# Вариант 1

Хеш-функция должна отображать ключ в целое число из диапазона 0...n-1. При этом количество коллизий должно быть ограниченным, а вычисление самой хеш-функции – очень быстрым. Некоторые методы удовлетворяют этим требованиям.

## метод деления

Наиболее часто используется метод деления(division method), требующий двух шагов. Сперва ключ должен быть преобразован в целое число, а затем полученное значение вписывается в диапазон 0...n-1 с помощью оператора получения остатка. На практике метод деления используется в большинстве приложений, работающих с хешированием.

Предположим, что ключ – пятизначное число. Хеш-функция извлекает две младшие цифры. Например, если это число равно 56389, то HF(56389) = 89. Две младшие цифры являются остатком от деления на 100.

|  |
| --- |
| int HF(int key)  {  return key % 100; // метод деления на 100  } |

Эффективность хеш-функции зависит от того, обеспечивает ли она равномерное распределение ключей в диапазоне 0...n-1. Если две последние цифры соответствуют году рождения, то будет слишком много коллизий при идентификации подростков, играющих в юношеской бейсбольной лиге.

Другой пример – ключ-символьная строка С++. Хеш-функция отображает эту строку в целое число посредством суммирования первого и последнего символов и последующего вычисления остатка от деления на 101 (размер таблицы).

|  |
| --- |
| // хеш-функция для символьной строки.  // Возвращает значение в диапазоне от 0 до 100  int HF(char \*key)  {  int len = strlen(key), hashf = 0;  // если длина ключа равна 0 или 1, возвратить key[0].  // иначе сложить первый и последний символ  if (len <= 1)  hashf = key[0];  else  hashf = key[0] + key[len-1];  return hashf % 101;  } |

Эта хеш-функция приводит к коллизии при одинаковых первом и последнем символах строки. Например, строки «start» и «slant» будут отображаться в индекс 29. Так же ведет себя хеш-функция, суммирующая все символы строки.

|  |
| --- |
| int HF(char \*key)  {  int hashf = 0;  // просуммировать все символы строки и разделить на 101  while (\*key)  hashf += \*key++;  return hashf % 101;  } |

Строки «bad» и «dab» преобразуются в один и тот же индекс. Лучшие результаты дает хеш-функция, производящая перемешивание битов в символах.

В общем случае при больших n индексы имеют больший разброс. Кроме того, математическая теория утверждает, что распределение будет более равномерным, если n – простое число.

**Другие методы хеширования**

## Метод середины квадрата (midsquare technique)

предусматривает преобразование ключа в целое число, возведение его в квадрат и возвращение в качестве значения функции последовательности битов, извлеченных из середины полученного числа. Предположим, что ключ есть целое 32-битное число. Тогда следующая хеш-функция извлекает средние 10 бит возведенного в квадрат ключа.

|  |
| --- |
| // возвратить средние 10 бит произведения key\*key  int HF(int key);  {  key \*= key; // возвести ключ в квадрат  key >>= 11; // отбросить 11 младших бит  return key % 1024 // возвратить 10 младших бит  } |

## При мультипликативном методе (multiplicative method)

используется случайное действительное число f в диапазоне от 0<f<1. Дробная часть произведения f \* key лежит в диапазоне от 0 до 1. Если это произведение умножить на n (размер хеш-таблицы), то целая часть полученного произведения даст значение хеш-функции в диапазоне 0...n-1.

|  |
| --- |
| // хеш-функция, использующая мультипликативный метод;  // возвращает значение в диапазоне 0...700  int HF(int key);  {  static RandomNumber rnd;  float f;  // умножить ключ на случайное число из диапазона 0...1  f = key \* rnd.fRandom();  // взять дробную часть  f = f - int(f);  // возвратить число в диапазоне 0...n-1  return 701\*f;  } |

# Разрешение коллизий

Несмотря на то, что два или более ключей могут хешироваться одинаково, они не могут занимать в хеш-таблице одну и ту же ячейку. Нам остаются два пути: либо найти для нового ключа другую позицию в таблице, либо создать для каждого значения хеш-функции отдельный список, в котором будут все ключи, отображающиеся при хешировании в это значение. Оба варианта представляют собой две классические стратегии разрешения коллизий – **открытую адресацию с линейным перебором** и **метод цепочек**. Мы проиллюстрируем на примере открытую адресацию, а сосредоточимся главным образом на втором методе, поскольку эта стратегия является доминирующей.

