КЛЮЧ** мы сможем точно так же вычислить индекс и сразу получить значение.

Как же получить из текстовых данных число? На самом деле способов много. Один из самых простых - это разбить строку на символы и с помощью какой-нибудь из таблиц кодировок сопоставить с каждым символом его числовой код. Потом мы просуммируем все полученные значения и получим итоговые. Хотелось бы сказать что это число и будет нашим индексом, но она нам не подходит: как видим число очень большое и, в некоторых случаях, оно может быть даже еще больше. Для того, чтобы получить индекс, который не превышает размеры таблицы, мы просто возьмем остаток от деления этого числа на размер таблицы. В итоге получаем число 5 и

CP-1251

$$\begin{array}{r} \text{С} - 209 \\ + \\ \text{А} - 224 \\ + \\ \text{Ш} - 248 \\ + \\ \text{А} - 224 \end{array} \left. \vphantom{\begin{array}{r} \text{С} - 209 \\ + \\ \text{А} - 224 \\ + \\ \text{Ш} - 248 \\ + \\ \text{А} - 224 \end{array}} \right\} 905 \bmod 6 = 5$$

Id ?
Саша
11-33-23

id	ИМЯ	Номер
0		
1		
2		
3		
4		
5	Саша	11-33-23

Введение

Внимательно посмотрим, что мы только что сделали.

У нас был текстовый ключ, мы проделали с ним ряд операций и на выходе получили число. Так вот, все те наши действия, которые преобразовали текст в число называются хэш-функцией.

Различных хэш-функций существует довольно много и сейчас мы рассмотрели только одну из них. К примеру, если бы у нас ключом было не **имя** человека, а его **номер телефона**, то нам даже не пришлось бы каждому символу сопоставлять какое-то число – мы могли просто взять и сложить все цифры из этого номера, и, в результате, мы также получили бы на выходе какое-то итоговое числовое значение.

Пока мы оставим хэш-функции в покое, но чуть позже мы к ним вернемся, чтобы обсудить критерии по которым можно их оценивать.

Хеш-таблицы (Hash tables)

Ключ
(Key)

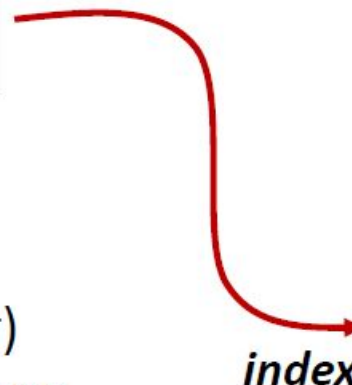
Хеш-функция
(Hash function)

Хеш-таблица
(Hash table)

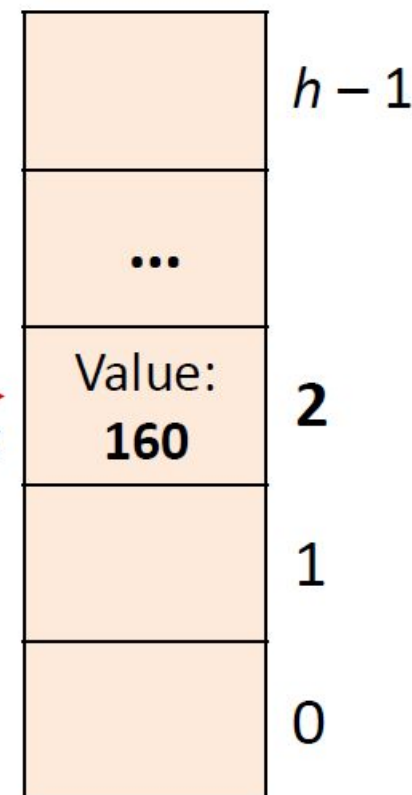
“Антилопа Гну”



`hash(key) -> int`



- **Хеш-функция** – отображает (преобразует) ключ (key) в номер элемента (index) массива (в целое число от 0 до $h - 1$)
- Время вычисления хеш-функции зависит от длины ключа и не зависит от количества элементов в массиве
- Ячейки массива называются buckets, slots



Хеш - значение

Функцией хеширования (функцией перемешивания, функцией рандомизации) называется функция, обеспечивающая отображение пространства ключей K в пространство записей A (т. е. преобразование ключа в адрес записи):

$$h: K \rightarrow A;$$

$$a = h(k), \text{ где } a - \text{адрес, } k - \text{ключ.}$$

Один из наиболее распространенных **алгоритмов хеширования** для строк получает **хэш-значение**, добавляя каждый байт строки к произведению предыдущего значения на некий **фиксированный множитель (хэш)**. Умножение распределяет биты из нового байта по всему до сих пор не считанному значению, так что в конце цикла мы получим хорошую смесь входных байтов. **Эмпирически** установлено, что значения **31, 33, 37, 39, 41** являются хорошими множителями в хэш-функции для строк ASCII.

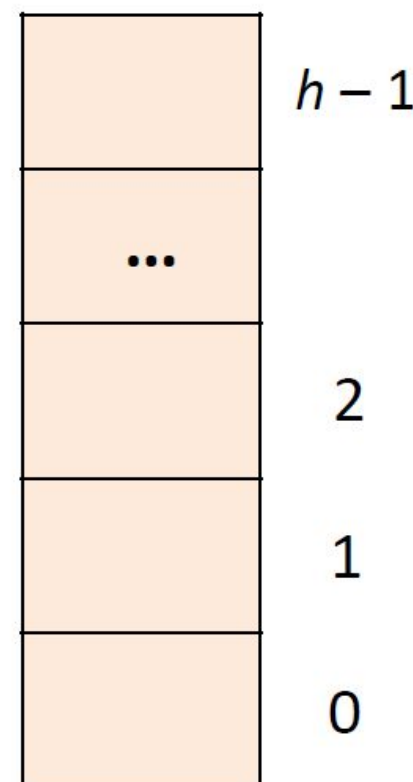
Хеш-таблицы (Hash tables)

- На практике, обычно известна информация о диапазоне значений ключей
- На основе этого выбирается размер h хеш-таблицы и выбирается хеш-функция
- **Коэффициент α заполнения хеш-таблицы (load factor, fill factor)** – отношение числа n хранимых элементов в хеш-таблицы к размеру h массива (среднее число элементов на одну ячейку)

$$\alpha = \frac{n}{h}$$

- Пример: $h = 128$, в хеш-таблицу добавили 50 элементов, тогда $\alpha = 50 / 128 \approx 0.39$
- От этого коэффициента зависит среднее время выполнения операций добавления, поиска и удаления элементов

Хеш-таблица (Hash table)



Методы создания хеш-функций:

- остатков от деления;
- функции середины квадрата;
- свертки;
- преобразования системы счисления.

Метод остатков от деления

- Остаток от деления целочисленного ключа **Key** на размерность массива **HashTableSize**:

$$\text{Key} \% \text{HashTableSize}$$

Результат – адрес записи в хеш-таблице.

Эта функция очень проста.

Для минимизации коллизий рекомендуется, чтобы размерность таблицы была простым числом.

Обычно операция деления по модулю применяется как последний шаг в более сложных функциях хеширования.

Метод остатков от деления. Пример

Пусть ключом является символьная строка.

Тогда хеш-код для нее – это остаток от деления суммы кодов литер, образующих строку, на размер таблицы.

Например,

$S = \text{"olympiad"}$, $\text{HashTableSize} = 100$,

о	l	у	m	p	i	a	d
111	108	121	109	112	105	97	100

Сумма кодов равна 863.

Хеш этой строки равен $863 \% 100 = 63$.

Функция середины квадрата

- преобразует значение ключа в число,
- ВОЗВОДИТ ЭТО ЧИСЛО В КВАДРАТ,
- из полученного числа выбирает несколько средних цифр,
- интерпретирует эти цифры как адрес записи.

Метод свертки

- Цифровое представление ключа разбивается на части, каждая из которых имеет длину, равную длине требуемого адреса.
- Над частями производятся определенные арифметические или поразрядные логические операции, результат которых интерпретируется как адрес.

Например, сумма кодов символов строки-ключа.

Функция преобразования системы счисления

- Ключ, записанный как число в системе счисления с основанием P , интерпретируется как число в системе счисления с основанием $Q > P$. Обычно выбирают $Q = P + 1$.
- Это число переводится из Q -с.с. в P -с.с., приводится к размеру пространства записей и интерпретируется как адрес.

Пусть $P = 2$, $Q = 3$. Ключ = $101101_{(2)}$

Значение этого числа в 3-с.с. = $3^5 + 3^3 + 3^2 + 1 = 243 + 27 + 9 + 1 = 280$

Тогда представление 280 в 2-с.с. будет: $100011000_{(2)}$

Чтобы привести это число к размеру пространства записей, мы можем использовать свертку, которая заключается в сложении соседних цифр числа. Разобьём его на группы по 3 цифры: 100 011 000 . Тогда свертка: $100 + 11 + 0 = 111$.

Затем мы вычисляем остаток от деления этого числа на размер пространства записей, например у нас это будет $100: 111 \% 100 = 11$.

