Алгоритмы и структуры данных

Лекция 4

Хеширование

Хеширование

Хеширование (хэширование) – 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.

Такие преобразования также называют хеш-функциями, а их результаты называют хеш, хеш-код или хеш-таблицей.

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.

Хеш-таблицы

Хеш-таблица – это структура данных, реализующая интерфейс ассоциативного массива.

Она позволяет хранить пары вида “ключ - значение” и выполнять операции:

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

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

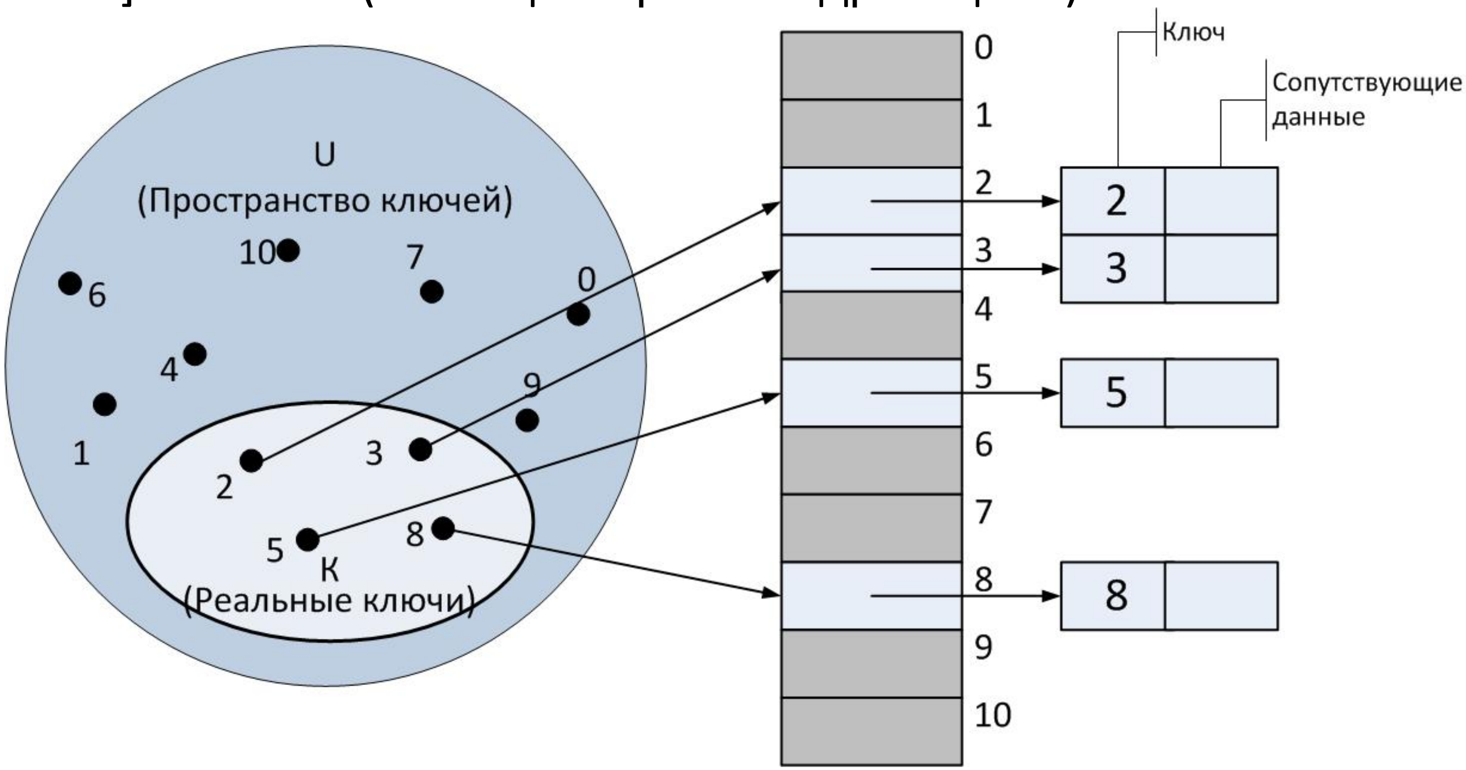
Области применения хеширования

- 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

Прямая адресация

U = {0, 1, …, m-1} – множество ключей

T [0 .. m-1] – массив (таблица с прямой адресацией)