## Открытая адресация с линейным перебором

Эта методика предполагает, что каждая ячейка таблицы помечена как незанятая. Поэтому при добавлении нового ключа всегда можно определить, занята ли данная ячейка таблицы или нет. Если да, алгоритм осуществляет перебор по кругу, пока не встретится «открытый адрес» (свободное место). Отсюда и название метода. Если размер таблицы велик относительно числа хранимых там ключей, метод работает хорошо, поскольку хеш-функция будет равномерно распределять ключи по всему диапазону и число коллизий будет минимальным. По мере того как коэффициент заполнения таблицы приближается к 1, эффективность процесса заметно падает.

Проиллюстрируем линейный перебор на примере семи записей.

Предположим, что данные имеют тип DataRecord и хранятся в 11-элементной таблице.

|  |
| --- |
| struct DataRecord  {  int key;  int data;  }; |

В качестве хеш-функции HF используется остаток от деления на 11, принимающий значения в диапазоне 0-10.

|  |
| --- |
| HF(item) = item.key % 11 |

В таблице хранятся следующие данные. Каждый элемент помечен числом проб, необходимых для его размещения в таблице.

|  |
| --- |
| Список: {54,1}, {77,3}, {94,5}, {89,7}, {14,8}, {45,2}, {76,9} |

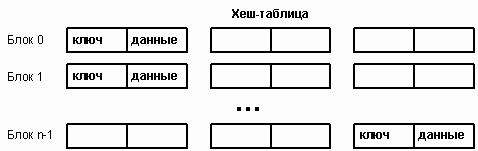
Хеширование первых пяти ключей дает пять различных индексов, по которым эти ключи запоминаются в таблице. Например, HF({54,1}) = 10, и этот элемент попадает в Table[10]. Первая коллизия возникает между ключами 89 и 45, так как оба они отображаются в индекс 1.

Элемент данных {89,7} идет первым в списке и занимает позицию Table[1]. При попытке записать {45,2} оказывается, что место Table[1] уже занято. Тогда начинается последовательный перебор ячеек таблицы с целью нахождения свободного места. В данном случае это Table[2]. На ключе 76 эффективность алгоритма сильно падает. Этот ключ хешируется в индекс 10 – место, уже занятое. В процессе перебора осуществляется просмотр еще пяти ячеек, прежде чем будет найдено свободное место в Table[4]. Общее число проб для размещения в таблице всех элементов списка равно 13, т.е. в среднем 1,9 проб на элемент.

## Метод цепочек

При другом подходе к хешированию таблица рассматривается как массив связанных списков или деревьев. Каждый такой список называется **блоком**(bucket) и содержит записи, отображаемые хеш-функцией в один и тот же табличный адрес. Эта стратегия разрешения коллизий называется **методом цепочек (chaining with separate lists)**.

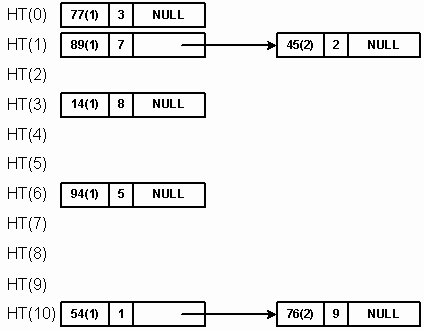
Если таблица является массивом связанных списков, то элемент данных просто вставляется в соответствующий список в качестве нового узла. Чтобы обнаружить элемент данных, нужно применить хеш-функцию для определения нужного связанного списка и выполнить там последовательный поиск.

  
*Рис. 29.*

Проиллюстрируем метод цепочек на семи записях типа DataRecord и хеш-функции HF.

|  |
| --- |
| Список: {54,1}, {77,3}, {94,5}, {89,7}, {14,8}, {45,2}, {76,9}  HF(item) = item.key % 11 |

Каждый новый элемент данных вставляется в хвост соответствующего связанного списка. На рисунке 30 каждое значение данных сопровождается (в скобках) числом проб, требуемых для запоминания этого значения в таблице.

  
*Рис. 30.*

Заметьте, что если считать пробой вставку нового узла, то их общее число при вставке семи элементов равно 9, т.е. в среднем 1,3 пробы на элемент данных.

В общем случае метод цепочек быстрее открытой адресации, так как просматривает только те ключи, которые попадают в один и тот же табличный адрес. Кроме того, открытая адресация предполагает наличие таблицы фиксированного размера, в то время как в методе цепочек элементы таблицы создаются динамически, а длина списка ограничена лишь количеством памяти. Основным недостатком метода цепочек являются дополнительные затраты памяти на поля указателей. В общем случае динамическая структура метода цепочек более предпочтительна для хеширования.

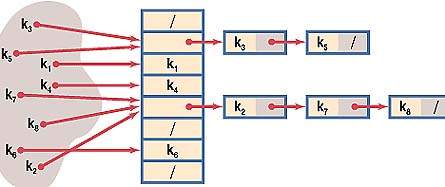
# Вариант 2

## Схемы хеширования

Традиционно принято выделять две схемы хеширования:

* хеширование с цепочками (со списками);
* хеширование с открытой адресацией.

