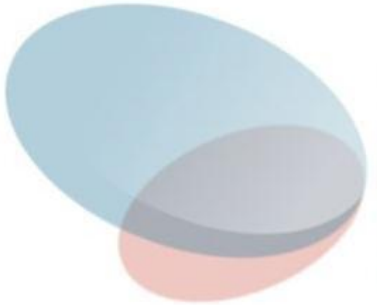


Хеширование
Лекции 7-8

Хеширование

Функция хеширования

Сокращение времени поиска можно осуществить путем локализации (уменьшения) 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.

Широко распространенный метод обеспечения **быстрого доступа к информации**, хранящейся **во внешней памяти – 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**.

Такие преобразования осуществляются с помощью **хеш-функций или функций свертки**, а их **результаты** называют **хешем**, **хеш-кодом, хеш-таблицей** или **дайджестом сообщения** (англ. message digest (digest – краткое изложение, справочник)).

Хеширование (hashing) – это процесс получения индекса элемента массива непосредственно в результате операций, производимых над ключом, который хранится вместе с элементом или даже совпадает с ним. Генерируемый индекс называется *хеш-адресом* (hash).

Хеширование

В рассмотренных ранее методах поиска число итераций в лучшем случае было пропорционально O(log n).

Поставим задачу **найти такой метод поиска, при котором число итераций не зависело бы от размера таблицы**, а в идеальном случае поиск сводился бы к одному шагу.

Таблицы прямого доступа

Простейшей организацией таблицы, обеспечивающей идеально быстрый поиск, является **таблица прямого доступа**. В такой таблице **ключ (поиска) является адресом записи в таблице или может быть преобразован в адрес**, причем таким образом, что **никакие два разных ключа не преобразуются в один и тот же адрес**.

Затем записи вносятся в таблицу – каждая на свое место, определяемое ее ключом.

При поиске **ключ используется как адрес, и по этому адресу выбирается запись.**

Если выбранная запись пустая, то записи с таким ключом в таблице нет.

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.

Хеширование

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 → A;
a = h(k), где a – адрес, k – ключ.

Идеальной хеш-функцией является такая функция, которая для любых двух неодинаковых ключей дает неодинаковые адреса: k1 ≠ k2 □ h(k1) ≠ h(k2). Ей соответствует таблица прямого доступа.

При попытке отображения точек из некоторого широкого пространства в узкое неизбежны ситуации, **когда разные точки в исходном пространстве отобразятся в одну и ту же точку в целевом пространстве**.

Ситуация, при которой **разные ключи отображаются в один и тот же адрес записи**, называется **коллизией (конфликтом)**, или **переполнением**, а такие **ключи** называются **синонимами**.

Другими словами, **коллизия** – это ситуация, когда **разным ключам соответствует одно и то же значение хеш-функции**.

Коллизии составляют основную проблему для хеш-таблиц.

Хеширование

Основные свойства хеш-функции

Необратимость. Если **хеш-функция**, преобразующая ключ в адрес, **может порождать коллизии**, то однозначной **обратной функции** k = h’(a), позволяющей восстановить ключ по известному адресу, **не существует**.

Поэтому **ключ должен обязательно входить в состав записи хешированной таблицы как одно из ее полей**.

Детерминированность. Для одного и того же значения ключа должно получаться одно и то же значение хеш-функции.

Хеширование

К **хеш-функции** в общем случае предъявляются следующие требования:

–она **не должна отображать какую-либо связь между значениями ключей в связь между значениями адресов (**случайное распределение);

– она должна обеспечивать **равномерное распределение отображений фактических ключей по пространству записей**;

– она должна порождать **как можно меньше коллизий** для данного фактического множества записей;

– она **должна быть простой и быстрой для вычисления**.

В алгоритмах криптографии существенно требование линейности алгоритма хеширования – он не должен распараллеливаться.

Хеширование

Если **хеш-функция распределяет совокупность отображений возможных ключей равномерно по множеству индексов**, то хеширование эффективно разбивает множество ключей.

Наихудший случай – когда **все ключи хешируются в один индекс**. При этом мы работаем с одним линейным списком, который и вынуждены последовательно сканировать каждый раз, когда что-нибудь делаем. Отсюда видно, как важна хорошая хеш-функция.

Хеширование

Хеш-таблица

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

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

Хеш-таблица – это структура данных, которая позволяет **хранить пары вида "ключ- значение"** и выполнять три операции:

- **добавления новой пары**,
- **поиска** и
- **удаления пары по ключу**.

Хеширование

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

Области применения

Хэш-таблицы часто применяются **в базах данных**, **в языковых процессорах типа компиляторов и ассемблеров**, где они повышают скорость обработки таблицы идентификаторов.

В качестве использования хеширования в повседневной жизни можно привести следующие примеры:

- **распределение книг в библиотеке по тематическим каталогам**,
- **упорядочивание в словарях по первым буквам слов**,
- **шифрование специальностей в вузах** и т.д.

Хеширование

Типы функций хеширования, их преимущества и недостатки

Приведем анализ некоторых наиболее простых из применяемых на практике хеш-функций.

Таблица прямого доступа

Простейшей организацией таблицы, обеспечивающей идеально быстрый поиск, является **таблица прямого доступа**. В такой таблице **ключ является адресом записи в таблице или может быть преобразован в адрес**, причем таким образом, что **никакие два разных ключа не преобразуются в один и тот же адрес**.

При создании таблицы прямого доступа выделяется память для хранения всей таблицы и таблица заполняется пустыми записями. **Затем записи вносятся в таблицу – каждая на свое место, определяемое ее ключом**. **При поиске ключ используется как адрес, и по этому адресу выбирается запись**. Если выбранная запись пустая, то записи с таким ключом вообще нет в таблице.

Таблицы прямого доступа очень эффективны в использовании, но, к сожалению, область их применения весьма ограничена из-за очень большого размера массива.

Хеширование

Метод деления

Если **ключей больше, чем элементов массива**, то в качестве хеш-функции можно использовать деление по модулю, то есть **остаток от деления целочисленного ключа на размерность массива**.

Построение хеш-функции методом деления состоит в **отображении ключа k в одну из ячеек путем получения остатка от деления k на m,** т.е. **хеш-функция имеет вид h(k)= k mod m. Результат** интерпретируется **как адрес записи**.

При использовании данного метода стараются избегать некоторых значений **m**.

Можно использовать любую размерность массива, но она должна быть такой, чтобы минимизировать число коллизий. Для этого в качестве размерности лучше использовать простое число.

Например, **m не должно быть степенью 2**, поскольку если m=2^p, то h(k) представляет собой просто p младших битов числа k. **Лучше строить хеш-функцию таким образом, чтобы ее результат зависел от всех битов ключа**.

Часто хорошие результаты можно получить, **выбирая в качестве m простое число**, достаточно далекое от степени двойки.

Хеширование

Рекомендации для символьной строки

Ключом может являться остаток от деления, например, **суммы кодов символов строки на размерность массива**.

Можно рассматривать строку символов, как целое число, записанное в соответствующей системе счисления.

Например, идентификатор pt можно рассматривать как пару десятичных чисел (112, 116), поскольку в ASCII- наборе символов код p=112 и код t=116. Рассматривая pt как число в системе счисления с основанием 128, находим, что оно соответствует значению 112*128+116=14452. В конкретных приложениях можно разработать метод для представления ключей в виде больших целых чисел.

- Операция деления по модулю обычно **применяется как последний шаг** в более сложных функциях хеширования и обеспечивает **приведение результата к размеру пространства записей**.

На практике, метод деления – самый распространенный.

Хеширование

Метод умножения (мультипликативный метод)

Построение хеш-функции методом умножения выполняется в два этапа.

1. **Сначала ключ k умножается на константу 0<A<1** и **находится дробная часть полученного произведения**.