Direct_Address_Search (T, k)

    return T[k]

Direct_Address_Insert (T, x)

    T[key[x]] ← x

Direct_Address_Delete (T, x)

    T[key[x]] ← NIL

Хеш-таблицы

Недостатки прямой адресации:

- Пространство ключей U велико, хранение таблицы размера |U| непрактично
- $|K| << |U| \Rightarrow$ выделенная память расходуется напрасно
- С другой стороны, размер ключей может быть больше размерности таблицы

Требования к памяти могут быть снижены до $\theta(|K|)$.

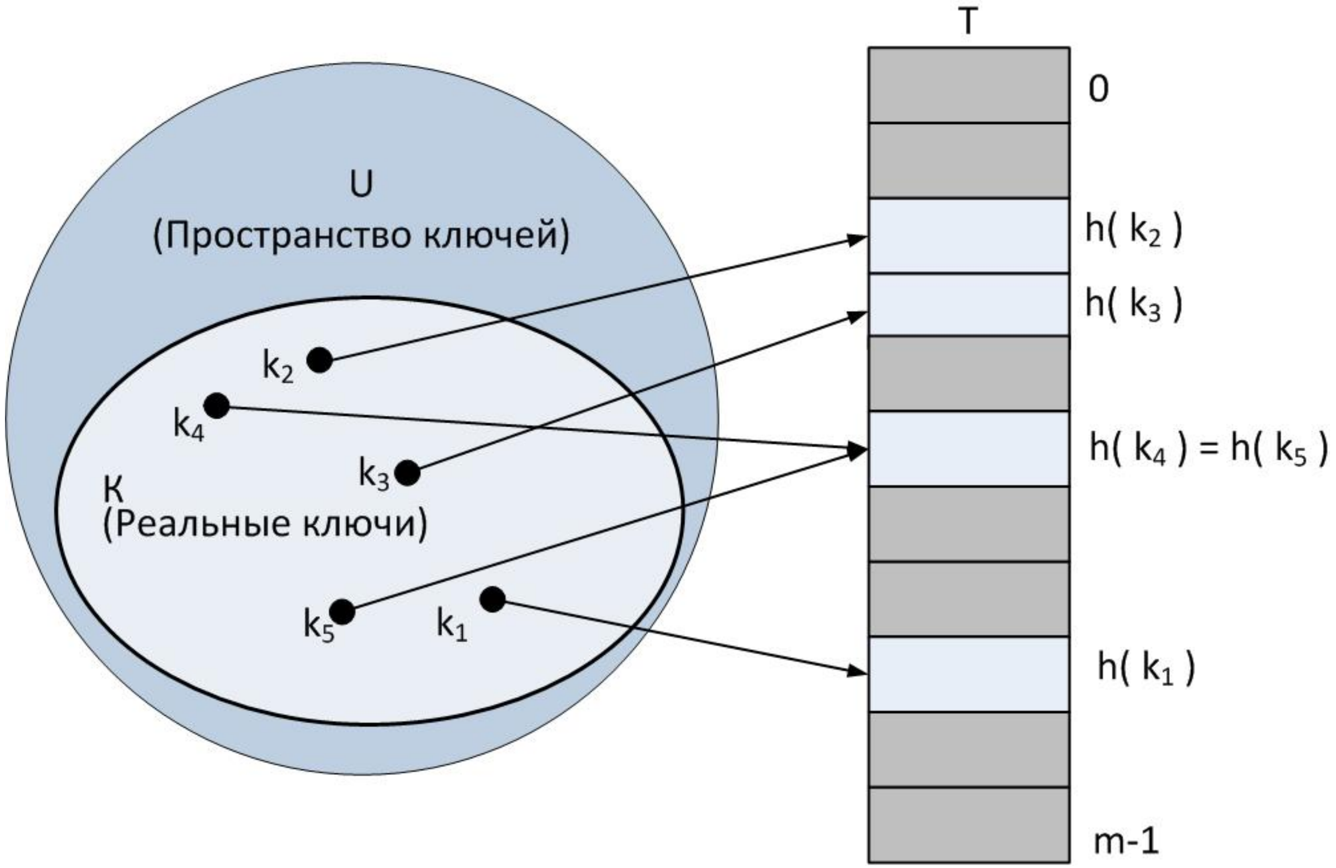
Хеш-функция:

$$h : U \to \{ 0, 1, \ldots, m-1\}$$

m – размер хеш-таблицы

Цель хеш-функции – уменьшить рабочий диапазон индексов массива и вместо |U| значений обойтись m значениями.

Хеш-таблицы и коллизии



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

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.

МЕТОДЫ РАЗРЕШЕНИЯ КОЛЛИЗИЙ

Коллизии осложняют использование хеш-таблиц, так как нарушают однозначность соответствия между хеш-кодами и данными.

Тем не менее существуют способы преодоления возникающих сложностей:

- ✔ метод цепочек – внешнее или открытое хеширование;
- ✔ метод открытой адресации – закрытое хеширование.

Открытое (внешнее) хеширование

- потенциальное множество (возможно, бесконечное) разбивается на конечное число классов;
- для m классов, пронумерованных от 0 до m-1, строится хеш-функция $h(x) : x \to \{0, \ldots, m-1\}$,

  где x – произвольный элемент исходного множества.

Часто классы называют **сегментами**.

Говорят, что **x принадлежит сегменту h(x)**.

Массив, называемый таблицей сегментов, содержит заголовки для m списков.

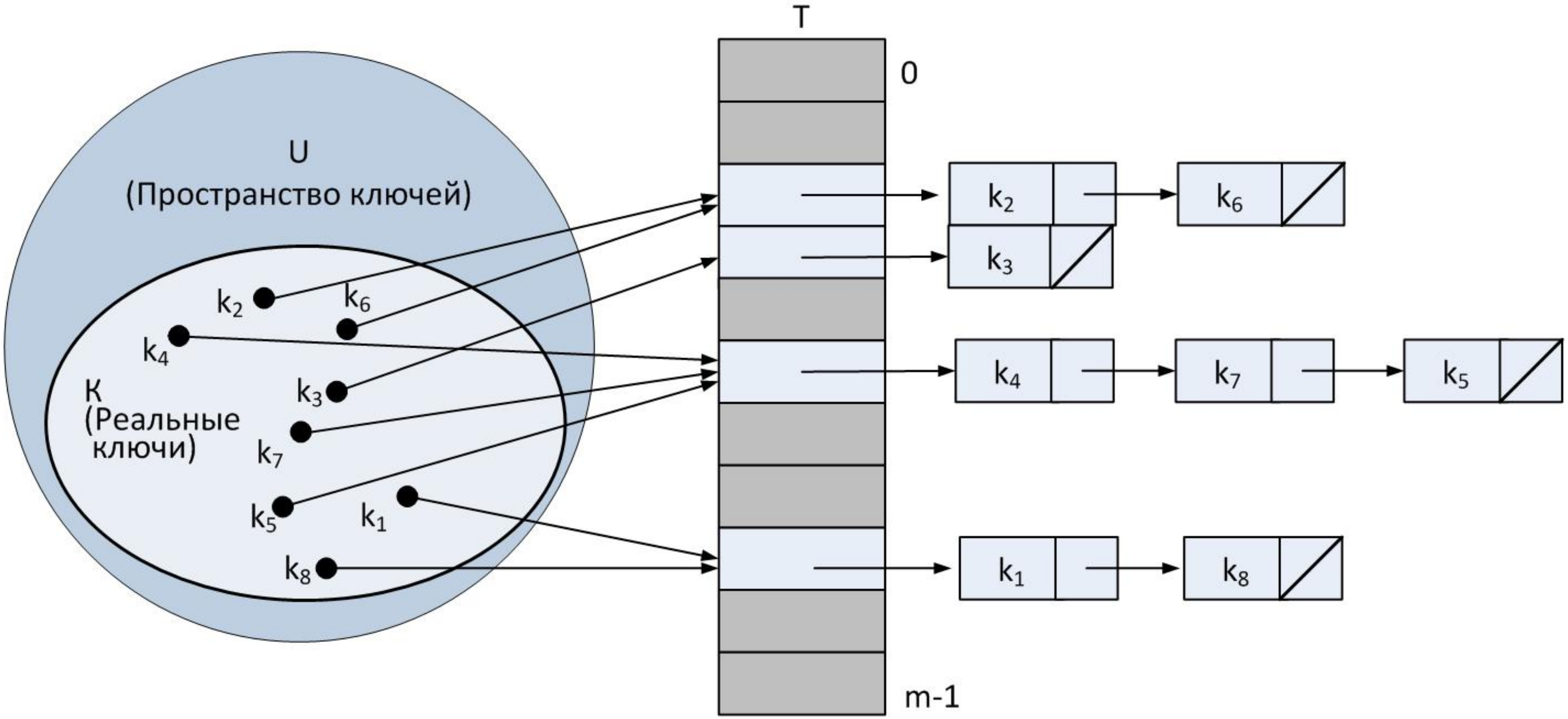
Если сегменты одинаковы по размеру, то средняя длина списков будет n/m.