В первом случае выбирается некая хеш-функция h(k) = i, где i трактуется как индекс в таблице списков t. Поскольку нельзя гарантировать, что не встретится двух разных ключей, которым соответствует один и тот же индекс i (конфликт, коллизия), такие «однородные» ключи просто помещаются в список, начинающийся в i-ячейке хеш-таблицы t (см. рисунок). Очевидно, что процесс заполнения хеш-таблицы будет достаточно простым, но при этом доступ к элементам потребует двух операций: вычисления индекса и поиска в соответствующем списке. Операции по занесению и поиску элементов при таком виде хеширования будут вестись в незамкнутом (открытом) пространстве памяти.



**Отображение ключей путем хеширования с цепочками**

Во втором случае все операции производятся в одном измерении, и таблица t является обычным одномерным массивом. Однако в этом случае подход к разрешению коллизий индексов иной: либо элементы с «однородными» ключами пытаются размещать в непосредственной близости от полученного индекса, либо осуществляют многократное хеширование (обычно двойное), когда для хорошего перемешивания последовательно применяется набор разных (взаимосвязанных) хеш-функций. Очевидно, что здесь и заполнение хеш-таблицы, и доступ к элементам будет весьма замысловатым. Хеш-адрес элемента с данным ключом как бы открыт: он постепенно уточняется. При этом все операции ведутся в замкнутом пространстве (в одномерном массиве).

Как нетрудно заметить, традиционная классификация методов хеширования может быть заменена другими: хеширование с цепочками можно относить по классификации Ахо, Хопкрофта и Ульмана [6] к так называемому «открытому» хешированию (в данной статье — к многомерному хешированию, поскольку при использовании списков речь идет о нескольких измерениях), а хеширование с открытой адресацией — к «закрытому» хешированию (одномерному).

## Выбор хеш-функции

— непростая задача. Такая функция должна удовлетворять двум требованиям: предусматривать быстрое вычисление и минимизировать количество коллизий. Самый простой вид хеш-функции: h(k) = k MOD M, где k — ключ-число, M — размер хеш-таблицы (простое число), MOD — остаток от целочисленного деления. Если ключ k — составной (состоит из нескольких слов/символов x1 ... xs), можно воспользоваться идеей Дж. Картера и М. Вегмана (1977): h(k) = (h1(x1) + h2(x2) + ... + hs(xs)) MOD M. На практике для выбора хеш-функции применяются различные эвристические подходы, учитывающие специфику задач.

Вопросы вычисления и уточнения хеш-адреса с алгоритмической точки зрения являются ключевыми, и на них мы остановимся при разборе схем хеширования.

### Одномерное хеширование

В одномерном хешировании (открытая адресация) можно выделить два основных метода:

* линейное исследование — В.Петерсон (1957), А.П.Ершов (1957);
* двойное хеширование — Гюи де Бальбин (1968), Г.Кнотт (1968), Белл, Каман (1970), Р.Брент (1973).

Идея метода линейного исследования состоит в том, чтобы в случае коллизии просматривать соседние ячейки таблицы размером M до тех пор, пока не будет найден искомый ключ k или же пустая позиция. Обычно просмотр ведется в виде последовательности проб: h(k), h(k)–1, h(k)–2, ..., 0, M–1, ... h(k)+1. Чтобы избежать эффекта скучивания, шаг просмотра можно выбирать не равным 1, а в виде числа, взаимно простого с M. Это приводит к идее квадратичного исследования. Здесь для i-й пробы h(k,i) = (h(k) + c1xi + c2xi2) MOD M.

В случае линейного исследования нужно быть крайне осторожным с реализацией функции удаления (Del): можно потерять другой ключ. По этой причине прибегают к приему логического удаления: ячейку в таблице помечают соответствующим признаком. Таким образом, ячейки становятся трех видов: пустые, занятые и удаленные. Стоит заметить, что среднее число проб при успешном поиске с помощью этого метода зависит не от порядка вставки ключей, а только от числа ключей с конкретным хеш-адресом. Алгоритм хорошо работает в начале заполнения таблицы, но затем все чаще встречаются длинные серии проб. И все же он достаточно прост и эффективен: при заполнении таблицы на 90% для поиска элемента в среднем требуется около 5,5 пробы.