2. Затем **полученное значение умножается на m (количество хеш-значений), и к нему применяется функция [ ],** возвращающая наибольшее целое, которое меньше аргумента, т.е.

   $h(k) = [m (kA \bmod 1)]$,

   где выражение $kA \bmod 1$ означает получение дробной части произведения kA, т.е. величину kA-[kA].

Достоинство метода умножения заключается в том, что **значение m можно выбирать произвольно**. Обычно величина **m** выбирается равной степени 2, так как умножение в этом случае реализуется простым сдвигом.

Хеширование

Метод умножения (мультипликативный метод)

Описанный метод вычисления хеш-функции по формуле $h(k) = [m (kA \bmod 1)]$ работает с любыми константами A, но некоторые значения дают лучшие результаты по сравнению с другими.

Кнут предложил использовать дающее неплохие результаты значение A, близкое к золотому сечению

$A \approx (\sqrt{5} - 1)/2 \approx 0.6180339887499\ldots$.

Пример. Пусть A=0.61, k=34, m=16.

Тогда $h(34)=[16^* (34^*0.61 \bmod 1)]=$

$=[16^* (20.74 \bmod 1)]=$

$=[16^*0.74]=[11.84]=11.$

Хеширование

Метод умножения (мультипликативный метод)

Реализация умножения простым сдвигом

h(k) =[m (kA mod 1)], **m** выбирается равным степени 2.

Пусть размер машинного слова равен w.

Ограничим возможные значения константы A значением s/(2^w), где s – целое число из диапазона 0<s<(2^w).

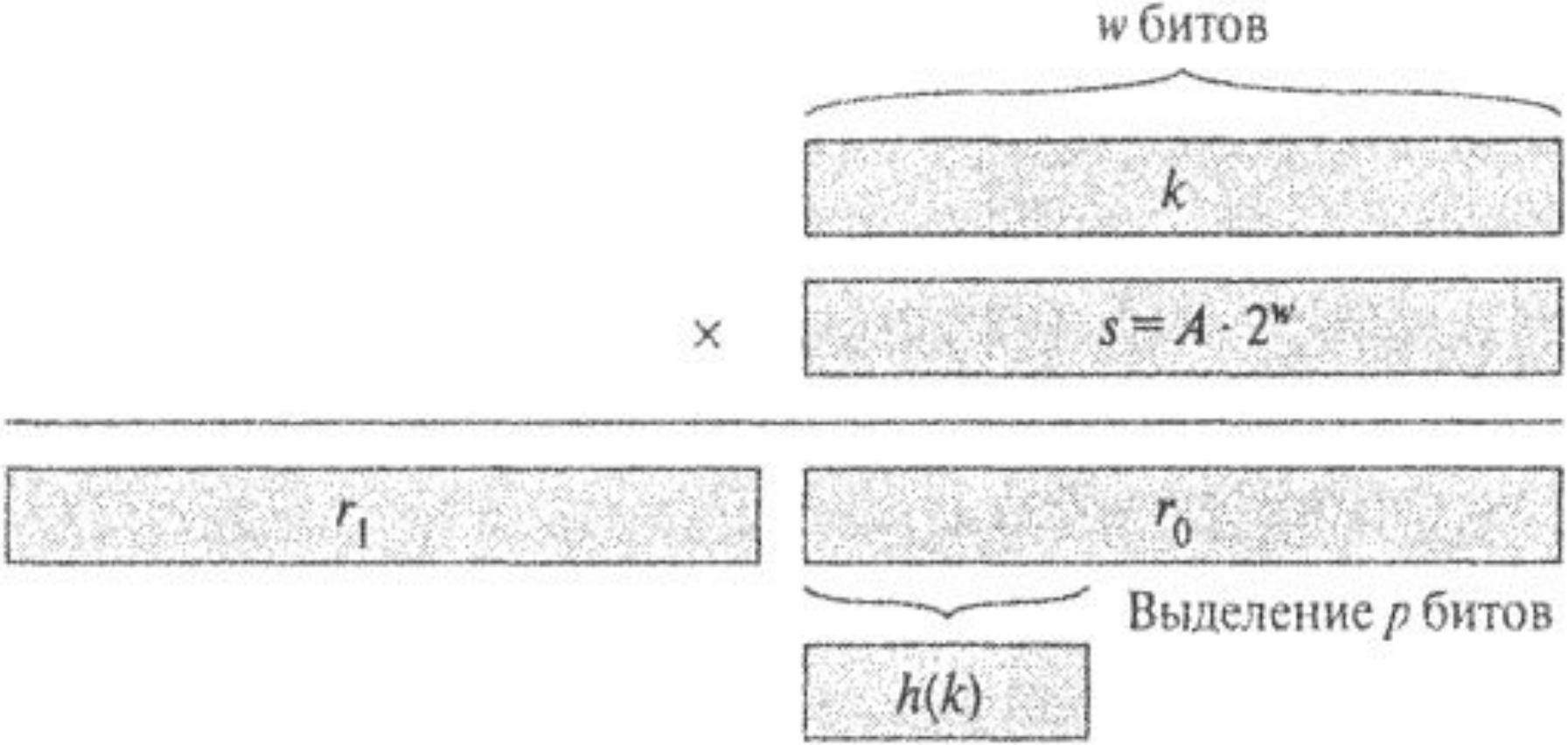
Тогда сначала умножаем k на w-битовое целое число s=A*(2^w).

Результат представляет собой 2w битовое число r1*2^w + r0, где r1 – старшее слово произведения, а r0 – младшее.

Старшие p битов числа r0 представляют собой искомое p-битовое хеш-значение.

Хеширование

Схема метода умножения
Реализация умножения простым сдвигом



Хеширование методом умножения

Хеширование

Метод умножения (мультипликативный метод)

Реализация умножения простым сдвигом

Пример 1

Пусть **k**=123 456; **p**=14; **m**=16384 и **w**=32.

Выбираем A в виде s/(2^w), ближайшее к величине (sqrt(5) - 1)/2 ≈ 0.61, так что A=2 654 435 769 / 2^32.

Тогда k*s=327 706 022 297 664 = (76 300 *2^32) + 17 612 864,

и, соответственно r1 = 76 300 и r0 = 17 612 864.

Старшие 14 битов числа r0 дают хеш-значение h(k)=67.

Пример 2

Пусть размер таблицы hashTableSize есть степень m=**2^p.**

Пусть мы работаем с таблицей из hashTableSize=2^5=32, то есть (2^5) элементов, **p=5**. Хеширование производится байтами (8 бит, unsigned char), **w=8**. Тогда необходимый множитель

s= 2^8*(sqrt(5) - 1)/2 = 158.

Далее, умножая 8-битовый ключ на 158, будем получать некоторое 16-битовое число. Для таблицы длиной 2^5 в качестве хеширующего значения берем **5 старших битов младшего слова**, содержащего такое произведение.

Хеширование

Мультипликативный метод

```
/* 8-bit index */
typedef unsigned char HashIndexType;
static const HashIndexType K = 158; /* 2^8*(sqrt(5) - 1)/2 */

/* 16-bit index */
typedef unsigned short int HashIndexType;
static const HashIndexType K = 40503; /* 2^16*(sqrt(5) - 1)/2 */

/* 32-bit index */
typedef unsigned long int HashIndexType;
static const HashIndexType K = 2654435769; /* 2^32*(sqrt(5) - 1)/2 */
/* w=bitwidth(HashIndexType), size of table=2**p */
static const int S = w - p;
/* Преобразование типа убирает старшее слово, т.е
 * лишние биты слева, а сдвиг убирает лишнее справа
*/
HashIndexType HashValue = (HashIndexType)(K * Key) >> S;
```

Хеширование

Мультипликативный метод

Пример. Пусть размер таблицы hashTableSize равен 1024 (2^10). Тогда нам достаточен 16-битный индекс и величине сдвига S будет присвоено значение S=w-p=16 - 10 = 6. В итоге получаем:

```
typedef unsigned short int HashIndexType;

HashIndexType Hash(int Key) {
    static const HashIndexType K = 40503;
    static const int S = 6;
    return (HashIndexType)(K * Key) >> S;
```

Хеширование

Аддитивный метод для строк переменной длины

(Размер таблицы равен 256). Для строк переменной длины хорошие результаты дает **сложение по модулю 256** (сложение всех символов и возврат остатка от деления на 256).