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])]

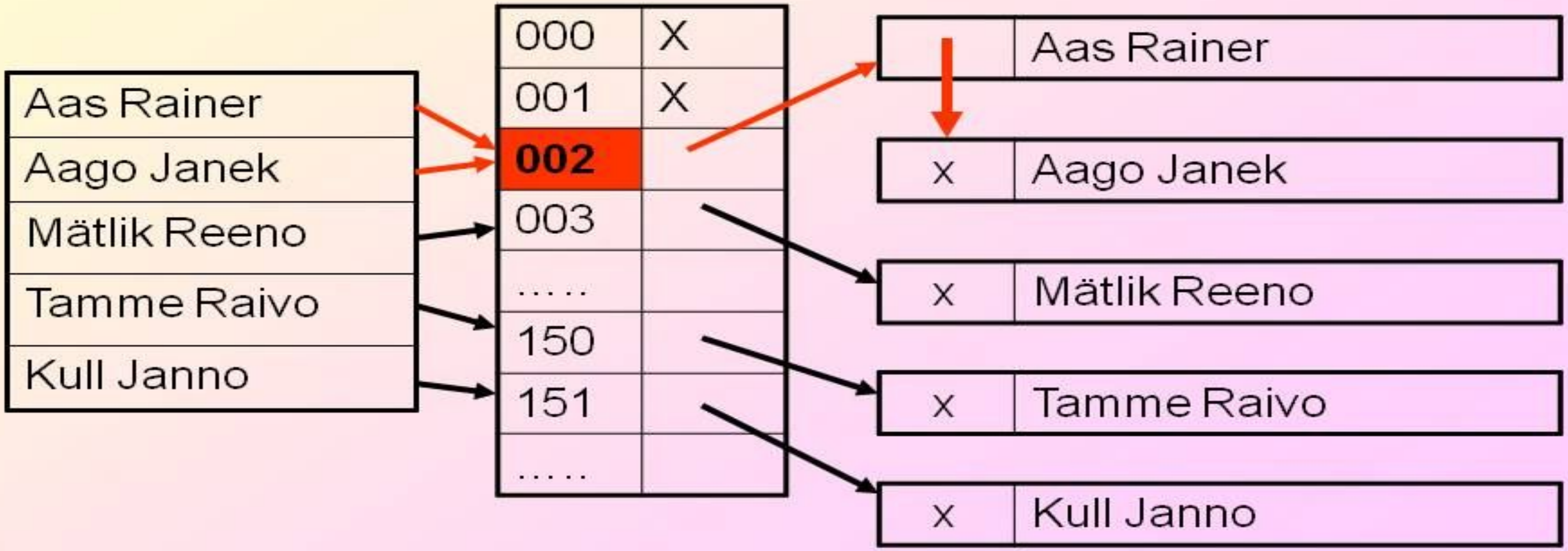
Chained_Hash_Search (T, k)