Таким образом, мы получили хеш-код, который мы можем использовать для сохранения данных в пространстве записей размером 100. Этот хеш-код будет уникальным для каждого ключа, и мы можем использовать его для быстрого поиска данных в пространстве записей.

Хеш-функция Дженкинса

Эта хеш-функция была разработана Бобом Дженкинсом и широко используется для вычисления хеш-кодов в различных приложениях, таких как обработка текста, сетевые протоколы и т.д.

Функция принимает два аргумента:

- key - указатель на массив байтов, который нужно преобразовать в хеш-код.
- len - длина массива байтов.

Внутри функции объявляется переменная hash типа uint32_t, которая инициализируется нулем. Затем происходит цикл, в котором происходит обработка каждого байта массива key.

После окончания цикла выполняются еще три операции над хеш-кодом:

1. выполняет сдвиг хеш-кода влево на 3 бита и добавляет результат к текущему хеш-коду.
2. выполняет сдвиг хеш-кода вправо на 11 бит, выполняет операцию XOR над результатом и текущим хеш-кодом и записывает результат в хеш-код.
3. выполняет сдвиг хеш-кода влево на 15 бит и добавляет результат к текущему хеш-коду.

После всех операций хеш-код

```
uint32_t jenkins(uint8_t *key, size_t len)
{
    uint32_t hash = 0;
    for (int i = 0; i < len; i++)
    {
        hash += key[i]; // добавляет текущий байт
                        // к хеш-коду
        hash += (hash << 10); // сдвиг хеш-кода
                             // влево на 10 бит и добавляет
        // результат к текущему хеш-коду
        hash ^= (hash >> 6); // выполняет сдвиг
                             // хеш-кода вправо на 6 бит, выполняет операцию
        // XOR над результатом и текущим хеш-кодом и
        // записывает результат в хеш-код
    }
    hash += (hash << 3);
    hash ^= (hash >> 11);
    hash += (hash << 15);
    return hash;
}
```

Еще пример хеш-функции

Эта программа представляет собой реализацию хеш-функции, которая принимает 32-битное целое число в качестве входного ключа и возвращает 32-битное хеш-значение.

Функция состоит из следующих шагов:

1. Сдвиг вправо на 16 бит: $n \gg 16$. Этот шаг сдвигает биты ключа вправо на 16 позиций, тем самым отбрасывая младшие 16 бит.
2. Побитовое исключающее ИЛИ (XOR) с исходным ключом: $n \wedge (n \gg 16)$. Этот шаг выполняет побитовое XOR между исходным ключом и результатом предыдущего шага, тем самым перемешивая биты ключа.
3. Умножение на константу: $n * 0x45D9F3B$. Этот шаг умножает текущее значение ключа на константу $0x45D9F3B$, которая является простой константой, выбранной для обеспечения лучшего перемешивания битов.

Повторение шагов 1-3 еще два раза.

Возврат хеш-значения: `return n`; Этот шаг возвращает полученное хеш-значение.

Эта хеш-функция является довольно простой и быстрой, но в то же время обеспечивает достаточно хорошее перемешивание битов ключа, чтобы минимизировать количество коллизий при использовании в хеш-таблицах. Она была разработана Томом Бендерсоном и представлена в его статье "Простая и быстрая хеш-функция" (Simple and Fast Hash Function).

```
uint32_t hash32(uint32_t n)
{
    n = (n >> 16) ^ n;
    n = n * 0x45D9F3B;
    n = (n >> 16) ^ n;
    n = n * 0x45D9F3B;
    n = (n >> 16) ^ n;
    return n;
}
```

Хеш-функция Кнута

Пусть x – целое, q – константа $\in \mathbb{R}$ – иррац.

$$f(x) = (q \cdot x) \bmod 1$$

$$h(x) = \lfloor ((q \cdot x) \bmod 1) \cdot m \rfloor$$

$$\lfloor q \cdot 2^{32} \rfloor = A - \text{целое}, q = \frac{\sqrt{5}-1}{2} - \text{золотое сечение}$$

$$m = 2^s$$

$$h(x) = \lfloor ((A \cdot x) \bmod 2^{32}) \cdot (m / 2^{32}) \rfloor$$

```
uint32_t knuth(uint32_t x) {  
    const uint32_t A = 2654435769;  
    uint32_t res = A * x;  
    return res >> (32 - s);  
}
```

Полиномиальная хеш-функция

Пусть $x = (x_0, x_1, x_2 \dots x_{l-1})$

p – простое число ($10^9 + 7$)

b – некоторое целое число (например, 31)

$$h(x) = (\sum_{i=0}^{l-1} x_i b^i) \bmod p$$

Используется в алгоритме Рабина-Карпа
(поиск подстроки в строке)

Введение

Теперь заполняем нашу хэш-таблицу и разложим по ячейкам остальные наши записи.

Первым делом разбиваем их на символы и складываем числа соответствующей каждому символу. Далее вычисляем остаток от деления на размер таблицы каждого из итоговых чисел :

Петя -> $207 + 229 + 242 + 255 = 933 \bmod 6 = 3$

Вася -> $194 + 224 + 241 + 255 = 914 \bmod 6 = 2$

Маша -> $204 + 224 + 248 + 224 = 900 \bmod 6 = 0$

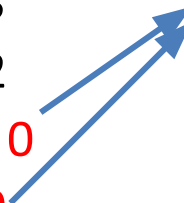
Коля -> $202 + 238 + 235 + 255 = 930 \bmod 6 = 0$

Таким образом сохраняем каждую из записей в соответствующую ячейку.

В начале всё идёт хорошо, но в какой-то момент нам встречается ситуация когда две разные записи и хотят попасть в одну и ту же ячейку.

На самом деле, это вполне ожидаемая ситуация, и такой случай называется **коллизией**. Коллизии довольно часто встречается при работе с хеш-таблицами, и конечно же придуман ряд методов для работы с ними.

id	ИМЯ	Номер
0	Маша	11-11-45
1		
2	Вася	44-11-56
3	Петя	32-32-25
4		
5	Саша	11-33-23



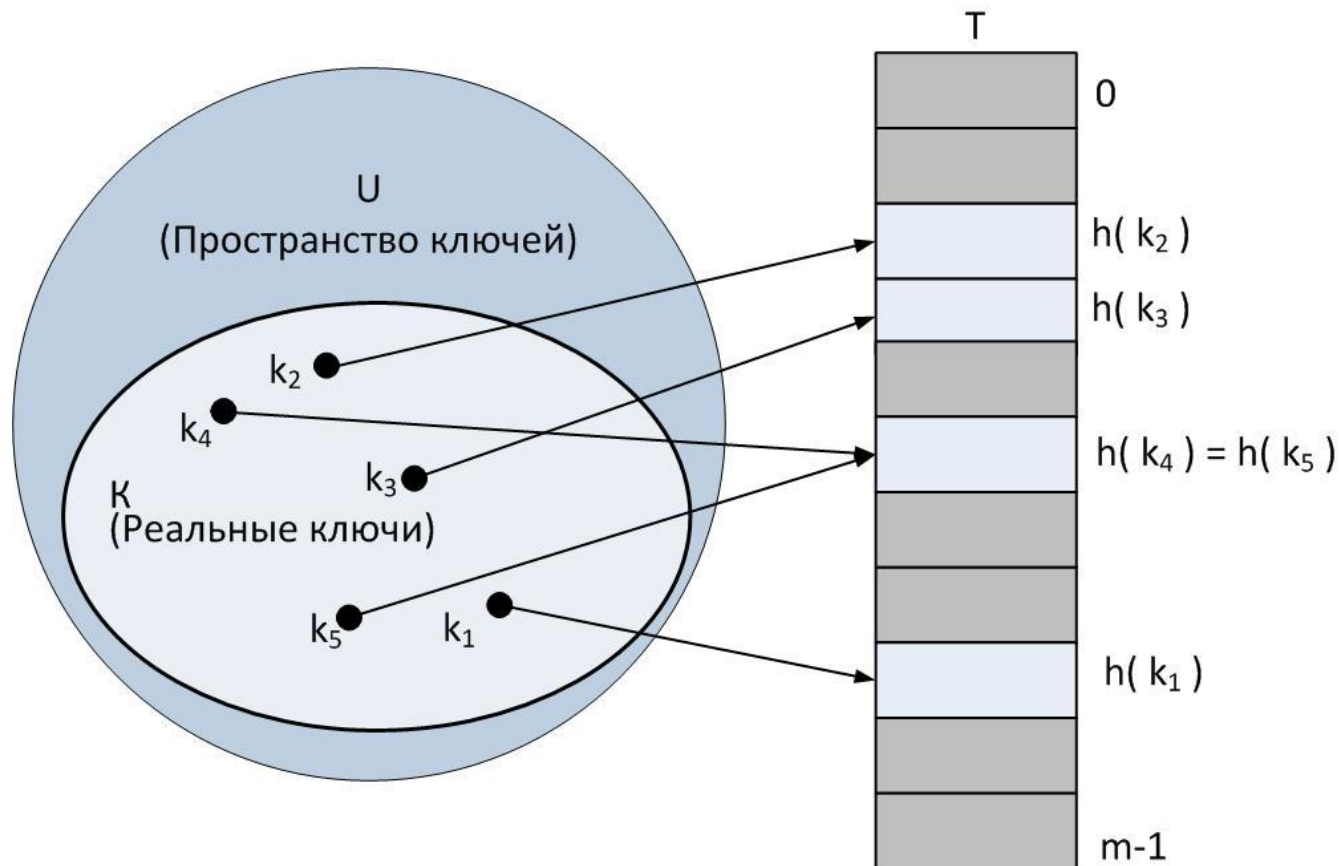
Коллизии

Существует множество пар “ключ - значение”, дающих одинаковые хеш-коды. В этом случае возникает **КОЛЛИЗИЯ**.

