Хеширование.

С хешированием мы сталкиваемся едва ли не на каждом шагу: при работе с браузером (список Web-ссылок), текстовым редактором и переводчиком (словарь), языками скриптов (Perl, Python, PHP и др.), компилятором (таблица символов). По словам Брайана Кернигана, это «одно из величайших изобретений информатики». Заглядывая в адресную книгу, энциклопедию, алфавитный указатель, мы даже не задумываемся, что упорядочение по алфавиту является не чем иным, как хешированием.

Хеширование есть разбиение множества ключей (однозначно характеризующих элементы хранения и представленных, как правило, в виде текстовых строк или чисел) на непересекающиеся подмножества (наборы элементов), обладающие определенным свойством. Это свойство описывается функцией хеширования, или **хеш-функцией**, и называется **хеш-адресом**. Решение обратной задачи возложено на **хеш-структуры (хеш-таблицы)**: по хеш-адресу они обеспечивают быстрый доступ к нужному элементу. В идеале для задач поиска хеш-адрес должен быть уникальным, чтобы за одно обращение получить доступ к элементу, характеризуемому заданным ключом (совершенная хеш-функция). Однако на практике идеал приходится заменять компромиссом и исходить из того, что получающиеся наборы с одинаковым хеш-адресом содержат более одного элемента.

Термин «хеширование» (hashing) в печатных работах по программированию появился сравнительно недавно (1967 г. [1]), хотя сам механизм был известен и ранее. Глагол «hash» в английском языке означает «рубить, крошить», т. е. создавать этакий «винегрет». Для русского языка академиком А.П. Ершовым [2] был предложен достаточно удачный эквивалент — «расстановка», созвучный с родственными понятиями комбинаторики, такими как «подстановка» и «перестановка». Однако пока он не прижился.

Как отмечает Дональд Кнут [3], идея хеширования впервые была высказана Г.П. Ланом при создании внутреннего меморандума IBM в январе 1953 г. с предложением использовать для разрешения коллизий хеш-адресов метод цепочек. В открытой печати хеширование впервые было описано Арнольдом Думи (1956), указавшим, что в качестве хеш-адреса удобно использовать остаток от деления на простое число. Подход к хешированию, отличный от метода цепочек, был предложен А.П. Ершовым (1957), который разработал и описал метод линейной открытой адресации. Среди других исследований можно отметить работы Петерсона (1957, [4]) и Морриса (1968, [5]). В первой реализовывался класс методов с открытой адресацией при работе с большими файлами, а во второй давался обширный обзор по хешированию и вводился термин «рассеянная память» (scatter storage).

Массивы — предшественники хеш-структур

Одна из важных задач, решаемых в программировании,— это обеспечение быстрого (прямого) доступа к данным по некоему коду (индексу, адресу). Неудивительно, что решающий эту задачу массив стал одним из главных строительных блоков, превосходя по использованию списки, которые определяют последовательный доступ к элементам. В математике массиву соответствуют понятия вектор (в одномерном случае) и матрица (в двумерном).

Как известно, массив задает отображение (A) множества индексов (I) на множество элементов (E), т. е. A: I —> E. Массив позволяет по индексу быстро найти требуемый элемент. Хеширование решает в точности такую же задачу. Однако здесь уже в роли индекса выступает хеш-адрес, который определяется как значение некоей хеш-функции, применяемой к уникальному ключу. В этом смысле хеш-структуры можно рассматривать как обобщение массива.

В программировании зависимость между индексом и значением записывается в виде: A = ARRAY I OF E. В роли индексирующего типа (I) обычно выбирается конкретный диапазон значений из целочисленного типа (хотя в общем случае в их роли могут выступать так называемые скалярные типы, т. е. булев тип, перечисления, множества и др.). Ну а элементы массива в зависимости от языка программирования могут быть любыми, начиная от битов, чисел и указателей (ссылок) и заканчивая составными типами произвольной глубины.