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

Chained_Hash_Delete(T, x)

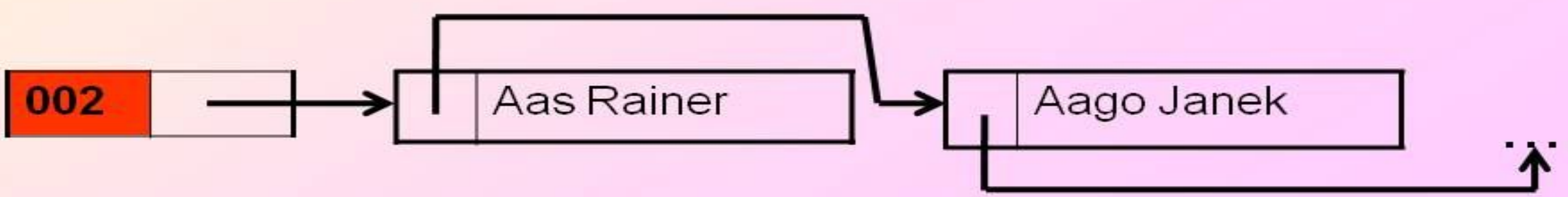
Удаление x из списка T [h (key[x])]

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.



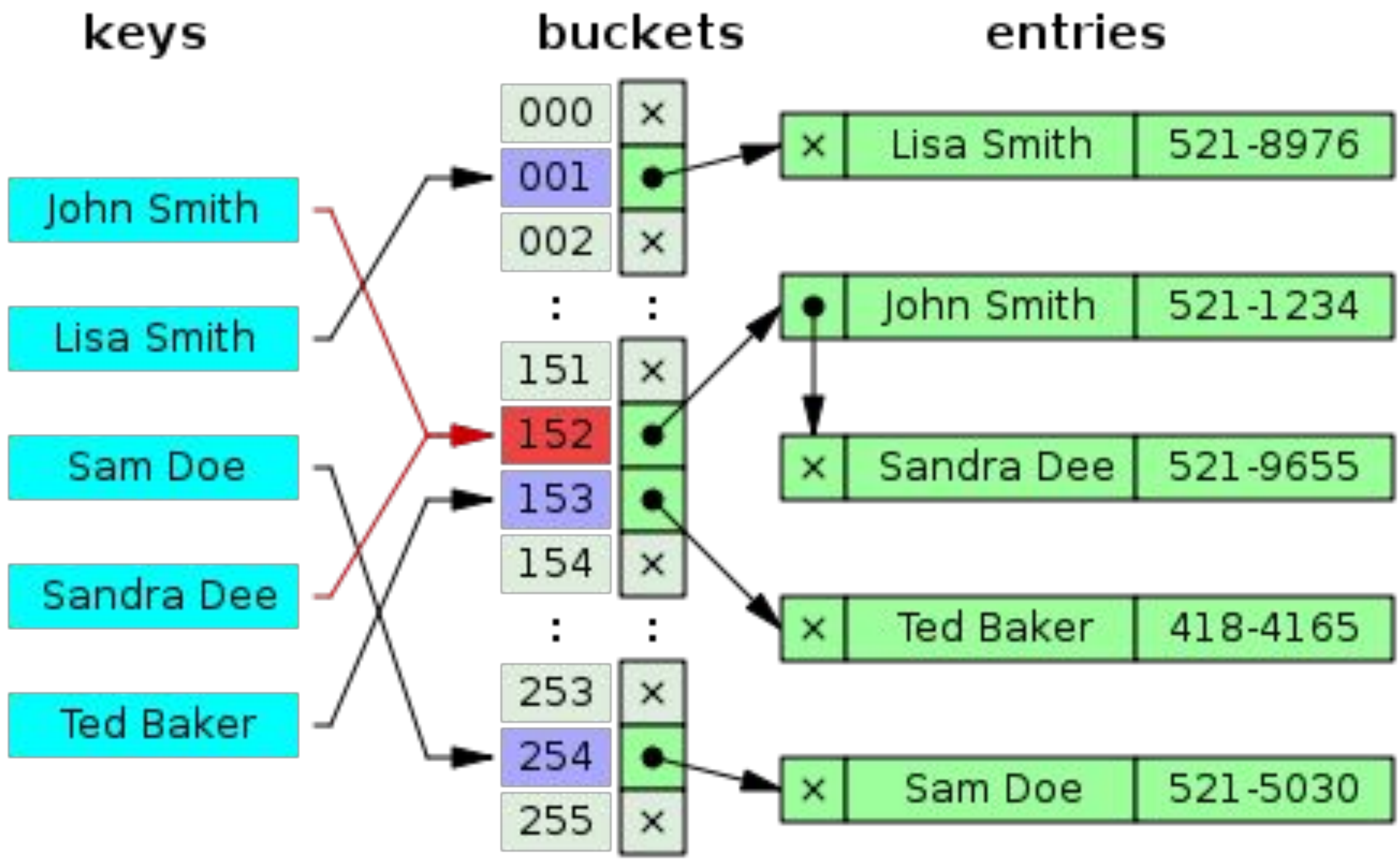
При предположении, что каждый элемент может попасть в любую позицию таблицы с равной вероятностью и независимо от того, куда попал любой другой элемент, ***среднее время работы операции поиска элемента составляет O(1 + k)***, где k – коэффициент заполнения таблицы.