Вероятность возникновения коллизий важна при оценке качества хеш-функций. Существует множество алгоритмов хеширования с различными характеристиками.

Выбор хэш-функции определяется спецификой решаемой задачи.

Хеш-таблицы и коллизии



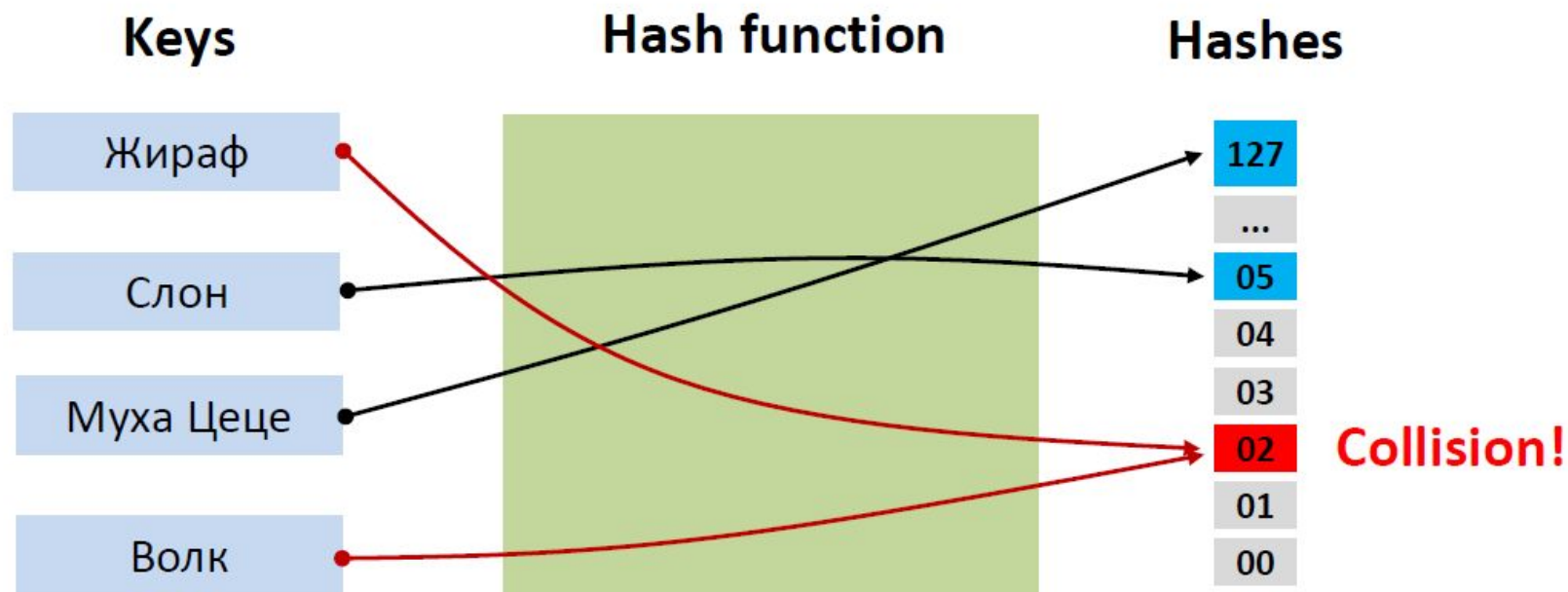
Коллизия – ситуация, когда два ключа хешированы в одну и ту же ячейку (ключи в этом случае называются синонимами).

Понятие коллизии (Collision)

- Коллизия (Collision) – это совпадение значений хеш-функции для двух разных ключей

$hash(\text{“Волк”}) = 2$

$hash(\text{“Жираф”}) = 2$



Существуют хеш-функции без коллизий – совершенные хеш-функции (perfect hash function)

Введение

Как побороть коллизию? Точнее – что делать... Первое же что приходит в голову - это взять и сохранить запись в следующую свободную ячейку. Такой способ называется **методом открытой(прямой) адресации**. Разумеется, это самый примитивный метод и конечно же в «больших программах» не всегда так поступают - сохраняют значение в следующую свободную ячейку. В общем случае, мы будем вычислять нужную ячейку по какой-нибудь формуле: мы можем двигаться в обратном направлении и делать ни один шаг, а перескакивать через одну и две ячейки. Или даже повторно применять хэш-функцию к полученному числу.

Давайте еще раз посмотрим, что у нас происходит и как мы с этим боремся. У нас есть какой-то ключ мы применяем к нему хэш-функцию. Получаем на выходе какое-то число и пытаемся использовать это число в качестве индекса. Но у нас ничего не получается поскольку в ячейке с этим же индексом уже что-то сохранен. То есть - мы получили коллизию. Нас это не пугает и мы просто применяем к полученному числу еще одно преобразование. В результате получаем еще одно число и теперь попробуем использовать его в качестве индекса. По итогу – видим, что в ячейке с таким индексом, тоже есть какое-то значение. Ничего страшного - применяем преобразования еще раз и получаем очередной индекс. Таким образом, мы применяем преобразование до тех пор пока не получим индекс пустой ячейки. Когда это произойдет, мы наконец сможем сохранить туда свое значение.

Все эти преобразования, которые мы применяем к числам, полученным из хэш функции, называется **пробиванием** и,

Введение

Это выглядит конечно же здорово, но давайте теперь разберемся как правильно искать сохранённую информацию. Мы снова возвращаемся к конфликтному ключу и применяем ему хэш-функцию. Как обычно получаем id и смотрим ячейку с этим номером. В этой ячейке, какая то ссылка, и перейдя по ней мы обнаружим информацию о первой записи. Поскольку мы искали не её, то мы проверяем нет ли у этой информации ссылки на следующее значение. Если такая ссылка имеется и перейдя по ней мы найдём то, что искали, то всё хорошо и на этом мы останавливаемся. Если нужной ссылки нет (указатель стоит на NULL), то значит мы ищем информацию которой нет в нашей хеш-таблице.

Итак, мы узнали что такое коллизии и как с ними работать. А именно, мы узнали про два метода их разрешения:

- Метод открытой адресации
- Метод цепочек

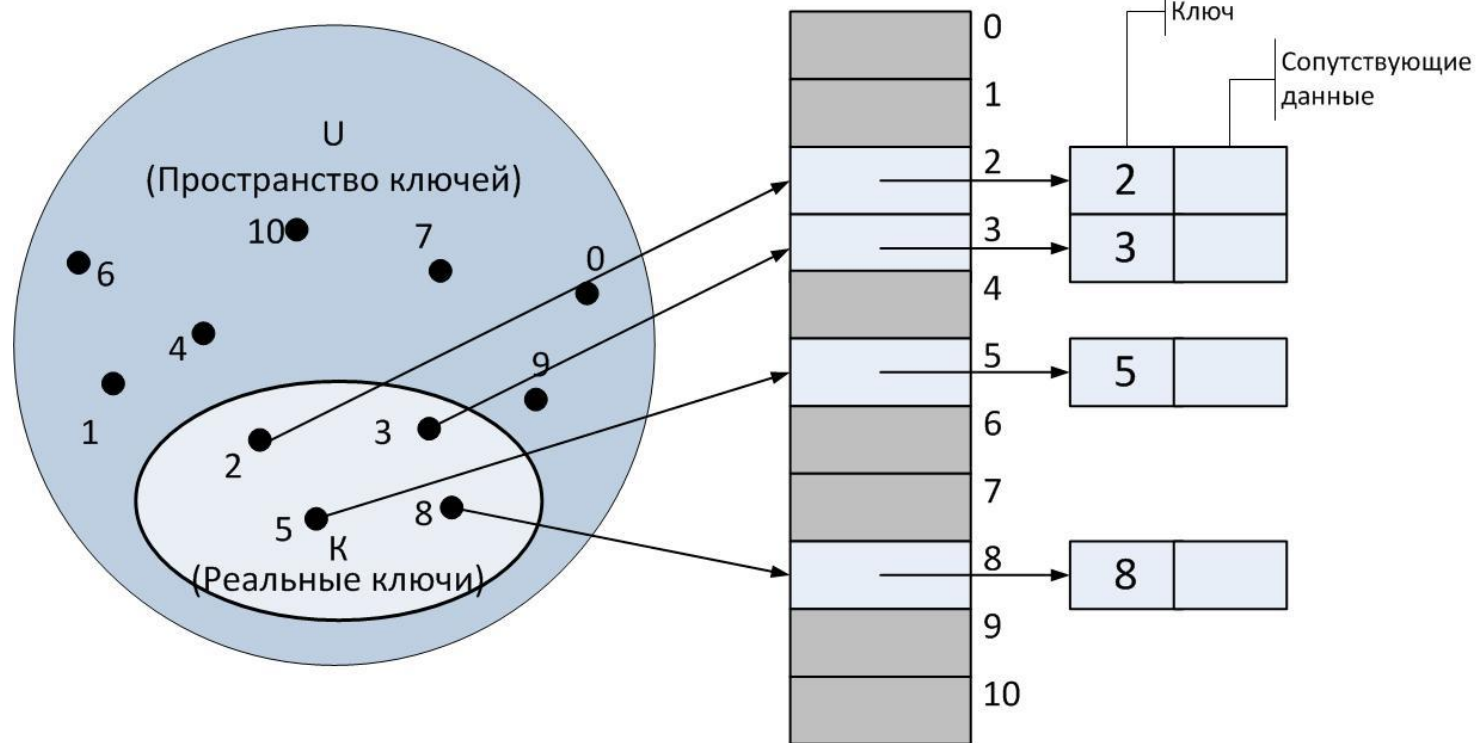
У каждого из них есть свои плюсы и минусы.