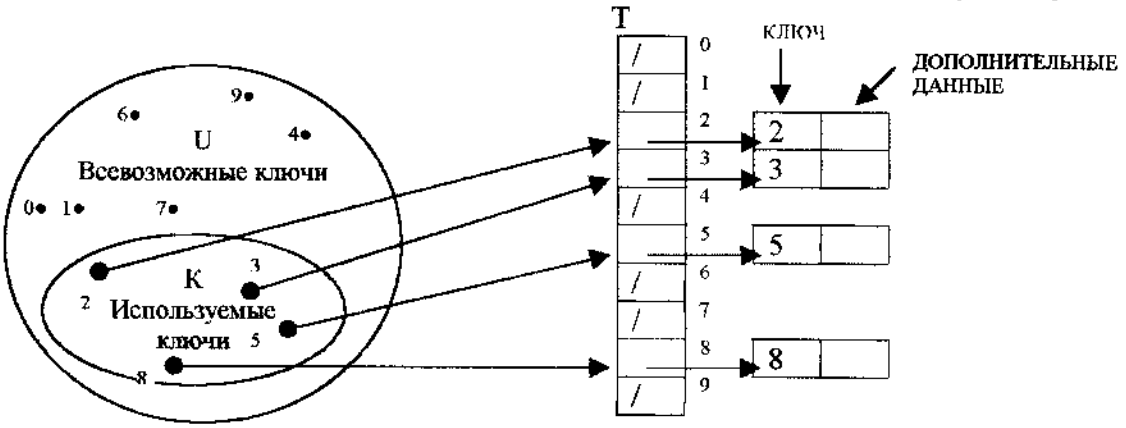
Выделяют два разных вида массивов: одномерные (наиболее общий случай) и многомерные (на каждом слое адресации используется массив фиксированной структуры). Во втором случае есть и особый подвид: ступенчатые массивы (jagged arrays). Они встречаются, в частности, в языке C# в том случае, когда на каждом слое адресации используется массив переменной структуры. Иначе говоря, здесь мы имеем дело с массивом разных массивов. В других языках такая конструкция легко описывается массивом разнородных указателей (каждый указывает на массив своей структуры), что фактически определяет массив списков.

Интересно, что Н. Вирт после многих лет использования в своих языках (Паскаль, Модула-2) в качестве индексирующего типа разных скалярных типов пришел к выводу, что лаконичное решение, воплощенное в языке Си (а точнее, унаследованное в Си от языков BCPL и B), носит куда более практичный характер. И в своих новых языках Оберон и Оберон-2 он отказался от идей Паскаля и ограничился заданием размера массива (количества индексируемых элементов), т. е. определением для индексов диапазона 0...n—1, где n — это размер массива: A = ARRAY 16 OF E. Связано это с эффективностью реализации и с активным использованием в программировании элементов модулярной арифметики. В Обероне предопределенная функция MOD («x MOD n»), как и в математике, соответствует остатку от целочисленного деления «x» на «n». Как показывает опыт, использование 0 в качестве начального индекса удобно в подавляющем большинстве задач. Механизмы хеширования опираются точно на ту же основу.

Прямая адресация

Рассмотрим организацию адресации, которая, в частом случае, реализована как организация обращения к элементам массива. Прямая адресация применима, если количество возможных ключей невелико. Пусть возможными ключами являются числа из множества U={0,1,...,m-1} (число m не очень велико). Предположим также, что ключи всех элементов различны.

Для хранения множества мы используем массив T[0..m-1], называемым **таблицей с прямой адресацией.** Каждая **позиция**, или **ячейка,** соответствует определённому ключу из множества U.



Здесь T[к] – место для записи указателя на элемент с ключом к; если элемента с таким ключом нет , то T[к] = ПУСТО. Реализация словарных операций тривиальна:

НАЙТИ_ПРЯМАЯ_АДРЕСАЦИЯ(T, к)
вернуть T[к]

ДОБАВИТЬ_ПРЯМАЯ_АДРЕСАЦИЯ(T, x)
T[КЛЮЧ[x]] ← x

УДАЛИТЬ_ПРЯМАЯ_АДРЕСАЦИЯ(T, к)
T[КЛЮЧ[x]] ← ПУСТО

Каждая из этих операций требует времени O(1).

Иногда можно сэкономить место, записывая в таблицу T не указатели на элементы множества, а сами эти элементы. Можно обойтись и без отдельного поля КЛЮЧ: ключом служит

индекс в массиве. Впрочем, если мы обходимся без ключей и указателей, то надо иметь способ указать, что данная позиция свободна.

Хеш-таблицы

Часто бывают нужны динамические множества, поддерживающие только словарные операции (добавление, поиск и удаление элемента). В этом случае часто применяют так называемое хеширование; соответствующая структура данных называется «хеш-таблица» (или «таблица расстановки»). В худшем случае поиск в хеш-таблице может занимать столько же времени, сколько поиск в списке (O(n)), но на практике хеширование весьма эффективно. При выполнении некоторых естественных условий математическое ожидание времени поиска элемента в хеш-таблице есть O(1).

Хеш-таблицу можно рассматривать как обобщение обычного массива. Если у нас достаточно памяти для массива, число элементов которого равно числе всех возможных ключей, для каждого возможного ключа можно отвести ячейку за время O(1). Однако если реальное количество записей значительно меньше, чем количество возможных ключей, то эффективнее применять хеширование: вычислять позицию записи в массиве, исходя из ключа.