В этом случае результат hashValue заключен между 0 и 255.

```
typedef unsigned char HashIndexType;

HashIndexType Hash(char *str) {
    HashIndexType h = 0;
    while (*str) h += *str++;
    return h;
}
```

Хеширование

Исключающее ИЛИ для строк переменной длины

(Размер таблицы равен 256). Этот метод аналогичен аддитивному, но успешно различает схожие слова и анаграммы (аддитивный метод даст одно значение для XY и YX). Метод заключается в том, что **к элементам строки последовательно применяется операция "исключающее или".** В нижеследующем алгоритме добавляется случайная компонента, чтобы еще улучшить результат.

```
typedef unsigned char HashIndexType;

unsigned char Rand8[256];

HashIndexType Hash(char *str) {
    unsigned char h = 0;
    while (*str) h = Rand8[h ^ *str++];
    return h;
}
```

Здесь Rand8 – таблица из 256 восьмибитовых случайных чисел. Их точный порядок не критичен. Корни этого метода лежат в криптографии; он оказался вполне эффективным.

Хеширование

Исключающее ИЛИ для строк переменной длины

(Размер таблицы <= 65536). Если **хешируем строку дважды**, то получим хеш-значение для таблицы любой длины до 65536. **Когда строка хешируется во второй раз**, **к первому символу прибавляется 1**. Получаемые два 8-битовых числа объединяются в одно 16-битовое.

Логическая операция исключающее ИЛИ выполняется с двумя битами (a и b). Результат выполнения логической операции XOR будет равен 1 (единице), если один из битов a или b равен 1 (единице), а другой равен 0 (нулю); в остальных случаях, когда биты имеют равные значения, результат равен 0 (нулю).

Метод функции середины квадрата

Следующей хеш-функцией является функция середины квадрата. Значение ключа преобразуется в число, это **число** затем **возводится в квадрат**, из него **выбираются несколько средних цифр** (определенное количество) и **интерпретируются как адрес записи**.

Хеширование

Метод свертки

Еще одной хеш-функцией можно назвать функцию 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, для сравнительно небольших таблиц с ключами – символьными строками неплохие результаты дает функция хеширования, в которой **адрес записи получается в результате сложения кодов символов, составляющих строку-ключ**.

Преобразование системы счисления

В качестве хеш-функции также применяют **функцию преобразования системы счисления**.

Ключ, записанный как число в некоторой системе счисления P, интерпретируется как число в системе счисления Q>P. Обычно выбирают Q=P+1. **Это число переводится из системы Q обратно в систему P, приводится к размеру пространства записей** и интерпретируется как адрес.

Хеширование

Методы разрешения коллизий

С помощью хеш-функций ключи преобразуются в адреса таблицы в заданном диапазоне. После того как выбрана хеш-функция, необходимо выбрать способ разрешения коллизий.

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

Способы разрешения коллизий:

- **метод цепочек** (**внешнее или открытое хеширование**);
- **метод открытой адресации** (**закрытое хеширование**).

Хеширование

Открытое (внешнее) хеширование или метод цепочек

– это технология разрешения коллизий, которая состоит в том, что **элементы множества с равными хеш-значениями связываются в цепочку-список.**

Основная идея базовой структуры при открытом (внешнем) хешировании заключается в том, что потенциальное **множество** (возможно, бесконечное) **разбивается на конечное число классов (сегментов)**.

Для B классов, пронумерованных от 0 до B-1, **строится хеш-функция h(k)** такая, что **для любого элемента k исходного множества** функция h(k) **принимает целочисленное значение из интервала 0,1,...,B-1**, соответствующее классу (сегменту), которому принадлежит элемент **k**.

Будем говорить, что элемент k принадлежит сегменту h(k). **Массив**, называемый таблицей сегментов и проиндексированный номерами сегментов 0,1,...,B-1, **содержит заголовки для B списков**. Элемент **k**, относящийся к i - у списку – это элемент исходного множества, для которого h(k) = i.

Хеширование

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

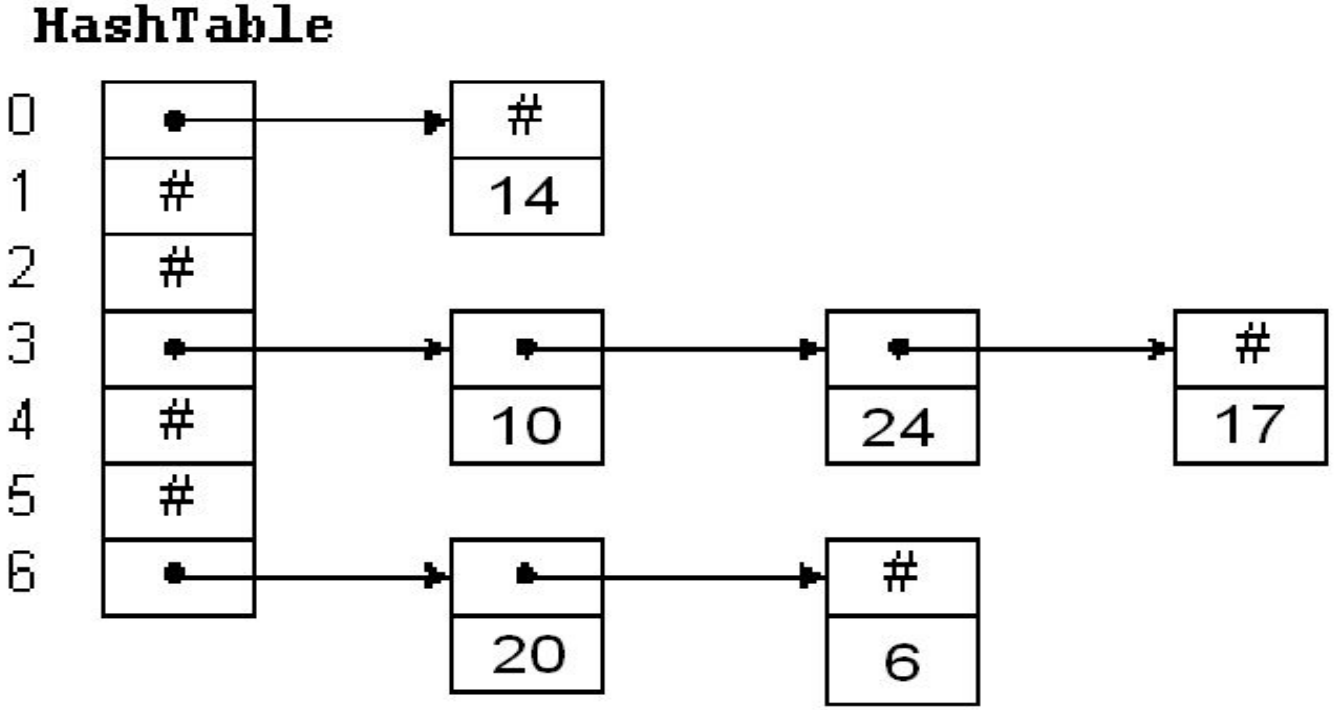
В позиции номер i хранится указатель на голову списка тех элементов, у которых хеш-значение ключа равно i.