Начнем с метода открытой адресации. Какие у него есть «минусы»: во-первых, его эффективность сильно зависит от выбора способа обхода, то есть если мы выберем его неудачно, то наша хэш-таблица будет очень неэффективной. К примеру хэш-таблица может заполниться не полностью или пути обхода могут стать такими длинными что поиски добавления информации будут занимать очень много времени. К примеру, если мы выбрали способ обхода настолько ужасно, что он нас вводит просто между двумя ячейками, то очевидно, что от такой хэш-таблицы будет очень мало толку.

Прямая адресация

$U = \{0, 1, \dots, m-1\}$ – множество ключей

$T[0 \dots m-1]$ – массив (таблица с прямой адресацией)



Direct_Address_Search (T, k)

return $T[k]$

Direct_Address_Insert (T, x)

$T[\text{key}[x]] \leftarrow x$

Direct_Address_Delete (T, x)

$T[\text{key}[x]] \leftarrow \text{NIL}$

Линейное опробование

Это последовательный перебор сегментов таблицы с некоторым фиксированным шагом:

$\text{адрес} = h(x) + ci$, где i – номер попытки разрешить коллизию;

c – константа, определяющая шаг перебора.

При шаге, равном единице, происходит последовательный перебор всех сегментов после текущего.

Двойное хэширование

Основана на нелинейной адресации, достигаемой за счет суммирования значений основной и дополнительной хэш-функций:

$$\text{адрес} = h(x) + ih_2(x).$$

Очевидно, что по мере заполнения хэш-таблицы будут происходить коллизии, и в результате их разрешения очередной адрес может выйти за пределы адресного пространства таблицы.

Чтобы это явление происходило реже, можно пойти на увеличение длины таблицы по сравнению с диапазоном адресов, выдаваемым хэш-функцией. С одной стороны, это приведет к сокращению числа коллизий и ускорению работы с хэш-таблицей, а с другой – к нерациональному расходованию памяти.

Даже при увеличении длины таблицы в два раза по сравнению с областью значений хэш-функции нет гарантии того, что в результате коллизий адрес не превысит длину таблицы. При этом в начальной части таблицы может оставаться достаточно свободных сегментов.

Поэтому на практике используют циклический переход к началу таблицы.

Квадратичное опробование

отличается от линейного тем, что шаг перебора сегментов нелинейно зависит от номера попытки найти свободный сегмент:

$$\text{адрес} = h(x) + a \cdot i + b \cdot i^2$$

i – номер попытки,

a и b – константы.

Введение

Кроме того, данный вид разрешения коллизий заставляет нас очень внимательно выбирать размер самой хеш-таблицы. Поскольку все данные будут лежать внутри неё и нет каких-то отдельных областей памяти куда мы можем записать значение которое в неё не влезло. То есть, если выбранный размер хеш-таблицы оказался слишком маленьким то в него банально не влезет вся наша информация. А слишком большой размер будет приводить к перерасходу памяти.

Из плюсов метода открытой адресации можно выделить, например, быстрый обход. Все данные у нас лежат в одном массиве, то есть находится в памяти где-то рядом, соответственно путешествуем между ними очень быстро. Кроме того, мы расходуем меньше памяти поскольку нам не нужно хранить указатели на следующие значения, как в методе цепочек.

Соответственно все перечисленные плюсы метода открытой адресации сразу же становятся минусами метода цепочек. Здесь мы расходуем дополнительную память на то, чтобы хранить ссылки. А сами ссылки ведут на какие-то разрозненные участки памяти и переходить по ним не быстро. Несомненный плюс данного метода заключается в том, что его очень просто реализовывать и количество нюансов, которые нужно учесть из-за которых все может пойти не так, гораздо меньше.

Если в хэш-таблице допускается удаление элементов, то при достижении пустого сегмента, не найдя элемента **x**, нельзя быть уверенным в том, что его вообще нет в таблице, т.к. сегмент может стать пустым уже после вставки элемента **x**. Поэтому, чтобы увеличить эффективность данной реализации, необходимо в сегмент, который освободился после операции удаления элемента, поместить специальную константу, которую назовем, например, **DEL**. В качестве альтернативы специальной константе можно использовать дополнительное поле таблицы, которое показывает состояние элемента.

Важно различать константы **DEL** и **NULL** – последняя находится в сегментах, которые никогда не содержали элементов. При таком подходе выполнение поиска элемента не требует просмотра всей хэш-таблицы. Кроме того, при вставке элементов сегменты, помеченные константой **DEL**, можно трактовать как свободные, таким образом, пространство, освобожденное после удаления элементов, можно рано или поздно использовать повторно.

Но, если невозможно непосредственно сразу после удаления элементов пометить освободившиеся сегменты, то следует предпочесть закрытому хэшированию схему открытого хэширования.

Существует несколько методов повторного хэширования, то есть определения местоположений $h(x)$, $h_1(x)$, $h_2(x)$, ... :

- линейное опробование;
- квадратичное опробование;
- двойное хэширование.

Введение

Теперь давайте посмотрим какие виды пробивания существуют. Конкретно то преобразование, которые мы использовали, когда сдвигались на одну ячейку, называется **линейным пробированием**. Разумеется это далеко не единственное решение. Существует также квадратичное пробирование, двойное хеширование и другие способы.

Кроме метода **открытой(прямой) адресации** существует еще один популярный способ разрешения коллизий, которая называется - **метод цепочек**. Суть этого метода заключается в том, что, когда мы встречаем коллизию, мы добавляем в конфликтную ячейку **ссылку** на следующие значения. То есть мы сохраняем это значение не в нашем массиве, а в какой-то отдельной области памяти.