Хеш-таблица

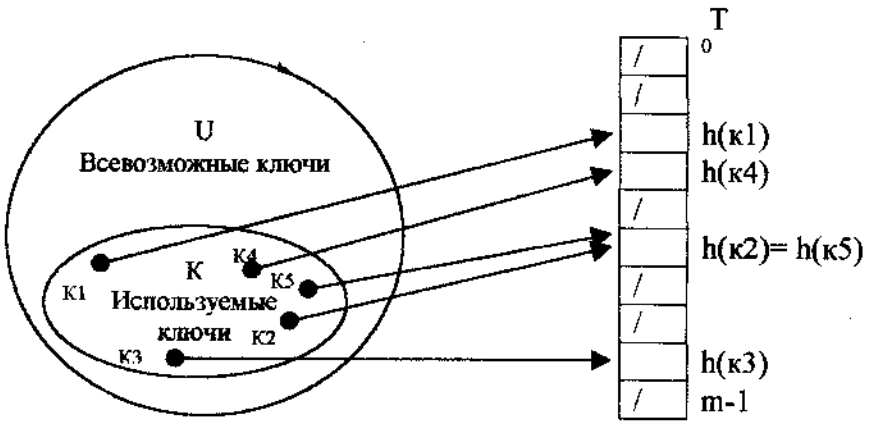
Прямая адресация обладает очевидным недостатком: если множество U всевозможных ключей велико, то хранить в памяти массив T размером |U|, то много памяти тратится зря.

Если количество записей в таблице существенно меньше, чем количество всевозможных ключей, то хеш-таблица занимает гораздо меньше места, чем таблица с прямой адресацией. Именно, хеш-таблица требует памяти объёмом O(|K|), где K – множество записей, при этом время поиска в хеш-таблице по-прежнему есть O(1) (в среднем).

В то время как при прямой адресации элементу с ключом k отводится позиция номер k, при хешировании этот элемент записывается в позицию номер h(k) в **хеш-таблице** T[0..m-1], где

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

- некоторая функция, называемая **хеш-функцией**. Число h(k) называется **хеш-значением** ключа k. Пользуясь массивом длины m, а не |U|, мы экономим память.



Хеш-функции

Идеальной хеш-функцией является такая hash-функция, которая для любых двух неодинаковых ключей дает неодинаковые адреса.

$$k1 \neq k2 \Rightarrow h(k1) \neq h(k2)$$

Подобрать такую функцию можно в случае, если все возможные значения ключей заранее известны. Такая организация данных носит название "совершенное хеширование". В случае заранее неопределенного множества значений ключей и ограниченной длины таблицы подбор совершенной функции затруднителен. Поэтому часто используют хеш-функции, которые не гарантируют выполнение условия.

На практике при выборе хеш-функций пользуются различными эвристиками, основанными на специфике задачи. Например, компилятор языка программирования хранит таблицу символов, в которой ключами являются идентификаторы программы. Часто в программе используются несколько похожих идентификаторов (например *pt* и pts). Хорошая хеш-функция будет стараться, чтобы хеш-значения у таких похожих идентификаторов были различны.

Обычно предполагают, что область определения хеш-функции – множество целых неотрицательных чисел. Если ключи не являются натуральными числами, их можно преобразовать к такому виду (хотя числа могут получиться большими). Например последовательность символов можно интерпретировать как числа, записанные в системе счисления с подходящим основанием: идентификатор pt – это пара чисел (112, 116) (таковы ASCII-коды букв p и t), или же число (112*128)+116=14452 (в системе счисления по основанию 128).

Деление с остатком

Построение хеш-функции методом деления с остатком состоит в том, что ключу k ставится в соответствие остаток от деления k на m, где m – число возможных хеш-значений:

$h(k) = k \bmod m$

Например, если размер хеш-таблицы равен m=12 и ключ равен 100, то хеш-значение равно 4.

Умножение

Построение хеш-функции методом умножения состоит в следующем. Пусть количество хеш-значений равно m. Зафиксируем константу M на интервале 0<M<1, и положим

$h(k) = \lfloor m\,(kM \bmod 1) \rfloor$,

где $kM \bmod 1$ – дробная часть kM.

Достоинство метода умножения в том, что он мало зависит от выбора m. Обычно в качестве m выбирают степень двойки, поскольку во многих компьютерах это реализуется как сдвиг слова. Метод умножения работает при любом выборе константы M, но некоторые значения M могут быть лучше других. Оптимальный выбор зависит от того, какого рода данные подвергаются хешированию. Кнут в своей книге приходит к выводу, что значение