Если таких элементов во множестве нет, в позиции i записан NULL.

Хеширование

Пример

На рисунке **hashTable – массив из 7 элементов**. Каждый элемент представляет собой указатель на линейный список, хранящий числа. **Хеш-функция** в примере **h(k)= k mod 7 делит ключ на 7 и использует остаток как индекс в таблице**. Это дает числа от 0 до 6. Поскольку для адресации в hashTable и нужны числа от 0 до 6, алгоритм гарантирует допустимые значения индексов.



Хеширование

Чтобы вставить в таблицу новый элемент, хешируем ключ, чтобы **определить список, в который его нужно добавить**, затем **вставляем элемент в начало этого списка**. **Например**, чтобы добавить 10, делим 10 на 7 и получаем остаток 3, 10 следует разместить в списке, на начало которого указывает hashTable[3].

Чтобы найти число, хешируем его и проходим по соответствующему списку.

Чтобы удалить число, находим его и удаляем элемент из содержащего это число списка.

При добавлении элементов в хеш-таблицу выделяются куски динамической памяти, которые организуются в виде связанных списков, каждый из которых соответствует входу хеш-таблицы. Поэтому этот метод называется **связыванием или методом цепочек**.

Хеширование

Пример. Как определить размер хеш-таблицы

Предположим, мы хотим создать хеш-таблицу с разрешением коллизий методом цепочек для хранения 2000 символьных строк, размер символов в которых равен 8 битам.

Пусть нас устраивает проверка в среднем трех элементов при неудачном поиске, **коэффициент заполнения хеш-таблицы** =3, поэтому мы можем выбрать размер таблицы равным m=701. Число 701 выбрано как простое число, близкое к величине 2000/3 и не делящееся на 2.

Рассматривая ключ k как целое число, получаем **h(k)= k mod 701.**

Хеширование

Чем меньше таблица, тем больше среднее время поиска ключа в ней (при фиксированном количестве элементов). Хеш-таблицу можно рассматривать как совокупность связанных списков. По мере того, как таблица растет, увеличивается количество списков и, соответственно, среднее число узлов в каждом списке уменьшается.

Если **исходное множество состоит из n элементов** и сегменты примерно одинаковы по размеру, то **средняя длина списков будет n/B элементов**, где B – число сегментов.

Если размер таблицы равен 1, то таблица вырождается в один список длины n. Если размер таблицы равен 2, и хеширование идеально, то придется иметь дело с двумя списками по n/2 элементов в каждом. Это сильно уменьшает длину списка, в котором нужно искать.

Идеальным хешированием называется метод, который в наихудшем случае выполняет поиск за O(1) обращений к памяти.

Хеширование

Если можно оценить величину **n** и выбрать **B** как можно ближе к этой величине, то в каждом списке будет один или два элемента. Тогда **время выполнения операторов поиска**, добавления, удаления **будет малой постоянной величиной, не зависящей от n**.

Среднее время поиска элемента есть **O(1), время для наихудшего случая – O(n)**.

Хеширование

Каждая **ячейка массива (хеш-таблицы)** является **указателем на связанный список (цепочку) пар ключ-значение**, соответствующих одному и тому же хеш-значению ключа. **Коллизии 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.

Коэффициент заполнения хеш-таблицы k=n/m – это количество хранимых элементов массива **n**, деленное на число возможных значений хеш-функции **m**.

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**.

Хеширование

Преимущества и недостатки связывания

Одно из **преимуществ** этого метода состоит в том, что **при его использовании хеш-таблицы никогда не переполняются**.

Один из **недостатков** связывания состоит в том, что если число связанных списков недостаточно велико, то **размер списков может стать большим**, при этом для вставки или поиска элемента необходимо будет проверить большое число элементов списка.

Можно немного ускорить поиск, если **использовать упорядоченные списки**. Тогда можно использовать методы поиска элементов в упорядоченных связных списках.

Использование упорядоченных списков позволяет ускорить поиск, но **не снимает настоящую проблему, связанную с большим размером списка**.

Лучшим, но более трудоемким решением, будет **создание хеш-таблицы большего размера** и **повторное хеширование элементов в новой таблице** так, чтобы связанные списки в ней имели меньший размер.

Хеширование

Блочное хеширование

Другой способ разрешения **коллизий** заключается в том, чтобы **выделить ряд блоков, каждый из которых может содержать несколько элементов**. Для вставки **элемента** в таблицу, он **отображается на один из блоков и затем помещается в этот блок.**

Если блок уже заполнен, то используется **обработка переполнения**.

Самый простой **метод обработки переполнения** состоит в том, чтобы **поместить все лишние элементы в специальные блоки в конце массива** «нормальных» блоков. Это позволяет при необходимости легко увеличивать размер хеш-таблицы. Если требуется больше дополнительных блоков, то размер массива блоков просто увеличивается, и в конце массива создаются новые дополнительные блоки.

Например, чтобы добавить новый элемент k в хеш-таблицу, которая содержит 5 блоков, вначале пытаемся поместить его в блок с номером k Mod 5. Если этот блок заполнен, элемент помещается в дополнительный блок.

Хеширование

Чтобы реализовать схему блочного хеширования, можно использовать массив указателей на блоки (массивы фиксированной длины), которые (массивы указателей) представляют собой массивы изменяемого размера.