$k = n / m$,

n – количество элементов таблицы,

m – размер таблицы.

Открытое хеширование (пример)



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 h1(x),h2(x),..., 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),h1(x),h2(x),..., 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 h3(x) – первый пустой сегмент. В такой ситуации невозможно нахождение элемента x в сегментах h4(x),h5(x) и далее, так как при вставке элемент x вставляется в первый пустой сегмент, следовательно, он находится где-то до сегмента h3(x).

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 хеширования.

Существует несколько методов повторного хэширования, то есть определения местоположений h(x),h1(x),h2(x),...:

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

Линейное опробование

Это последовательный перебор сегментов таблицы с некоторым фиксированным шагом:

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.

Квадратичное опробование

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

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

i – номер попытки,

a и b – константы.

Двойное хэширование

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

адрес=h(x)+ih2(x).

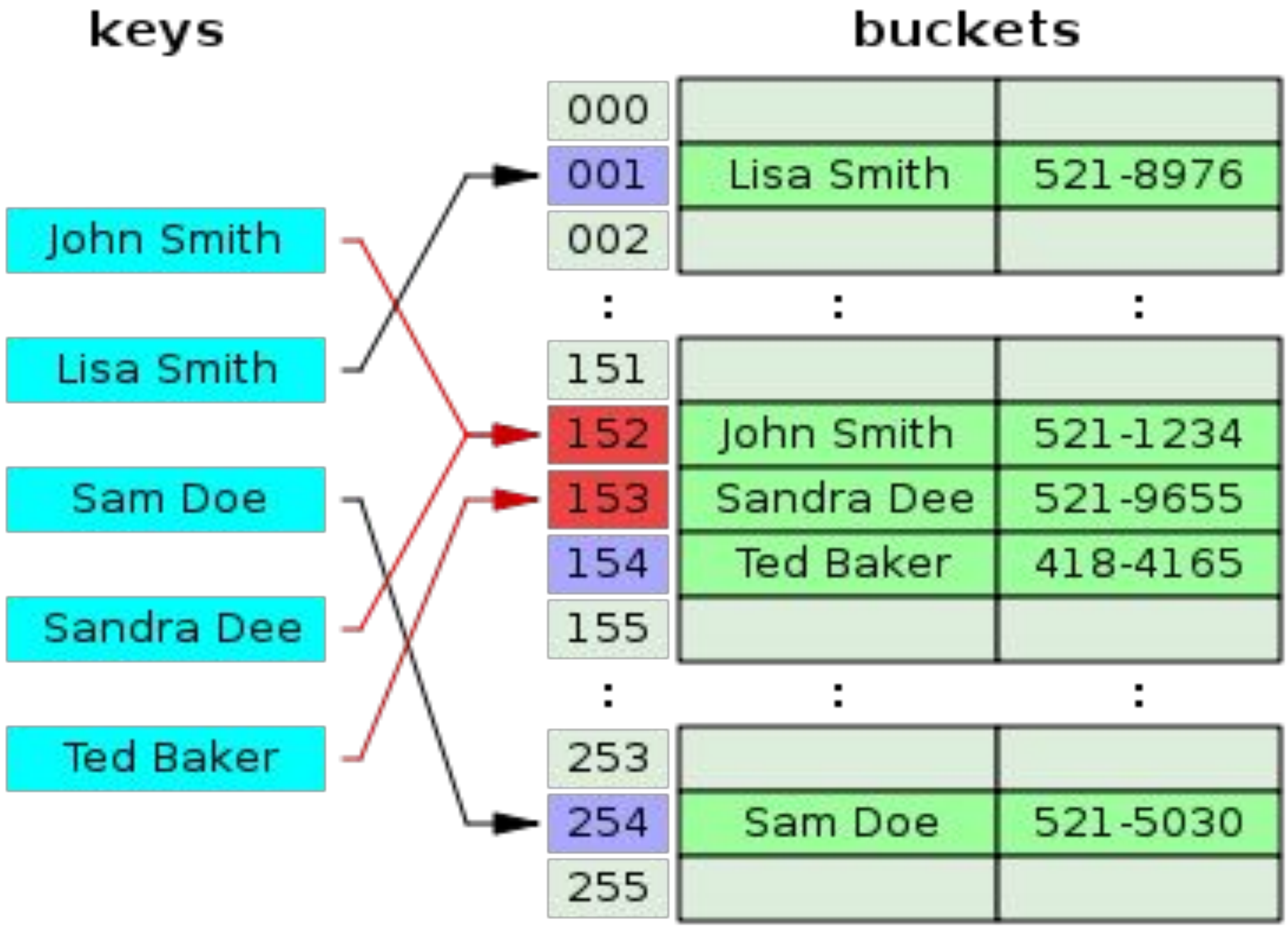
Очевидно, что по мере заполнения хэш-таблицы будут происходить коллизии, и в результате их разрешения очередной адрес может выйти за пределы адресного пространства таблицы.

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.

Закрытое хеширование (пример)



Требования к хеш-функциям

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

- она должна быть ***простой*** с вычислительной точки зрения;
- она должна ***распределять ключи*** в хеш-таблице наиболее ***равномерно***;
- она ***не должна*** отображать какую-либо связь между значениями ключей в связь между значениями адресов;
- она должна ***<u>минимизировать число коллизий</u>***, то есть ситуаций, когда разным ключам соответствует одно значение хеш-функции.

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:

  Key % 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 = “olympiad”, HashTableSize = 100,

o	l	y	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 его в 2-с.с. будет: $100011000_{(2)}$

Свертка: 100 + 11 + 0 = 111

111 % 100 = 11.

Хеш-функция Дженкинса

```
uint32_t jenkins(uint8_t *key, size_t len)
{
    uint32_t hash = 0;
    for (int i = 0; i < len; i++)
    {
       hash += key[i];
       hash += (hash << 10);
       hash ^= (hash >> 6);
    }
    hash += (hash << 3);
    hash ^= (hash >> 11);
    hash += (hash << 15);
    return hash;
}
```

Еще пример хеш-функции

```
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 $\in R$ – иррац.

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

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

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

$m = 2^s$

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

```
uint 32_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 .. 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)