$M \approx (\sqrt{5} - 1) / 2 = 0{,}6180339887...$

является довольно удачным. Если k = 123456, m = 10000 , то h(k) = 41

Универсальное хеширование

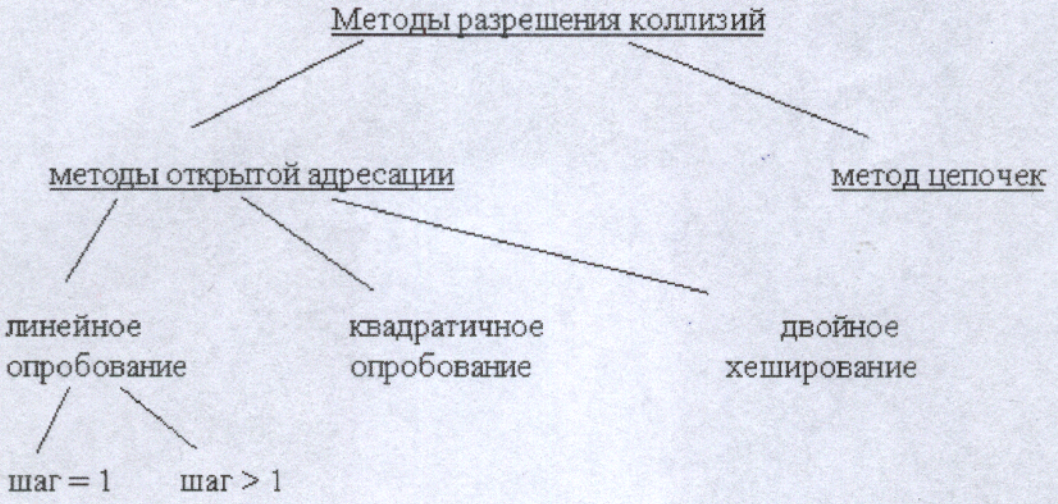
Если недоброжелатель будет специально подбирать данные для хеширования, то (зная функцию h(k)) он может устроить так, что все n ключей будут соответствовать одной позиции в таблице, в результате чего время поиска будет равно O(n). Любая фиксированная хеш-функция может быть дискредитирована таким образом. Единственный выход из положения – выбирать хеш-функцию случайным образом, не зависящим от того, какие именно данные вы хешируете. Такой подход называется универсальным хешированием.

Основная идея универсального хеширования – выбирать хеш-функцию во время исполнения программы случайным образом из некоторого множества. Стало быть, при повторном вызове с теми же входными данными алгоритм будет работать уже по-другому.

Коллизии и их разрешение.

Проблема в том, что хеш-значения двух разных ключей могут совпасть. В таких случаях говорят, что случилась коллизия, или столкновение. Но хеш-функциями можно пользоваться и при наличии столкновений.

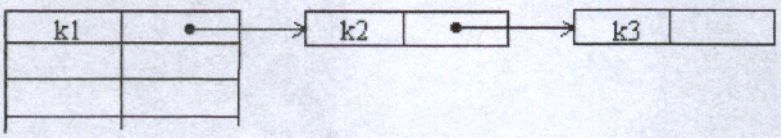
Для разрешения коллизий используются различные методы, которые в основном сводятся к методам “цепочек“ и “открытой адресации“.



Необходимо выбрать хеш-функцию так, чтобы коллизии были невозможны. Но при |U|> m неизбежно существуют разные ключи, имеющие одно и то же хеш-значение. Рассмотрим простейший способ обработки коллизий с помощью цепочек.

Разрешение коллизий с помощью цепочек - Ф. Уильямс, 1959

Методом цепочек называется метод, в котором для разрешения коллизий во все записи вводятся указатели, используемые для организации списков – “цепочек переполнения”. В случае возникновения коллизии при заполнении таблицы в список для требуемого адреса хеш-таблицы добавляется еще один элемент. Технология сцепления элементов состоит в том, что элементы множества, которым соответствует одно и то же хеш-значение, связываются в цепочку-список. В позиции номер j хранится указатель на голову списка тех элементов, у которых хеш-значение ключа равно j; если таких элементов в множестве нет, в позиции j записан ПУСТО.



$k1 \neq k2$
$h(k1)=h(k2)$
$k3 \neq k1$
$h(k3)=h(k1)$

Поиск в хеш-таблице с цепочками переполнения осуществляется следующим образом. Сначала вычисляется адрес по значению ключа. Затем осуществляется последовательный поиск в списке, связанном с вычисленным адресом.

Процедура удаления из таблицы сводится к поиску элемента и его удалению из цепочки переполнения.

Операции добавления, поиска и удаления реализуются легко:

ДОБАВИТЬ_ХЕШ-ТАБЛИЦА(T, x)
добавить x в голову списка T[h(КЛЮЧ[x])]

НАЙТИ_ХЕШ-ТАБЛИЦА(T, к)
найти элемент с ключом к в списке T[h(к)]

УДАЛИТЬ_ХЕШ-ТАБЛИЦА(T, x)
удалить x из списка T[h(КЛЮЧ[x])]

Операция добавления работает в худшем случае за время O(1). Максимальное время работы операции поиска пропорционально длине списка. Удаление элемента можно провести за время O(1) – при условии, что списки двусторонне связаны.

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).