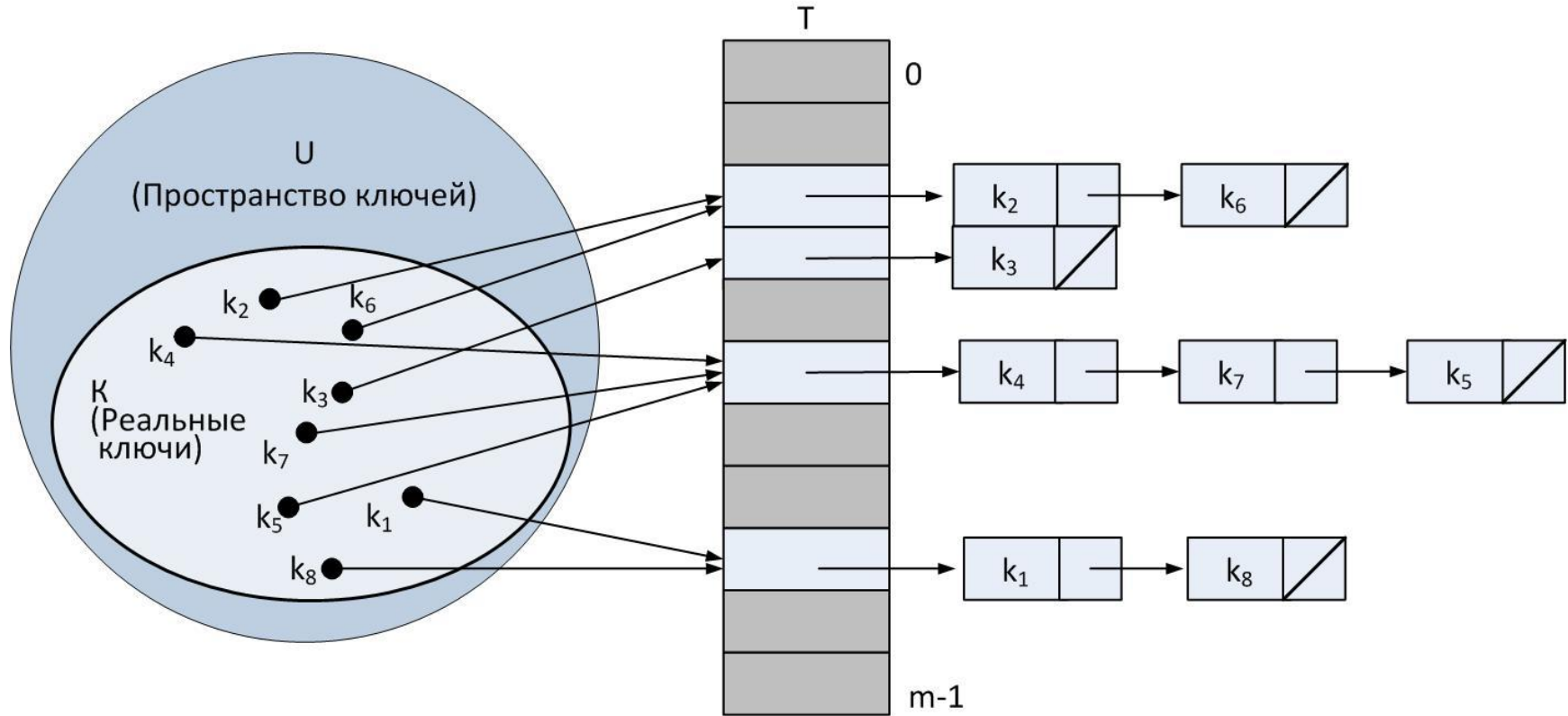
Как это работает. У нас есть имя человека, которое служит **ключом**, и его номер телефона. К этому имени применяем хэш-функцию и получаем на выходе число. Далее, ищем ячейку с соответствующим индексом, но на этот раз мы сохраняем в ячейку не саму информацию о человеке, а только ссылку на область памяти, где лежит эта информация. То есть, у нас где-то в отдельной области памяти, есть имя человека и его номер телефона и в нашей таблице мы храним ссылку на эту область. И так двигаемся далее, добавляем в хэш-таблицу следующее новое значение. Получаем id новой ячейки и сохраняем туда ссылку на информацию о следующем человеке. У нас все идет хорошо, до тех пор пока нам не встретится коллизия - хеши двух разных ключей одинаковы. Тогда, мы не будем искать для решения отдельную ячейку, а ссылку на наши данные будет хранить сама область памяти, в которой размещена информация, которая успела занять ячейку хеш-таблицы первой.

МЕТОД ЦЕПОЧЕК

Технология сцепления элементов состоит в том, что элементы множества, которым соответствует одно и то же хеш-значение, связываются в цепочку-список:

- в позиции номер i хранится указатель на голову списка тех элементов, у которых хэш-значение ключа равно i ;
- если таких элементов в множестве нет, в позиции i записан NULL.

Разрешение коллизии при помощи цепочек (открытое хеширование)



`Chained_Hash_Insert(T, x)`

Вставить x в заголовок списка $T [h (key[x])]$

`Chained_Hash_Search (T, k)`

Поиск элемента с ключом k в списке $T [h (k)]$

`Chained_Hash_Delete(T, x)`

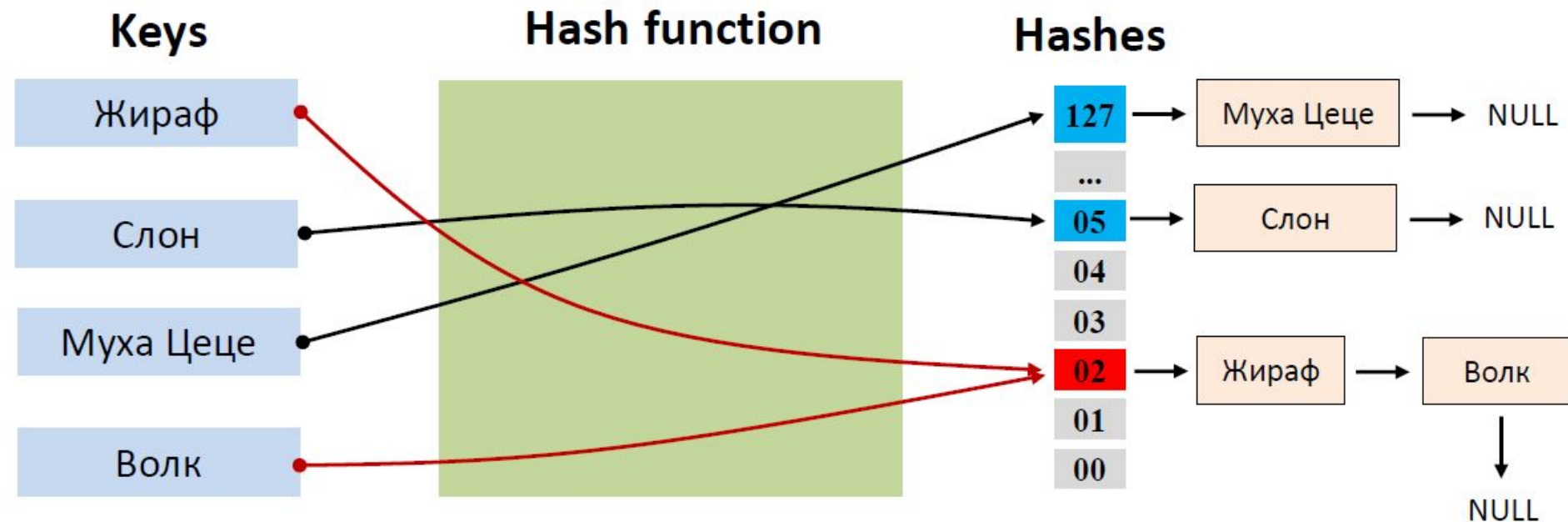
Удаление x из списка $T [h (key[x])]$

Разрешение коллизий (Collision resolution)

Метод цепочек (Chaining) – закрытая адресация

Элементы с одинаковым значением хеш-функции объединяются в связный список. Указатель на список хранится в соответствующей ячейке хеш-таблицы.

- При коллизии элемент добавляется в начало списка.
- Поиск и удаление элемента требуют просмотра всего списка.



Введение

Конечно же, у метода открытой адресации с использованием пробирования есть свои недостатки.

Во-первых, что будет если мы хотим сохранить новое значение, а таблица уже заполнена? Как вариант мы можем создать новую таблицу, скопировать туда все значения и пользоваться уже ей.

Во-вторых, что произойдет если мы хотим удалить значение из хеш-таблицы? Для этого мы можем пометить соответствующие ячейки таблицы как удаленные, затем во время поиска мы будем их пропускать, а при добавлении новых значений мы будем считать их пустыми и, соответственно, записывать туда информацию.

Что произойдет если мы очень часто удаляли что-то из таблицы и в результате накопили очень много ячеек которые помечены как удаленные?! Поиск по такой таблице будет работать значительно медленнее, потому что нам приходится пропускать очень много ячеек. Для того чтобы дальше без проблем работать с этой таблицей применим некую «магию», например мы соберем все оставшиеся не удаленные значения и перенесём их в новую таблицу. Таким образом мы избавимся от всех мусорных ячеек.

И так мы имели две проблемы и похожим образом от них избавились. В общем случае данный процесс называется рехешированием

Требования к хеш-функциям

Теперь поговорим, что же такое хорошая хэш-функция. Для того, чтобы быть хорошей она должна обладать четырьмя свойствами:

Детерминизм. Суть его заключается в том, что для одного и того же значения ключа хэш-функция **всегда должна выдавать одно и то же значение**, то есть не должно происходить такой ситуации, что сегодня *key1* имеет индекс 0 а завтра 8.