Пример

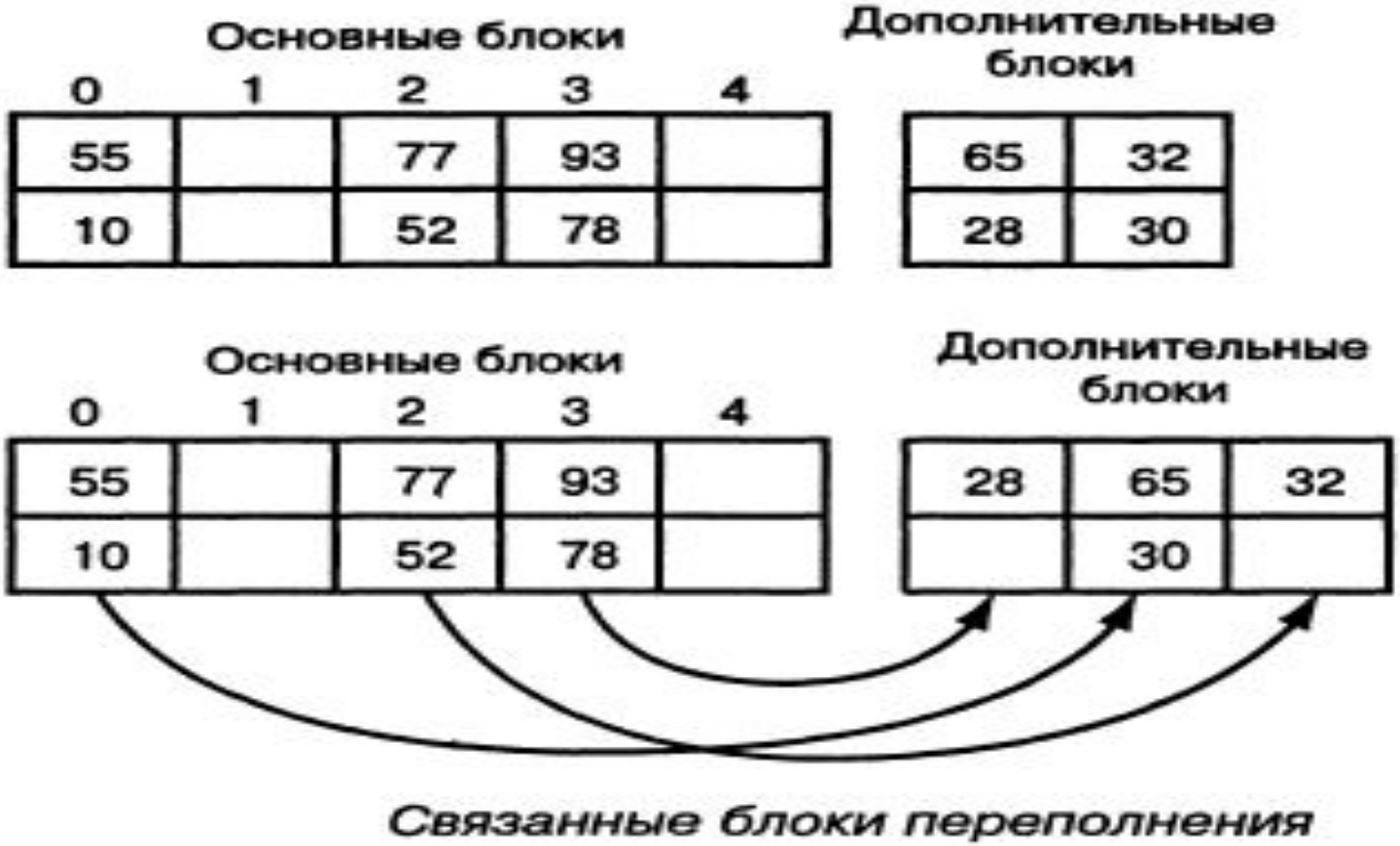
Хеширование с использованием блоков

Хеширование

Связывание блоков

Вместо того, чтобы хранить все лишние элементы в одних и тех же дополнительных блоках, **для каждого заполненного блока создается своя цепочка блоков.** Если множество блоков переполнено, то это может сэкономить довольно много времени.

Два варианта расположения элементов в блоках h(k)=k mod 5.



Хеширование

Преимущества и недостатки применения блоков

Вставка и добавление элемента в хеш-таблицу с блоками выполняется достаточно быстро, даже если таблица почти заполнена.

Операции над хеш-таблицей, использующей блоки, обычно выполняются быстрее, чем над таблицей со связанными списками.

Если хеш-таблица находится на диске, **блочный алгоритм может считывать за одно обращение к диску весь блок**.

При использовании связанных списков, следующий элемент может находиться на диске не обязательно рядом с предыдущим. При этом для каждой проверки элемента потребуется обращение к диску.

Удаление элемента из таблицы сложнее выполнить с использованием блоков, чем при применении связанных списков.

Хеширование

Закрытое (внутреннее) хеширование или метод открытой адресации

– это технология разрешения коллизий, которая предполагает **хранение записей в самой хеш-таблице.**

Каждая ячейка таблицы содержит либо элемент динамического множества, либо NULL.

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

В этом случае, если **ячейка с вычисленным индексом занята**, то можно **просматривать следующие записи** таблицы в *определенном порядке* до тех пор, **пока не будет найден ключ k** или **пустая позиция в таблице**.

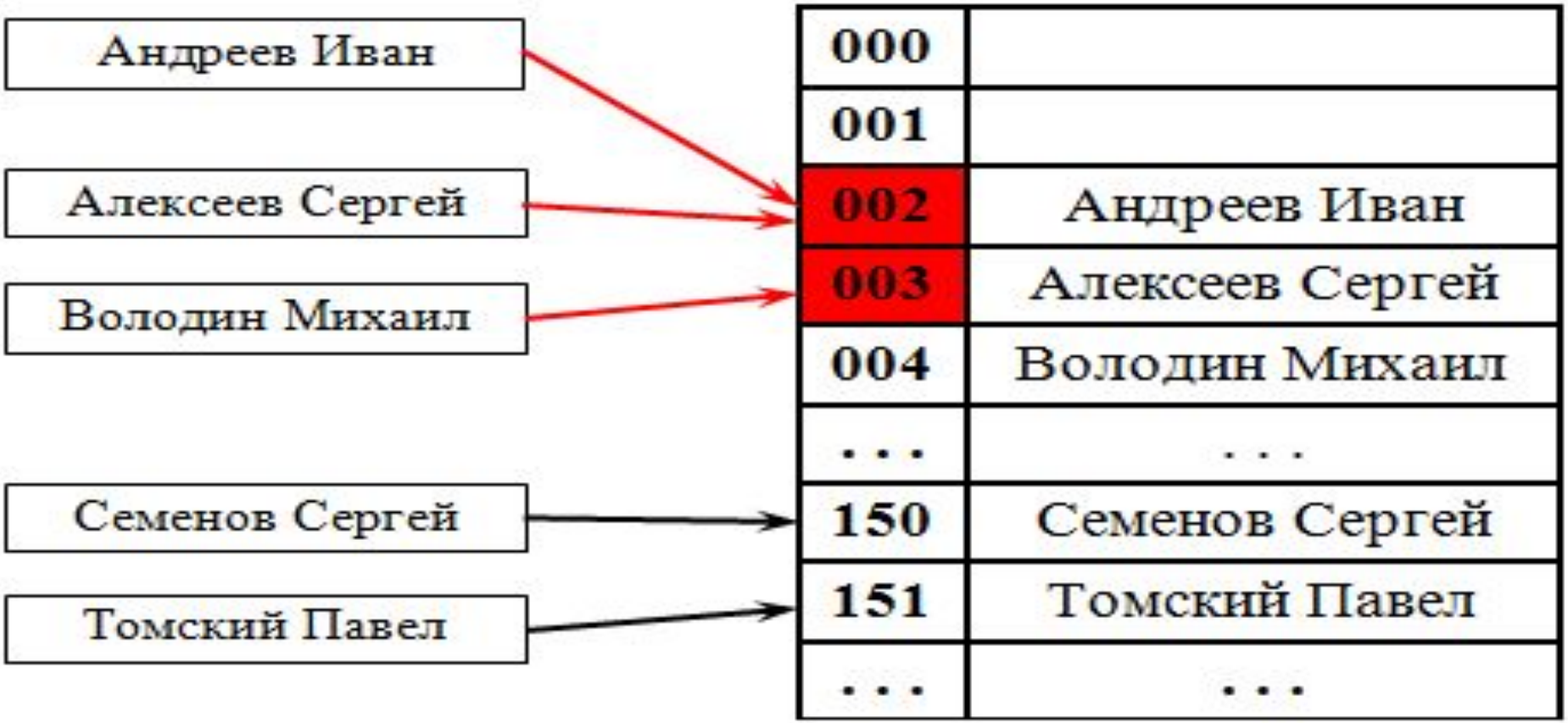
Для вычисления шага можно **применить формулу**, которая и **определит способ изменения шага**.

Порядок следования элементов в хеш-таблице зависит от того, в каком порядке мы добавляем данные в хеш-таблицу.

Размер хеш-таблицы должен быть достаточно большим, чтобы в ней оставалось разумно большое число пустых мест.

Хеширование

Пример. На рисунке **разрешение коллизий** осуществляется **методом открытой адресации**. Пусть два значения претендуют на ключ 002, для одного из них ‘Андреев’ находится первое свободное (еще незанятое) место в таблице. Теперь **ячейка с ключом 002 занята. Если шаг=1, то значение ‘Алексеев’ помещается в следующую пустую ячейку с индексом 003.** Пусть значение ‘Волошин’ претендует на ячейку с ключом 003, но она занята. Поэтому оно помещается в следующую пустую ячейку с индексом 004 (шаг=1).



Разрешение коллизий при помощи открытой адресации

Хеширование

Методика повторного хеширования

Добавление элемента

При закрытом хешировании применяется **методика повторного хеширования**. Если осуществляется попытка поместить элемент **k** в сегмент с номером **h(k),** который уже занят другим элементом (коллизия), то **в соответствии с методикой повторного хеширования выбирается последовательность других номеров сегментов h1(k),h2(k),..., куда можно поместить элемент k**.