Открытая адресация

В отличие от хеширования с цепочками, при открытой адресации никаких списков нет, а все записи хранятся в самой хеш-таблице, каждая ячейка таблицы содержит либо элемент динамического множества, либо ПУСТО. Поиск заключается в том, что мы определённым образом просматриваем элементы таблицы, пока не найдём то, что ищем, или не удостоверимся, что элемента с таким ключом в таблице нет. Тем самым число хранимых элементов не может быть больше размера таблицы.

Чтобы добавить новый элемент в таблицу с открытой адресацией, ячейки которой занумерованы целыми числами от 0 до m-1, мы просматриваем её, пока не найдём свободное место. Если просматривать ячейки таблицы каждый раз подряд, потребуется время O(n), но суть в том, что порядок просмотра таблицы зависит от ключа. Мы добавляем к хеш-функции второй аргумент – номер попытки (нумерацию начинаем с 0), так что хеш-функция имеет вид:

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

(U – множество ключей). Последовательность испробованных мест, или последовательность проб, для данного ключа к имеет вид

$\langle h(к, 0), h(к, 1), \ldots, h(к, m-1)\rangle$;

функция h должна быть такой, чтобы каждое из чисел от 0 до m-1 встретилось в этой последовательности ровно один раз (для каждого ключа все позиции должны быть доступны).

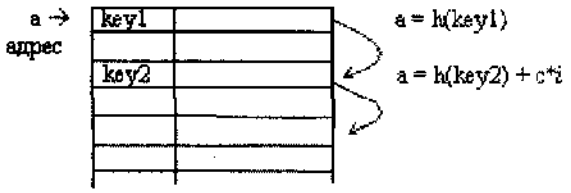
ДОБАВИТЬ_ОТКРЫТАЯ_АДРЕСАЦИЯ(T, к)
i ← 0
повторить j ← h(k,i)
если T[j] = ПУСТО
тогда T[j] ← k
вернуть j
иначе i ← i + 1
до тех пор пока i = m
вывести «переполнение хеш-таблицы»

ПОИСК_ОТКРЫТАЯ_АДРЕСАЦИЯ(T, к)
i ← 0
    повторить j ← h(k,i)
        если T[j] = к
            тогда вернуть j
            i ← i + 1
    до тех пор пока T[j] = ПУСТО или i = m
вернуть ПУСТО

Удалить элемент из таблицы с открытой адресацией не так просто. Записывать на место удалённого элемента ПУСТО нельзя, так как не сможем найти те элементы, в момент добавления которых в таблицу это место было занято. Возможно решение – записывать на место удалённого элемента не ПУСТО, а специальное значение УДАЛЁН, и при добавлении рассматривать ячейку с записью УДАЛЁН как свободную, а при поиске, как занятую.

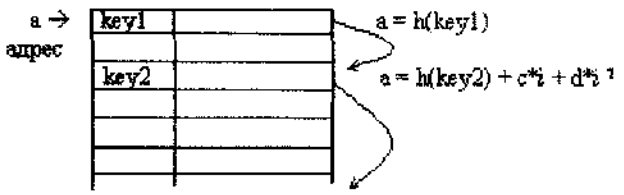
Для вычисления последовательности испробованных мест применяют один из трёх способов: линейное, квадратичное и двойное хеширование.

а) Линейное опробование



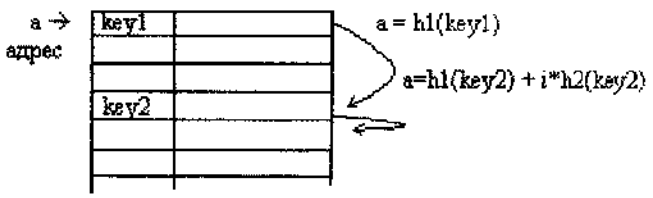
key1 ≠ key2
a=h(key1) = h(key2)

б) Квадратичное опробование



key1 ≠ key2
a=h(key1) = h(key2)

в) Двойное хеширование



key1 ≠ key2
a=h1(key1) = h1(key2)

Разрешение коллизий при добавлении элементов методами открытой адресации.

Линейная последовательность проб— В.Петерсон (1957), А.П.Ершов (1957).

Пусть h':U→{0,1,…,m-1} – обычная хеш-функция. Функция, определяющая линейную последовательность проб, задаётся формулой

$h(k,i) = (h'(k) + i) \bmod m$

Открытую адресацию с линейной последовательностью проб легко реализовать, но у этого метода есть один недостаток: он может привести к образованию кластеров, то есть длинных последовательностей занятых ячеек, идущих подряд. Это удлиняет поиск.

Квадратичная последовательность проб

Функция определяющая квадратичную последовательность проб, задаётся формулой

$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$

где, по-прежнему, h' – обычная хеш-функция, а c_1 и $c_2 \neq 0$ – некоторые константы.