Двойное хеширование на этапе уточнения хеш-адреса использует не просмотр, а вычисление значения другой хеш-функции, т. е. применяет h1(k) и h2(k). Значения h1(k) должны опять-таки лежать в диапазоне 0...M–1, а вот функция h2(k) должна порождать значения от 1 до M–1, причем взаимно простые с M (если M — простое число, то h2(k) — любое в указанном диапазоне, а если M = 2p , то h2(k) — нечетное). Если число занятых ячеек обозначить N, то среднее количество проб в этом алгоритме будет составлять (M+1) / (M–N+1).

### Многомерное хеширование

Схема многомерного хеширования (метод цепочек, Ф. Уильямс, 1959) довольно проста: в случае возникновения коллизий после вычисления хеш-функции ключи с одним хеш-адресом соединяются в цепочку. Здесь приходится решать такую проблему, как обеспечение равномерности заполнения хеш-таблицы. К тому же было бы неплохо, если бы она оказалась достаточно сбалансированной, чтобы содержать предельно короткие цепочки. Интересно, что если таблица будет заполнена наполовину, среднее число проб при неудачном поиске составит 1,18. Если таблица будет заполнена полностью, то для нахождения элемента потребуется в среднем 1,8 пробы. Метод цепочек экономичен с точки зрения проб, но неэффективно расходует память. Интересный обзор эффективности методов вместе с демонстрационным Java-аплетом приведен в курсе канадского университета McGill по алгоритмам и структурам данных ([*www.cs.mcgill.ca/~cs251/OldCourses/ 1997/topic12*](http://www.cs.mcgill.ca/~cs251/OldCourses/1997/topic12)).

Во всех наших рассуждениях следует, однако, иметь в виду, что среднее время, основанное на теории вероятностей, может значительно отличаться от реального в каждом конкретном случае. Так что несмотря на стройные математические выкладки, жизнь вносит свои существенные коррективы. Современные процессоры работают при тактовой частоте порядка 1 ГГц, соответственно каждый такт занимает 1 нс. Время доступа к оперативной памяти составляет 7—10 нс. Отсюда следует, что на каждое обращение к памяти времени требуется в 7—10 раз больше, чем на обработку инструкции процессора. При наличии кэш-памяти проблема частично решается, однако в общем случае такой разрыв на порядок будет сохраняться. Для хеширования желательно, чтобы элементы хранились как можно более компактно и ближе друг к другу, а это возможно как раз при одномерном хешировании.

### Области применения и другие методы

Одно из побочных применений хеширования состоит в том, что оно создает своего рода слепок, «отпечаток пальца» для сообщения, текстовой строки, области памяти и т. п. Такой «отпечаток пальца» может стремиться как к «уникальности», так и к «похожести» (яркий пример слепка — контрольная сумма CRC). В этом качестве одной из важнейших областей применения является криптография. Здесь требования к хеш-функциям имеют свои особенности. Помимо скорости вычисления хеш-функции требуется значительно осложнить восстановление сообщения (ключа) по хеш-адресу. Соответственно необходимо затруднить нахождение другого сообщения с тем же хеш-адресом. При построении хеш-функции однонаправленного характера обычно используют функцию сжатия (выдает значение длины n при входных данных больше длины m и работает с несколькими входными блоками). При хешировании учитывается длина сообщения, чтобы исключить проблему появления одинаковых хеш-адресов для сообщений разной длины. Наибольшую известность имеют следующие хеш-функции [7]: MD4, MD5, RIPEMD-128 (128 бит), RIPEMD-160, SHA (160 бит). В российском стандарте цифровой подписи используется разработанная отечественными криптографами хеш-функция (256 бит) стандарта ГОСТ Р 34.11—94.

Хеширование можно рассматривать и как расстановку, и как снятие отпечатков, и как раскрашивание. В самом деле, то, что каждому ключу ставится в соответствие целое число, дает основание говорить о хеш-адресе как о хеш-краске.

Многомерное хеширование отражает известный принцип Дирихле: при любом размещении (n+1) предметов по «n» ящикам всегда найдется ящик с двумя предметами. В случае, если нас интересует наличие k предметов в одном ящике, он формулируется так: при любом размещении (rxk – r + 1) предметов по r ящикам найдется k предметов в одном ящике. Несмотря на свою тривиальность, принцип Дирихле весьма полезен — на нем основаны многие теоремы математики фундаментального характера (теоремы Матиясевича, Дирихле, Эйлера – Ферма, Рамсея). Если большая структура разбивается на непересекающиеся части, то наличие какой подструктуры можно гарантировать в одной из частей? И обратная задача: сколь богатой должна быть большая структура, чтобы любое ее разбиение содержало часть предписанной природы? Здесь разбиение множества ключей на цепочки («ящики») приводит к смежным задачам, в частности, к теории Рамсея — специальной ветви комбинаторики, которая имеет дело со структурами, сохраняющимися под действием разбиений.