Каждое из этих местоположений последовательно проверяется, **пока не будет найдено свободное место**. Если свободных сегментов нет, то, следовательно, таблица заполнена, и элемент **k** добавить нельзя.

Хеширование

Поиск элемента

При поиске элемента k необходимо **просмотреть все местоположения h(k),h1(k),h2(k),..., пока не будет найден k или пока не встретится пустой сегмент**.

А)Предположим, что **в хеш-таблице не допускается удаление элементов**. Пусть h3(k) – первый пустой сегмент. В такой ситуации невозможно нахождение элемента **k** в сегментах h4(k), h5(k) и далее, так как при вставке элемент **k** вставляется в первый пустой сегмент, следовательно, он находится где-то до сегмента h3(k). Значит элемент **k** отсутствует.

Б)Если **в хеш-таблице допускается удаление элементов**, то не найдя элемента **k**, при достижении пустого сегмента нельзя быть уверенным в том, что его вообще нет в таблице, так как сегмент может стать пустым уже после вставки элемента **k**.

Рассмотрим два подхода к организации корректного удаления элемента из таблицы.

Первый подход. Необходимо осуществить повторное хеширование всех элементов с h-го (позиция удаляемого элемента) до первого после него пустого, т.е. необходимо заново разместить в таблице все элементы между удаленным элементом и следующей незанятой позицией таблицы.

Хеширование

Второй подход. Чтобы увеличить эффективность данной реализации, **необходимо в сегмент, который освободился после операции удаления элемента, поместить специальную константу**, например, **deleted**.

В качестве альтернативы специальной константе можно использовать **дополнительное поле таблицы, которое показывает состояние элемента.**

При этом необходимо **модифицировать процедуру поиска существующего элемента** так, **чтобы она считала удаленные ячейки занятыми**, а **процедуру добавления – чтобы она их считала свободными** и сбрасывала значение флага при добавлении.

Важно различать константы **deleted** и **NULL** – последняя находится в сегментах, которые никогда не содержали элементов.

При таком подходе **выполнение поиска элемента** не требует просмотра всей хеш-таблицы.

Кроме того, **при вставке элементов** сегменты, помеченные константой **deleted**, можно трактовать **как свободные**, таким образом, пространство, освобожденное после удаления элементов, можно рано или поздно использовать повторно.

Хеширование

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

В любой момент должно соблюдаться **требование**: **для любого элемента множества участок справа от его исконного места до его фактического места полностью заполнен**.

Благодаря этому **поиск элемента k** осуществляется легко: **встав на h(k), двигаемся направо, пока не дойдем до пустого места или до элемента k**. В первом случае элемент **k** отсутствует в множестве, во втором присутствует.

Если элемент отсутствует, то его можно добавить на найденное пустое место.

Если присутствует, то можно его удалить.

Хеширование

Повторное хеширование

– это поиск местоположения для очередного элемента таблицы с учетом шага перемещения.

Методы повторного хеширования (определение местоположений h(k), h1(k), h2(k),...):

- **линейное опробование (зондирование)**;
- **квадратичное опробование (зондирование)**;
- **двойное хеширование**.

Линейное опробование

сводится к последовательному перебору сегментов таблицы с некоторым фиксированным шагом:

Адрес = (h(k)+c*i) mod m, i=0,1, …

где i – **номер попытки** разрешить коллизию;

c – константа, определяющая **шаг перебора**.

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

Хеширование

Квадратичное опробование

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

Адрес = (h(k)+c*i +d*i^2) mod m, i=0,1, …

где i – **номер попытки** разрешить коллизию,

c и d – **константы**.

Благодаря нелинейности такой адресации **уменьшается число проб при большом числе ключей-синонимов**.

Однако даже относительно небольшое число проб может быстро привести к выходу за адресное пространство небольшой таблицы вследствие квадратичной зависимости адреса от номера попытки.

Хеширование

Двойное хеширование

Основано на **нелинейной адресации,** достигаемой за счет **суммирования значений основной и дополнительной хеш-функций**:

Адрес = (h1(k)+i*h2(k)) mod m, i=0,1, …

По мере заполнения хеш-таблицы могут происходить коллизии, и в результате их разрешения очередной адрес может выйти за пределы адресного пространства таблицы.

Чтобы последовательность испробованных мест покрыла всю таблицу, можно, например, задать хеш-функции следующим образом:

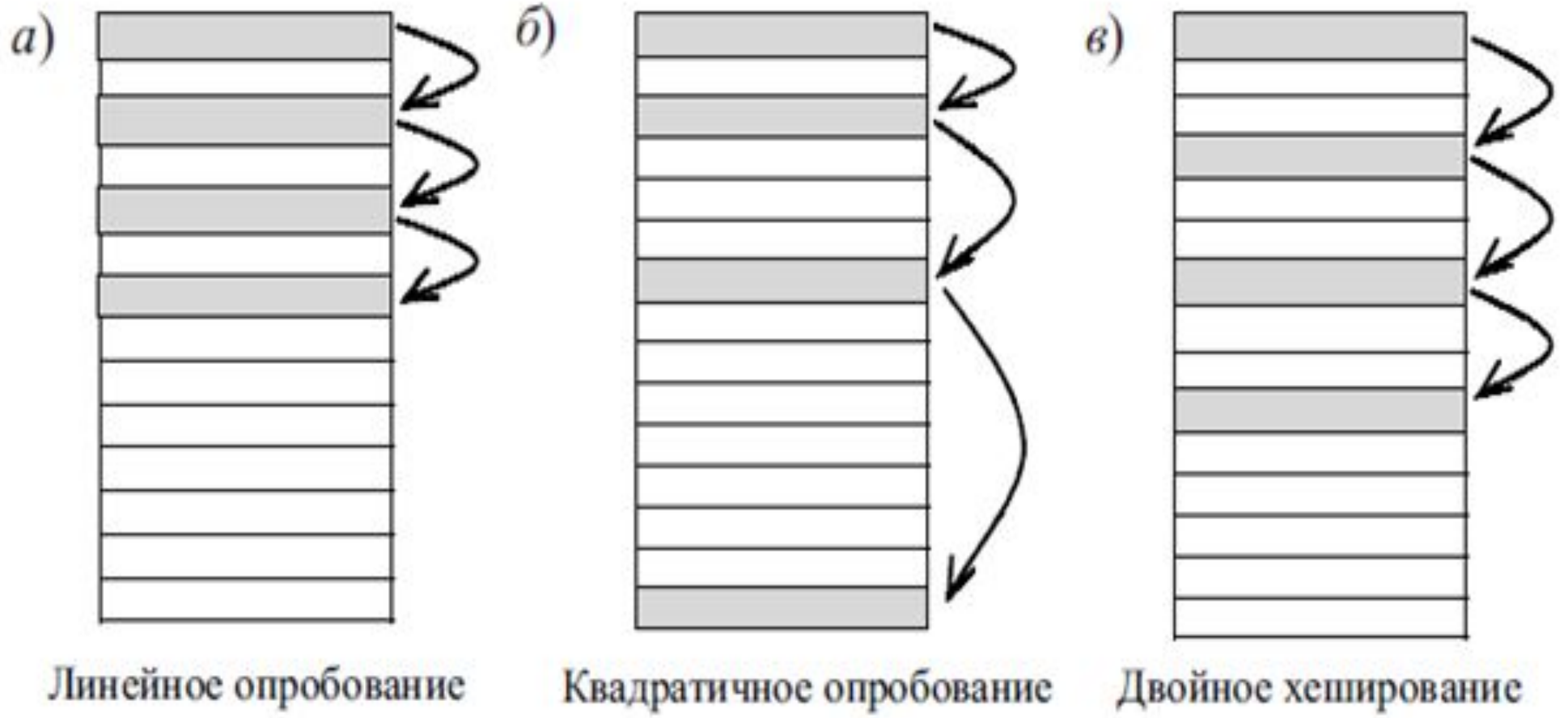
$h1(k)=k \mod m$,

$h2(k)=1+(k \mod m1)$,

где m – простое, $m1 < m$ (например, m1 = m – 1 или m1 = m – 2).