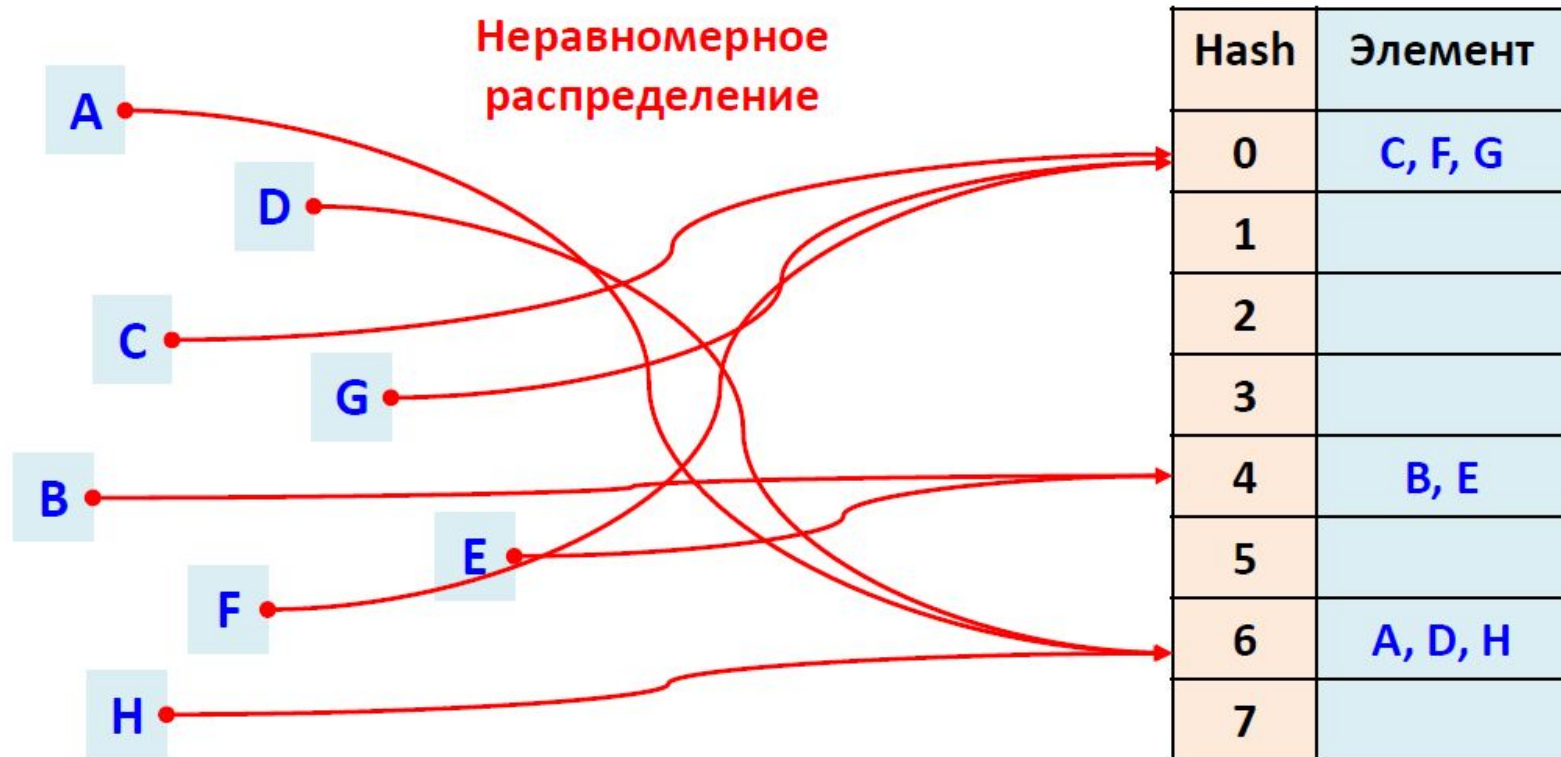
Равномерность. Оно заключается в том, чтобы данные по таблице распределялись равномерно. То есть, **не должно** получаться такой ситуации, что у, например, *key1*, *key12*, *key38* внезапно получился один и тот же индекс. Чем реже такое встречается тем меньше, то есть чем меньше коллизии у нас будет возникать – тем лучше. Конечно же не может быть такой хэш-функции которая бы работала вообще без коллизии, как минимум потому что возможных вариаций ключей у нас может быть бесконечное количество, а вот размер массива (самой хеш-таблицы) конечно же ограничен.

Эффективность. Оно очень простое и заключается в том, что хэш-функция должна вычисляться быстро. Здесь все очевидно - чем быстрее мы вычисляем индекс по ключу, тем быстрее мы получаем и сохраняем информацию.

Ограниченность. Об этом свойстве мы уже говорили ранее и, заключается оно в том что индексы, которые выдает хэш-функция должны быть в пределах таблицы, если это свойство не выполняется то мы просто не сможем сохранять некоторые значения в хеш-таблицу потому, что у

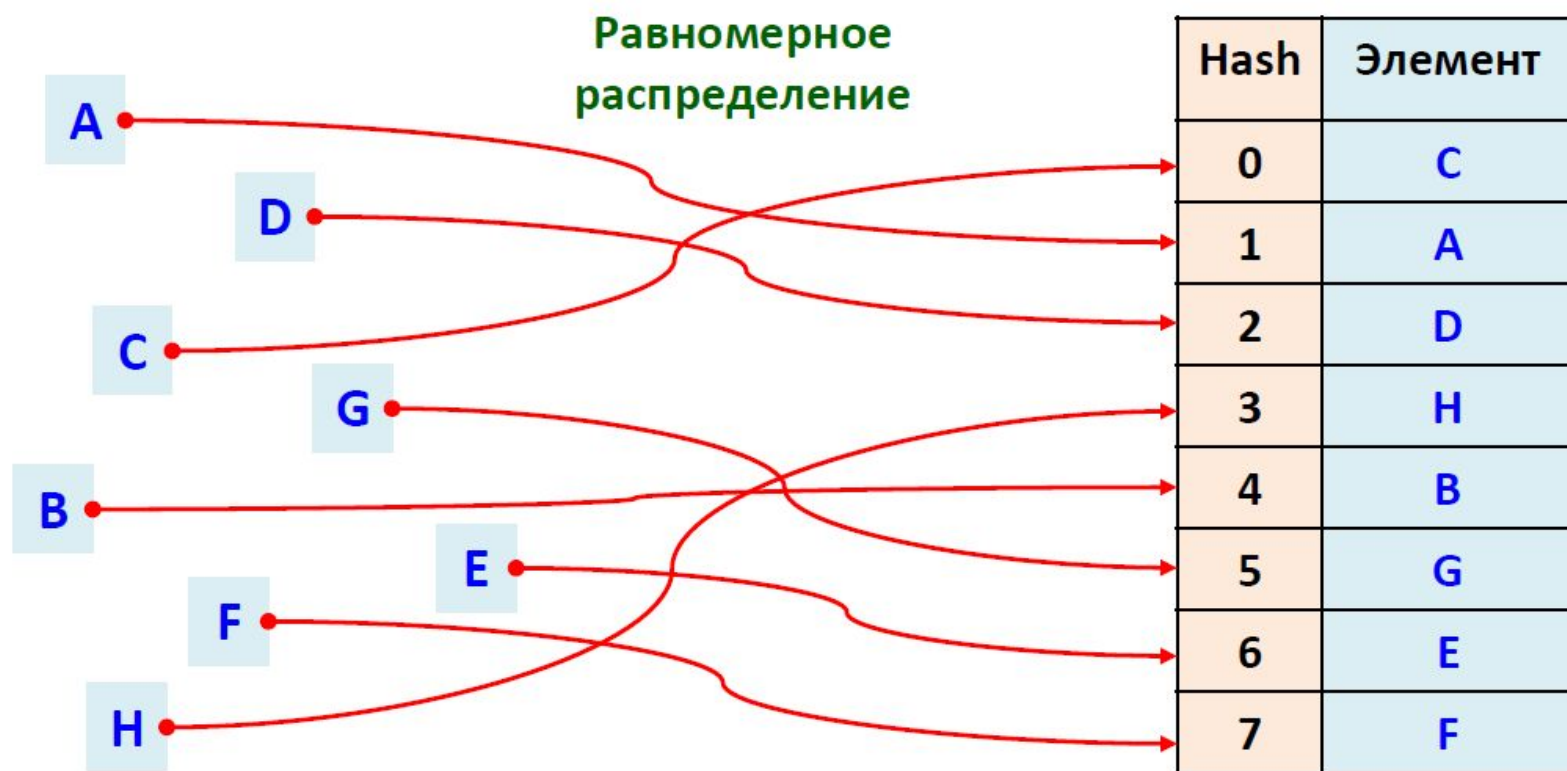
Требования к хеш-функциям

- **Равномерность (uniform distribution)** – хеш-функция должна равномерно заполнять индексы массива возвращаемыми номерами
- Желательно, чтобы все хэш-коды формировались с одинаковой равномерной распределенной вероятностью



Требования к хеш-функциям

- **Равномерность (uniform distribution)** – хеш-функция должна равномерно заполнять индексы массива возвращаемыми номерами
- Желательно, чтобы все хэш-коды формировались с одинаковой равномерной распределенной вероятностью



Требования к хеш-функциям (дополнительные)

С точки зрения практического применения, хорошей является такая хеш-функция, которая удовлетворяет ещё и следующим дополнительным условиям:

- Она должна быть **необратимой**;
- Она **не должна** отображать какую-либо связь между значениями ключей в связь между значениями адресов;
- В алгоритмах криптографии существенно требование линейности алгоритма хеширования – **он не должен распараллеливаться**.

При предположении, что каждый элемент может попасть в любую позицию таблицы с равной вероятностью и независимо от того, куда попал любой другой элемент, **среднее время работы операции поиска элемента составляет $O(1 + k)$** , где k – коэффициент заполнения таблицы.

$$k = n / m,$$

n – количество элементов таблицы,

m – размер таблицы.

Пятиминутка

Пусть для хеширования целых чисел в 5-сегментную хеш-таблицу используется хеш-функция $h(i) = i \bmod 5$.
Приведите результирующую хеш-таблицу после вставки в нее следующих чисел в указанном порядке:

476 526 691 873 212

Открытое хеширование (метод цепочек)

0	
1	476 526 691
2	212
3	873
4	

Закрытое хеширование с линейной методикой разрешения коллизий

0	212
1	476
2	526
3	691
4	873

Вопросы экзаменационных билетов

1. Структуры данных

1.1. Пусть функция `put` добавляет число в очередь, а функция `get` извлекает число из очереди и печатает его.