Тенденции к образованию кластеров больше нет, но аналогичный эффект появляется в (более мягкой) форме образования вторичных кластеров.

Двойное хеширование - Гюи де Бальбин (1968), Г.Кнотт (1968), Белл, Каман (1970), Р.Брент (1973).

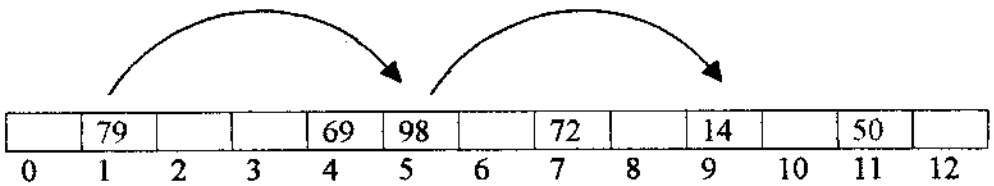
Двойное хеширование – один из лучших методов открытой адресации. При двойном хешировании функция h имеет вид:

$h(k,i) = (h_1(k) + i\, h_2(k)) \bmod m$

где h_1 и h_2 – обычные хеш-функции. Иными словами, последовательность проб при работе с ключом k представляет собой арифметическую прогрессию (по модулю m) с первым членом $h_1(k)$ и шагом $h_2(k)$.

Чтобы последовательность испробованных мест покрыла всю таблицу, значение $h_2(k)$ должно быть взаимно простым с m. Простой способ добиться этого условия – выбрать в качестве m степень двойки, а функцию h_2 взять такую, чтобы она принимала только нечётные значения.

Пример, добавление элемента в таблицу с открытой адресацией при двойном хешировании. В нашем случае $m = 13$, $h_1(k) = k \bmod 13$, $h_2(k) = 1 + (k \bmod 11)$. Если $k = 14$, то последовательность проб будет такая: 1 и 5 заняты, 9 свободно, помещаем туда.



Опишем алгоритмы вставки и поиска для метода линейное опробование.

Вставка

1. i = 0
2. a = h(key) + i*c
3. **если** t(a) = свободно **то** t(a) = key, *записать элемент, стоп элемент добавлен*
4. i = i + 1. Перейти к шагу 2

Поиск

1. i = 0
2. a = h(key) + i*c
3. **если** t(a) = key **то** стоп элемент найден
4. **если** t(a) = свободно **то** стоп элемент не найден
5. i = i + 1. Перейти к шагу 2

Аналогичным образом можно было бы сформулировать алгоритмы добавления и поиска элементов для любой схемы открытой адресации. Отличия будут только в выражении, используемом для вычисления адреса (шаг 2). С процедурой удаления дело обстоит не так просто, так как она в данном случае не будет являться обратной процедуре вставки.

Дело в том, что элементы таблицы находятся в двух состояниях: свободно и занято. Если удалить элемент, переведя его в состояние свободно, то после такого удаления алгоритм поиска будет работать некорректно. Предположим, что ключ удаляемого элемента имеет в таблице ключи синонимы. В том случае, если за удаляемым элементом в результате разрешения коллизий были размещены элементы с другими ключами, то поиск этих элементов после удаления всегда будет давать отрицательный результат, так как алгоритм поиска останавливается на первом элементе, находящемся в состоянии свободно.

Скорректировать эту ситуацию можно различными способами. Самый простой из них заключается в том, чтобы производить поиск элемента не до первого свободного места, а до конца таблицы. Однако такая модификация алгоритма сведет на нет весь выигрыш в ускорении доступа к данным, который достигается в результате хеширования.

Другой способ сводится к тому, чтобы проследить адреса всех ключей-синонимов для ключа удаляемого элемента и при необходимости пере разместить соответствующие записи в таблице. Скорость поиска после такой операции не уменьшится, но затраты времени на само пере размещение элементов могут оказаться очень значительными.

Существует подход, который свободен от перечисленных недостатков. Его суть состоит в том, что для элементов хеш-таблицы добавляется состояние "удалено". Данное состояние в процессе поиска интерпретируется, как занято, а в процессе записи как свободно.

Сформулируем алгоритмы вставки поиска и удаления для хеш-таблицы, имеющей три состояния элементов.

Вставка

1. i = 0
2. a = h(key) + i*c
3. **если** t(a) = свободно или t(a) = удалено, **то** t(a) = key, записать элемент, *стоп элемент добавлен*
4. i = i + 1
5. **идти** к шагу 2

Удаление

1. i = 0
2. a = h(key) + i*c
3. **если** t(a) = key, **то** t(a) =удалено, *стоп элемент удален*
4. **если** t(a) = свободно, **то** *стоп элемент не найден*
5. i = i + 1
6. **идти** к шагу 2

Поиск

1. i = 0
2. a = h(key) + i*c
3. **если** t(a) = key, **то** *стоп элемент найден*
4. **если** t(a) = свободно, **то** *стоп элемент не найден*
5. i = i + 1,
6. **идти** к шагу 2

Алгоритм поиска для хеш-таблицы, имеющей три состояния, практически не отличается от алгоритма поиска без учета удалений. Разница заключается в том, что при организации самой таблицы необходимо отмечать свободные и удаленные элементы. Это можно сделать, зарезервировав два значения ключевого поля. Другой вариант реализации может предусматривать введение дополнительного поля, в котором фиксируется состояние элемента. Длина такого поля может составлять всего два бита, что вполне достаточно для фиксации одного из трех состояний. На языке TurboPascal данное поле удобно описать типом Byte или Char.