Хеширование

Методы повторного хеширования



Организация данных при закрытом хешировании

Хеширование

Реструктуризация хеш-таблиц

Чтобы выход за пределы адресного пространства таблицы происходил реже, **можно увеличить длину таблицы по сравнению с диапазоном адресов**, выдаваемым хеш-функцией. С одной стороны, это приведет к сокращению числа коллизий и ускорению работы с хеш-таблицей, а с другой – к нерациональному расходованию памяти.

Даже при увеличении длины таблицы в два раза по сравнению с областью значений хеш-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.**

Хеширование

Циклический переход к началу таблицы

В случае многократного превышения адресного пространства и, соответственно, **многократного циклического перехода к началу** будет происходить **просмотр одних и тех** же ранее занятых **сегментов**, тогда как между ними могут быть еще свободные сегменты.

Более корректным будет использование **сдвига адреса на 1 в случае каждого циклического перехода** к началу таблицы. Это повышает **вероятность нахождения свободных сегментов.**

Хеширование

Анализ скорости выполнения операций вставки и других операций при закрытом хешировании

В случае применения схемы закрытого хеширования **скорость выполнения вставки и других операций** зависит **не только от равномерности распределения элементов** по сегментам хеш-функцией, но и **от выбранной методики повторного хеширования (опробования)** для разрешения коллизий, связанных с попытками вставки элементов в уже заполненные сегменты.

Например, методика линейного опробования для разрешения коллизий – не самый лучший выбор.

Как только несколько последовательных сегментов будут заполнены, образуя группу, любой новый элемент при попытке вставки в эти сегменты будет вставлен в конец этой группы, увеличивая тем самым длину группы последовательно заполненных сегментов. Другими словами, **для поиска пустого сегмента в случае непрерывного расположения заполненных сегментов необходимо просмотреть больше сегментов, чем при случайном распределении заполненных сегментов.**

Вывод. При непрерывном расположении заполненных сегментов увеличивается время выполнения вставки нового элемента и других операций.

Хеширование

Преимущества и недостатки закрытого хеширования

Используется массив определенной длины, превышающей количество данных, **требующий больше памяти, чем при внешнем хешировании.**

Закрытые хеш-таблицы особенно эффективны, когда **максимальные размеры входящего набора данных уже известны**. Доказано, что, когда закрытая таблица заполняется более чем на 50 процентов, ее эффективность значительно ухудшается.

Закрытые таблицы обычно **используются для быстродействующего макетирования** (создания макетов, они просты для программирования и быстры) и там, **где быстрый доступ** (нет связанного списка) **приоритетен**, **а память легко доступна.**

Хеширование

При любом методе разрешения коллизий необходимо ограничить количество операций при поиске элемента. Если для поиска элемента необходимо более 3 – 4 сравнений, то эффективность использования такой хеш-таблицы пропадает и ее следует реструктуризировать (т.е. найти другую хеш-функцию), чтобы **минимизировать количество сравнений для поиска элемента.**

Для успешной работы алгоритмов поиска, **последовательность проб должна быть такой, чтобы все ячейки хеш-таблицы оказались просмотренными ровно по одному разу**.

Хеширование

Задания

1. Применить повторное хеширование для вставки элементов 18, 14, 9, 20, 19, 12, 5, 27, 16, 34 в первоначально пустую хеш-таблицу (размером m=11) с открытой адресацией. Разрешение конфликтов (коллизий) производить с применением линейного зондирования с шагом c=1.

Найти в таблице элементы 16, 29, 15.

Удалить из таблицы элементы 9. Вставить элемент 30.

2. Применить повторное хеширование для вставки элементов 18, 14, 9, 20, 19, 12, 5, 27, 16, 34 в первоначально пустую хеш-таблицу (размером m=11) с открытой адресацией. Для разрешения коллизий использовать двойное хеширование с хеш-функциями h1(k)=k mod 11 и h2(k)=1+(k mod 7).

Найти в таблице элементы 16, 29, 15.

Удалить из таблицы элементы 9. Вставить элемент 30.

Указание. Пустые ячейки помечать Empty, а с удаленными элементами Del.

Хеширование

Идеальное хеширование

Идеальным хешированием называется метод, который в наихудшем случае выполняет поиск за O(1) обращений к памяти.

В этом случае используется двухуровневая методика хеширования с универсальным хешированием на каждом уровне.

Первый уровень аналогичен хешированию с исключением коллизий методом цепочек с хеш-функцией $h(k): K \rightarrow \{0, 1, 2, \ldots, m-1\}$.

На втором уровне вместо списков ключей синонимов используются небольшие вторичные хеш-таблицы S_j, каждая со своей хеш-функцией $h_j(k)$, $j=0, 1, 2, \ldots, m-1$, которые формируются методом открытой адресации (закрытого хеширования).

Все хеш-функции выбираются из универсального семейства F.

Определение. Пусть F – конечное семейство функций, отображающее множество ключей K на множество значений хеш-функции {0, 1, 2, …, m-1}. Семейство F называется **универсальным**, если для любых двух ключей $k1, k2 \in K$ число функций $h \in F$, для которых $h(k1) = h(k2)$, равно $|F|/m$.

Хеширование

Идея хеширования впервые была высказана **Г.П. Ланом** (Hans Peter Luhn (July 1, 1896 – August 19, 1964) - ученый компьютерщик, работал в IBM) при создании внутреннего меморандума в январе 1953 г. с предложением **использовать для разрешения коллизий метод цепочек**.

Примерно в это же время другой сотрудник IBM, **Жини М. Амдал** (Gene Myron Amdahl - ученый компьютерщик), высказал идею использования **открытой линейной адресации**.

В открытой печати хеширование впервые было описано **Арнольдом Думи** (Arnold I. Dumey) в 1956 году, указавшим, что в качестве **хеш-адреса** удобно **использовать остаток от деления на простое число**. А. Думи **описывал метод цепочек** для разрешения коллизий, но не говорил об открытой адресации.

Подход к хешированию, отличный от метода цепочек, был предложен **А.П. Ершовым** в 1957 году, который разработал и описал **метод линейной открытой адресации**.

(Интернет-Университет Информационных Технологий http://www.INTUIT.ru)

Хеширование

Андре́й Петро́вич Ершо́в (19 апреля 1931 - 8 декабря 1988) — советский учёный, один из основателей теоретического и системного программирования, создатель Сибирской школы информатики, академик АН СССР. Его работы оказали огромное влияние на формирование и развитие вычислительной техники не только в СССР, но и во всём мире.

Большее влияние работы Андрея Петровича Ершова оказали на будущего идеолога программирования Дональда Кнута, впоследствии они стали друзьями.

Из Воспоминаний Дональда Кнута об Андрее Ершове:

‘Это началось ещё когда я был студентом последнего курса. Тогда только появилась книга Андрея «Программирование для БЭСМ», и мы, группа студентов, смогли убедить преподавателя русского языка включить её в курс в качестве одного из двух сборников текстов для изучения научной лексики’.