Что будет напечатано при выполнении последовательности вызовов функций:

```
put(3); get(); put(7); get(); put(11); put(12); put(19); get(); get(); get();
```

Ответ: **3 7 11 12 19**

1.2. Перевести данное арифметическое выражение в обратную польскую запись:

$(8 / ((3 - 1) / (8 + (5 + 6))))$

Ответ: **8 3 1 - 8 5 6 + + / /**

1.3. Является ли последовательность 27 26 17 14 16 11 12 2 8 0 10 *невозрастающей* кучей?

Да ☒

Нет ☐

1.4. Построить неубывающую кучу для данной последовательности чисел, начиная с пустой кучи и добавляя элементы в указанном порядке:

7 4 1 9 23 13 24 21 5 19 29

1 5 4 7 19 13 24 21 9 23 29
см. след. слайд

1.5. Пусть для хеширования целых чисел 6-сегментную хеш-таблицу используется хеш-функция

$h(i) = i \bmod 6$.

Приведите результирующую хеш-таблицу после вставки в неё следующих чисел в указанном порядке:

146 492 77 291 484 176

Использовать закрытое хеширование с линейной методикой разрешения коллизий с шагом = 1

492 176 146 291 484 77

id	0	1	2	3	4	5
Значение	492	176	146	291	484	77

$146 \bmod 6 = 2$

$492 \bmod 6 = 0$

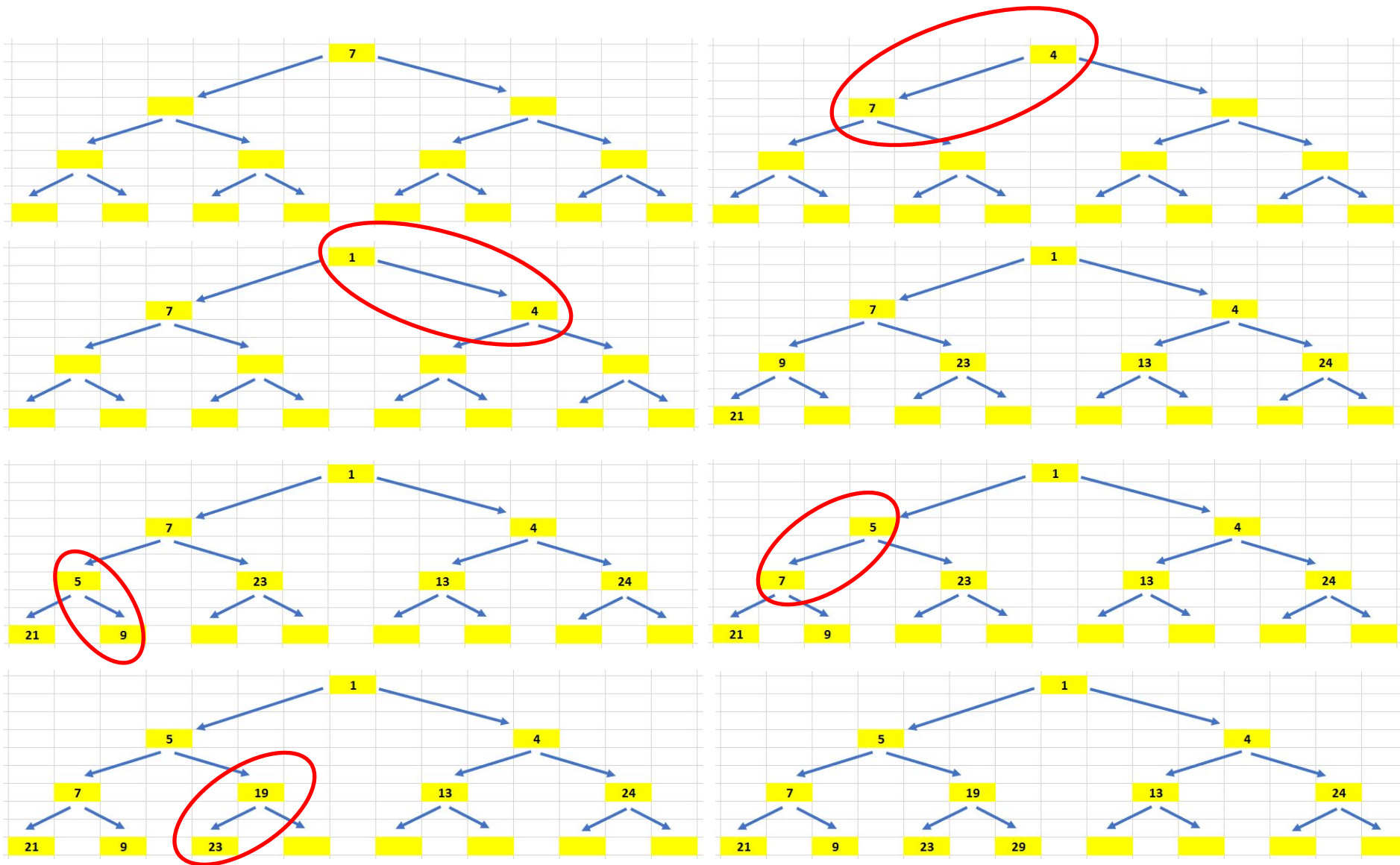
$77 \bmod 6 = 5$

$291 \bmod 6 = 3$

$484 \bmod 6 = 4$

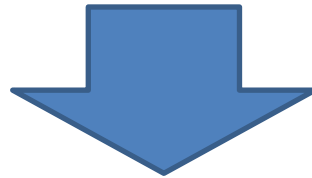
$176 \bmod 6 = 2$

1.4 Входная последовательность: 7 4 1 9 23 13 24 21 5 19 29



Ответ: 1 5 4 7 19 13 24 21 9 23 29

Дополнительная информация.



Хеш-функции (Hash function)

- **Хеш-функция (Hash function)** – это функция преобразующая значения ключа (например: строки, числа, файла) в целое число
- Значение возвращаемое хеш-функцией называется хеш-кодом (hash code), контрольной суммой (hash sum) или хешем (hash)

```
#define HASH_MUL 31  
#define HASH_SIZE 128
```

i	v	a	n	o	v
105	118	97	110	111	118

```
int main()  
{  
    unsigned int h = hash("ivanov");  
}
```

Хеш-функции (Hash function)

```
#define HASH_MUL 31
#define HASH_SIZE 128

unsigned int hash(char *s) {
    unsigned int h = 0;
    char *p;

    for (p = s; *p != '\0'; p++)
        h = h * HASH_MUL + (unsigned int)*p;
    return h % HASH_SIZE;
}
```

```
h = 0 * HASH_MUL + 105
h = 105 * HASH_MUL + 118
h = 3373 * HASH_MUL + 97
h = 104660 * HASH_MUL + 110
h = 3244570 * HASH_MUL + 111
h = 100581781 * HASH_MUL + 118
return 3118035329 % HASH_SIZE
```

i	v	a	n	o	v
105	118	97	110	111	118

```
// hash("ivanov") = 1
```

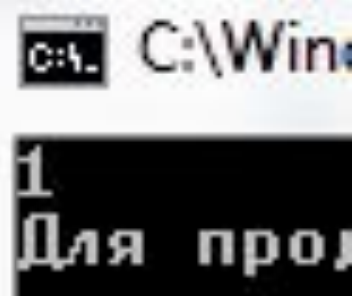
```
#include<iostream>

#define HASH_MUL 31
#define HASH_SIZE 128

using namespace std;

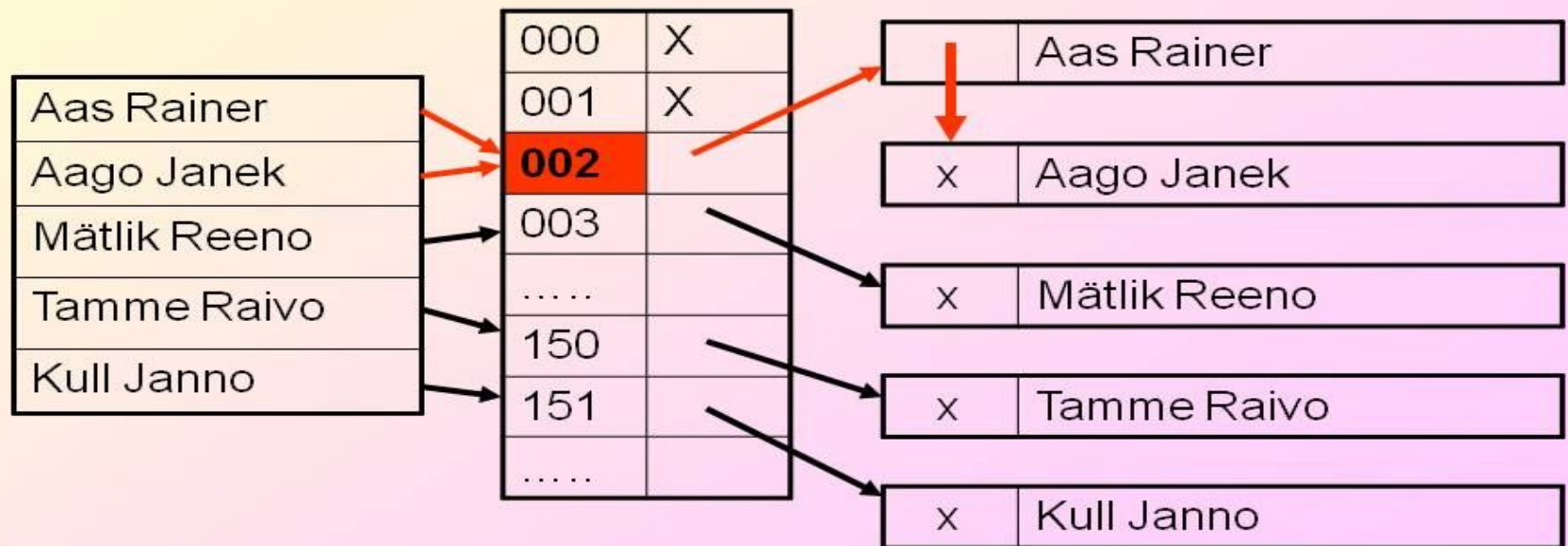
unsigned int hash(char *s) {
    unsigned int h = 0;
    char*p;
    for(p = s; *p != '\0'; p++)
        h = h * HASH_MUL + (unsigned int)*p;
    return h % HASH_SIZE;
}

int main() {
    unsigned int h = hash("ivanov");
    cout << h;
    return 0;
}
```