Переполнение таблицы и рехеширование

Очевидно, что по мере заполнения хеш-таблицы будут происходить коллизии и в результате их разрешения методами открытой адресации очередной адрес может выйти за пределы адресного пространства таблицы. Что бы это явление происходило реже, можно пойти на увеличение длины таблицы по сравнению с диапазоном адресов, выдаваемым хеш-функцией.

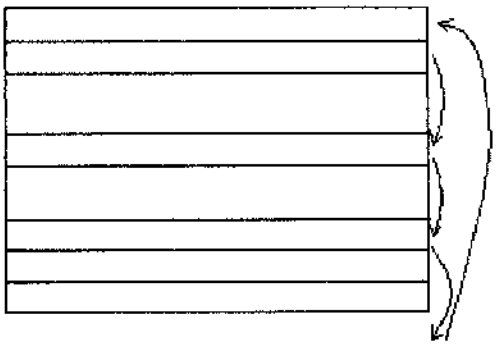


Рис.3.5. Циклический переход к началу таблицы.

С одной стороны это приведет к сокращению числа коллизий и ускорению работы с хеш-таблицей, а с другой – к нерациональному расходованию адресного пространства. 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 элементов. Поэтому на практике используют циклический переход к началу таблицы.

Рассмотрим данный способ на примере метода линейного опробования. При вычислении адреса очередного элемента можно ограничить адрес, взяв в качестве такового остаток от целочисленного деления адреса на длину таблицы n.

Вставка

1. i = 0
2. a = (h(key) + c*i) mod n
3. **если** t(a) = свободно или t(a) = удалено, **то** t(a) = key, записать элемент, *стоп элемент добавлен*
4. i = i + 1
5. **идти** к шагу 2

В данном алгоритме мы не учитываем возможность многократного превышения адресного пространства. Более корректным будет алгоритм, использующий сдвиг адреса на 1 элемент в случае каждого повторного превышения адресного пространства. Это повышает вероятность найти свободные элементы в случае повторных циклических переходов к началу таблицы.

Вставка

1. i = 0
2. a = ((h(key) + c*i) div n + (h(key) + c*i) mod n) mod n
3. **если** t(a) = свободно или t(a) = удалено, **то** t(a) = key, записать элемент, *стоп элемент добавлен*
4. i = i + 1
5. **идти** к шагу 2
6. a = ((h(key) + c*i) div n + (h(key) + c*i) mod n) mod n

Рассматривая возможность выхода за пределы адресного пространства таблицы, мы не учитывали факторы заполненности таблицы и удачного выбора хеш-функции. При большой заполненности таблицы возникают частые коллизии и циклические переходы в начало таблицы. При неудачном выборе хеш-функции происходят аналогичные явления. В наихудшем варианте при полном заполнении таблицы алгоритмы циклического поиска свободного места приведут к зацикливанию. Поэтому при использовании хеш-таблиц необходимо стараться избегать очень

плотного заполнения таблиц. Обычно длину таблицы выбирают из расчета двукратного превышения предполагаемого максимального числа записей. Не всегда при организации хеширования можно правильно оценить требуемую длину таблицы, поэтому в случае большой заполненности таблицы может понадобиться рехеширование. В этом случае увеличивают длину таблицы, изменяют хеш-функцию и переупорядочивают данные.

Производить отдельную оценку плотности заполнения таблицы после каждой операции вставки нецелесообразно, поэтому можно производить такую оценку косвенным образом – по числу коллизий во время одной вставки. Достаточно определить некоторый порог числа коллизий, при превышении которого следует произвести рехеширование. Кроме того, такая проверка гарантирует невозможность зацикливания алгоритма в случае повторного просмотра элементов таблицы.

Рассмотрим алгоритм вставки, реализующий предлагаемый подход.

Вставка

1. i = 0
2. a = ((h(key) + c*i) div n + (h(key) + c*i) mod n) mod n
3. **если** t(a) = свободно или t(a) = удалено, **то** t(a) = key, записать элемент, *стоп элемент добавлен*
4. **если** i > m , **то** *стоп требуется рехеширование*
5. i = i + 1
6. **идти** к шагу 2

В данном алгоритме номер итерации сравнивается с пороговым числом m. Следует заметить, что алгоритмы вставки, поиска и удаления должны использовать идентичное образование адреса очередной записи.