Хеширование

Пример. Программная реализация закрытого хеширования. С.1.

```
#include "stdafx.h"
#include <iostream>
#include <fstream>
using namespace std;

typedef int T;  // тип элементов
typedef int hashTableIndex;// индекс в хеш-таблице
int hashTableSize;
T *hashTable;
bool *used;

hashTableIndex myhash(T data);
void insertData(T data);
void deleteData(T data);
bool findData (T data);
int dist (hashTableIndex a,hashTableIndex b);
```

Хеширование

Пример. Программная реализация закрытого хеширования. С.2.

```
int _tmain(int argc, _TCHAR* argv[]){
  int i, *a, maxnum;
  cout << "Введите количество элементов maxnum : ";
  cin >> maxnum;
    cout << "Введите размер хеш-таблицы hashTableSize : ";
  cin >> hashTableSize;
  a = new int[maxnum];
  hashTable = new T[hashTableSize];
  used = new bool[hashTableSize];
  for (i = 0; i < hashTableSize; i++){
    hashTable[i] = 0;
    used[i] = false;
  }
 // генерация массива
  for (i = 0; i < maxnum; i++)
    a[i] = rand();
```

Хеширование

Пример. Программная реализация закрытого хеширования. С.3.

```
// заполнение хеш-таблицы элементами массива
 for (i = 0; i < maxnum; i++)
   insertData(a[i]);
 // поиск элементов массива по хеш-таблице
 for (i = maxnum-1; i >= 0; i--)
   findData(a[i]);
 // вывод элементов массива в файл List.txt
 ofstream out("List.txt");
 for (i = 0; i < maxnum; i++){
   out << a[i];
   if ( i < maxnum - 1 ) out << "\t";
 }
 out.close();
 // сохранение хеш-таблицы в файл HashTable.txt
 out.open("HashTable.txt");
 for (i = 0; i < hashTableSize; i++){
   out << i << "  :  " << used[i] << " : " << hashTable[i] << endl;
 }
 out.close();
```

Хеширование

Пример. Программная реализация закрытого хеширования. С.4.

```
// очистка хеш-таблицы
  for (i = maxnum-1; i >= 0; i--) {
    deleteData(a[i]);
  }
  system("pause");
  return 0;
}

// хеш-функция размещения величины
hashTableIndex myhash(T data) {
    return (data % hashTableSize);
}
```

Хеширование

Пример. Программная реализация закрытого хеширования. С.5.

```
// функция поиска местоположения и вставки величины в таблицу
void insertData(T data) {
  hashTableIndex bucket;
    bucket = myhash(data);
  while  ( used[bucket] && hashTable[bucket] != data)
    bucket = (bucket + 1) % hashTableSize;
  if ( !used[bucket] ) {
    used[bucket] = true;
    hashTable[bucket] = data;
  }
}
// функция поиска величины, равной data
bool findData (T data) {
  hashTableIndex bucket;
  bucket = myhash(data);
  while ( used[bucket] && hashTable[bucket] != data )
    bucket = (bucket + 1) % hashTableSize;
  return used[bucket] && hashTable[bucket] == data;
}
```

Хеширование

Пример. Программная реализация закрытого хеширования. С.6.

```
//функция удаления величины из таблицы
void deleteData(T data){
  int bucket, gap;
  bucket = myhash(data);
  while ( used[bucket] && hashTable[bucket] != data )
    bucket = (bucket + 1) % hashTableSize;
  if ( used[bucket] && hashTable[bucket] == data ){
    used[bucket] = false;
    gap = bucket;
    bucket = (bucket + 1) % hashTableSize;
    while ( used[bucket] ){
      if ( bucket == myhash(hashTable[bucket]) )
        bucket = (bucket + 1) % hashTableSize;
      else if ( dist(myhash(hashTable[bucket]),bucket) < dist(gap,bucket) )
        bucket = (bucket + 1) % hashTableSize;
```

Хеширование

Пример. Программная реализация закрытого хеширования. С.7.

```
      else {
        used[gap] = true;
        hashTable[gap] = hashTable[bucket];
        used[bucket] = false;
        gap = bucket;
        bucket++;
      }
    }
  }
}
// функция вычисления расстояние от a до b (по часовой стрелке,
       слева направо)
int dist (hashTableIndex a,hashTableIndex b){
  return (b - a + hashTableSize) % hashTableSize;
}
```

Хеширование

Пример. Программная реализация открытого хеширования. С.1.

```
#include "stdafx.h"
#include <iostream>
#include <fstream>
using namespace std;

typedef int T;  // тип элементов
typedef int hashTableIndex; // индекс в хеш-таблице
#define compEQ(a,b) (a == b)
typedef struct Node_ {
	T data;// данные, хранящиеся в вершине
	struct Node_ *next; // следующая вершина
} Node;
Node **hashTable;
int hashTableSize;
hashTableIndex myhash(T data);
Node *insertNode(T data);
void deleteNode(T data);
Node *findNode (T data);
```

Хеширование

Пример. Программная реализация открытого хеширования. С.2.

```
int _tmain(int argc, _TCHAR* argv[]){
  int i, *a, maxnum;
  cout << "Введите количество элементов maxnum : ";
  cin >> maxnum;
  cout << "Введите размер хеш-таблицы HashTableSize : ";
  cin >> hashTableSize;
  a = new int[maxnum];
  hashTable = new Node*[hashTableSize];
  for (i = 0; i < hashTableSize; i++)
    hashTable[i] = NULL;
  // генерация массива
  for (i = 0; i < maxnum; i++)
    a[i] = rand();
  // заполнение хеш-таблицы элементами массива
  for (i = 0; i < maxnum; i++) {
    insertNode(a[i]);
  }
```

Хеширование

Пример. Программная реализация открытого хеширования. С.3.

```
// поиск элементов массива по хеш-таблице
  for (i = maxnum-1; i >= 0; i--) {
    findNode(a[i]);
  }
 // вывод элементов массива в файл List.txt
  ofstream out("List.txt");
  for (i = 0; i < maxnum; i++){
    out << a[i];
    if ( i < maxnum - 1 ) out << "\t";
  }
  out.close();
```

Хеширование

Пример. Программная реализация открытого хеширования. С.4.

```
// сохранение хеш-таблицы в файл HashTable.txt
  out.open("HashTable.txt");
  for (i = 0; i < hashTableSize; i++){
    out << i << "  :  ";
    Node *Temp = hashTable[i];
    while ( Temp ){
      out << Temp->data << " -> ";
      Temp = Temp->next;
    }
    out << endl;
  }
  out.close();
  // очистка хеш-таблицы
    for (i = maxnum-1; i >= 0; i--) {
        deleteNode(a[i]);
    }
  system("pause");
  return 0;
}
```

Хеширование

Пример. Программная реализация открытого хеширования. С.5.

```
// хеш-функция размещения вершины
hashTableIndex myhash(T data) {
  return (data % hashTableSize);
}
// функция поиска местоположения и вставки вершины в таблицу
Node *insertNode(T data) {
  Node *p, *p0;
  hashTableIndex bucket;
  // вставка вершины в начало списка
  bucket = myhash(data);
  if ((p = new Node) == 0) {
    fprintf (stderr, "Нехватка памяти (insertNode)\n");
    exit(1);
  }
  p0 = hashTable[bucket];
  hashTable[bucket] = p;
  p->next = p0;
  p->data = data;
  return p;
}
```

Хеширование

Пример. Программная реализация открытого хеширования. С.6.

```
//функция удаления вершины из таблицы
void deleteNode(T data) {
  Node *p0, *p;
  hashTableIndex bucket;
  p0 = 0;
  bucket = myhash(data);
  p = hashTable[bucket];
  while (p && !compEQ(p->data, data)) {
    p0 = p;
    p = p->next;
  }
  if (!p) return;
  if (p0)
    p0->next = p->next;
  else
    hashTable[bucket] = p->next;
  free (p);
}
```

Хеширование

Пример. Программная реализация открытого хеширования. С.7.

```
// функция поиска вершины со значением data
Node *findNode (T data) {
  Node *p;
  p = hashTable[myhash(data)];
  while (p && !compEQ(p->data, data))
    p = p->next;
  return p;
}
```