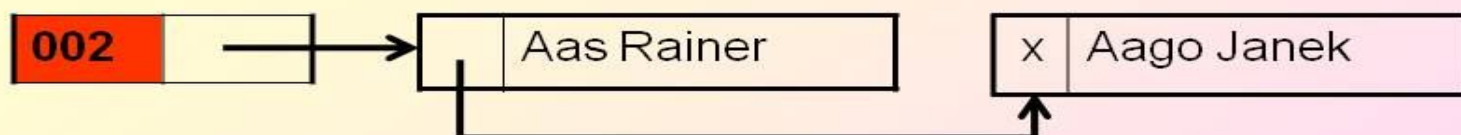


Пример реализации метода цепочек при разрешении коллизий:
→ на ключ 002 претендуют два значения, которые организуются в линейный список.

Каждая ячейка массива является указателем на связный список (цепочку) пар ключ-значение, соответствующих одному и тому же хэш-значению ключа. Коллизии просто приводят к тому, что появляются цепочки длиной более одного элемента.



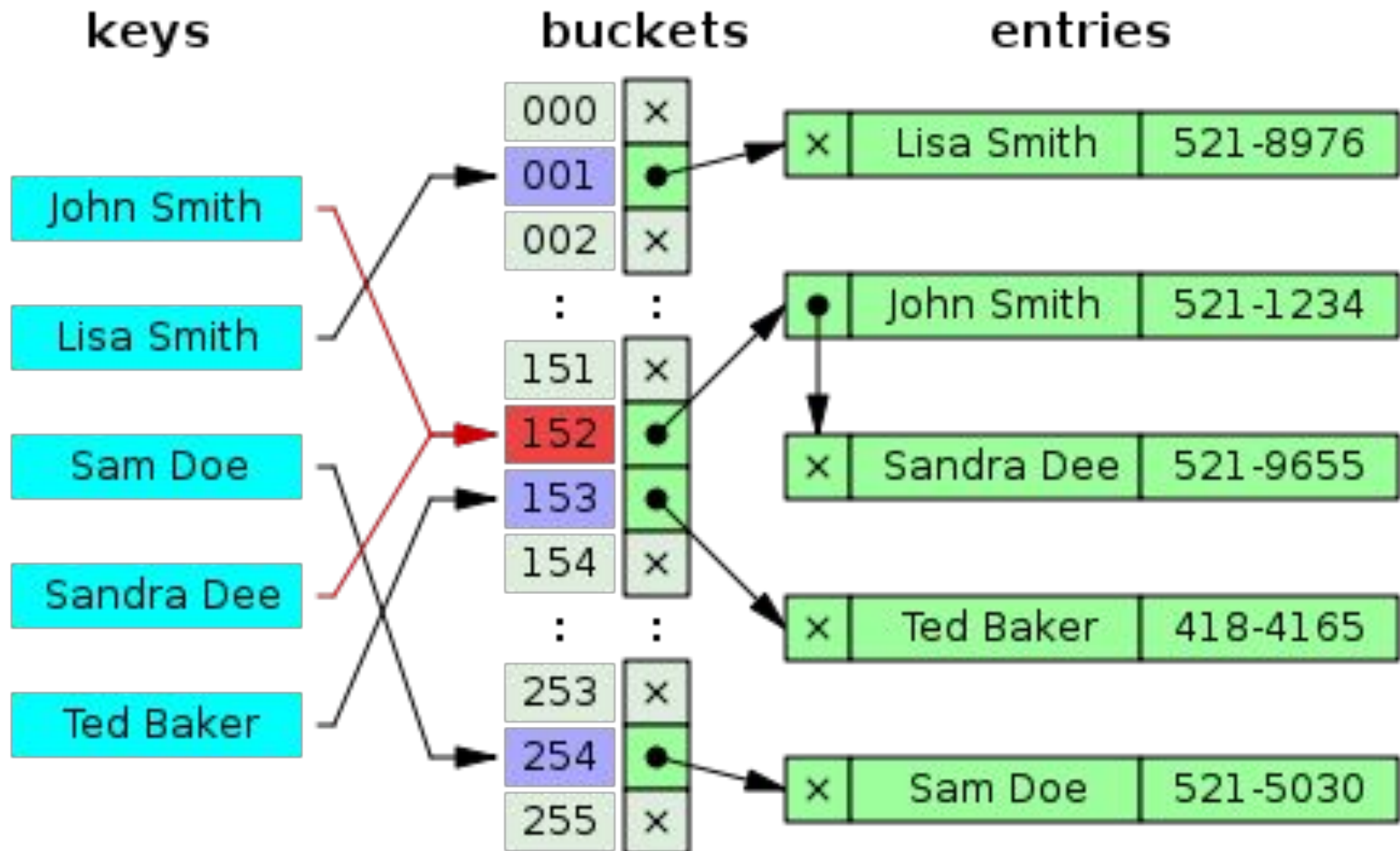
Операции **поиска** или **удаления** данных требуют просмотра всех элементов соответствующей ему цепочки, чтобы найти в ней элемент с заданным ключом.



Для **добавления** данных нужно добавить элемент в конец или начало соответствующего списка, и, в случае если коэффициент заполнения станет слишком велик, увеличить размер массива и перестроить таблицу.



Открытое хеширование (пример)



Разрешение коллизий (Collision resolution)

Открытая адресация (Open addressing)

В каждой ячейке хеш-таблицы хранится не указатель на связный список, а один элемент (ключ, значение)

Если ячейка с индексом $\text{hash}(\text{key})$ занята, то осуществляется поиск свободной ячейки в следующих позициях таблицы

Линейное хеширование (linear probing) – проверяются позиции:

$\text{hash}(\text{key}) + 1, \text{hash}(\text{key}) + 2, \dots, (\text{hash}(\text{key}) + i) \bmod h, \dots$

Если свободных ячеек нет, то таблица заполнена

Пример:

- $\text{hash}(D) = 3$, но ячейка с индексом 3 занята.
- Присматриваем ячейки: 4 – занята, 5 – свободна.

Hash	Элемент
0	B
1	
2	
3	A
4	C
5	D
6	
7	

Закрытое хеширование

При закрытом (внутреннем) хэшировании в хэш-таблице хранятся непосредственно сами элементы, а не заголовки списков элементов. Поэтому в каждой записи (сегменте) может храниться только один элемент.

При закрытом хэшировании применяется методика **повторного хэширования**:

- Если осуществляется попытка поместить элемент x в сегмент с номером $h(x)$, который уже занят другим элементом (коллизия), то в соответствии с методикой повторного хэширования выбирается последовательность других номеров сегментов $h_1(x), h_2(x), \dots$, куда можно поместить элемент x .
- Каждое из этих местоположений последовательно проверяется, пока не будет найдено свободное. Если свободных сегментов нет, то, следовательно, таблица заполнена, и элемент x добавить нельзя.

При поиске элемента x необходимо просмотреть все местоположения $h(x), h_1(x), h_2(x), \dots$, пока не будет найден x или пока не встретится пустой сегмент. Чтобы объяснить, почему можно остановить поиск при достижении пустого сегмента, предположим, что в хэш-таблице не допускается удаление элементов. Пусть $h_3(x)$ – первый пустой сегмент. В такой ситуации невозможно нахождение элемента x в сегментах $h_4(x), h_5(x)$ и далее, так как при вставке элемент x вставляется в первый пустой сегмент, следовательно, он находится где-то до сегмента $h_3(x)$.