Удаление

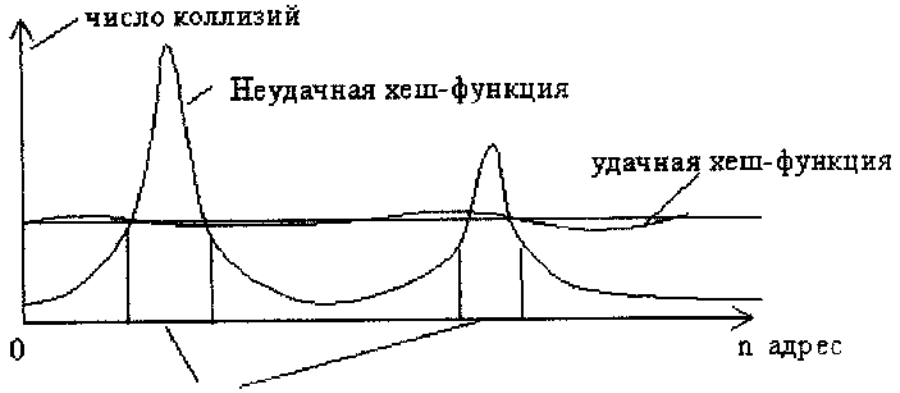
1. i = 0
2. a = ((h(key) + c*i) div n + (h(key) + c*i) mod n) mod n
3. **если** t(a) = key, **то** t(a) =удалено, *стоп элемент удален*
4. **если** t(a) = свободно или i>m, **то** *стоп элемент не найден*
5. i = i + 1
6. **идти** к шагу 2

Поиск

1. i = 0
2. a = ((h(key) + c*i) div n + (h(key) + c*i) mod n) mod n
3. **если** t(a) = key, **то** *стоп элемент найден*
4. **если** t(a) = свободно или i>m, **то** *стоп элемент не найден*
5. i = i + 1
6. **идти** к шагу 2

Оценка качества хеш-функции

Как уже было отмечено, очень важен правильный выбор хеш-функции. При удачном построении хеш-функции таблица заполняется более равномерно, уменьшается число коллизий и уменьшается время выполнения операций поиска, вставки и удаления. Для того чтобы предварительно оценить качество хеш-функции можно провести имитационное моделирование. Моделирование проводится следующим образом. Формируется целочисленный массив, длина которого совпадает с длиной хеш-таблицы. Случайно генерируется достаточно большое число ключей, для каждого ключа вычисляется хеш-функция. В элементах массива просчитывается число генераций данного адреса. По результатам такого моделирования можно построить график распределения значений хеш-функции. Для получения корректных оценок число генерируемых ключей должно в несколько раз превышать длину таблицы



Распределение коллизий в адресном пространстве таблицы

Если число элементов таблицы достаточно велико, то график строится не для отдельных адресов, а для групп адресов. Например, все адресное пространство разбивается на 100 фрагментов и подсчитывается число попаданий адреса для каждого фрагмента. Большие неравномерности свидетельствуют о высокой вероятности коллизий в отдельных местах таблицы. Разумеется, такая оценка является приближенной, но она позволяет предварительно оценить качество хеш-функции и избежать грубых ошибок при ее построении.

Оценка будет более точной, если генерируемые ключи будут более близки к реальным ключам, используемым при заполнении хеш-таблицы. Для символьных ключей очень важно добиться соответствия генерируемых кодов символов тем кодам символов, которые имеются в реальном ключе. Для этого стоит проанализировать, какие символы могут быть использованы в ключе.

Например, если ключ представляет собой фамилию на русском языке, то будут использованы русские буквы. Причем первый символ может быть большой буквой, а остальные – малыми. Если ключ представляет собой номерной знак автомобиля, то также несложно определить допустимые коды символов в определенных позициях ключа.

Рассмотрим пример *генерации ключа из десяти латинских букв, первая из которых является большой, а остальные –малыми*.

Пример 1: ключ – 10 символов, 1-й большая латинская буква, 2-10 малые латинские буквы

```
var i:integer; s:string[10];
begin
        s[1]:=chr(random(90-65)+65);
        for i:=2 to 10 do s[i]:=chr(random(122-97)+97);
end
```

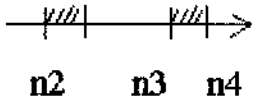
В данном фрагменте используется тот факт, что допустимые коды символов располагаются последовательными непрерывными участками в кодовой таблице. Рассмотрим более общий случай. Допустим, необходимо сгенерировать ключ из m символов с кодами в диапазоне от n1 до n2.

Пример 2: *Генерация ключа из m символов с кодами в диапазоне от n1 до n2 (диапазон непрерывный)*

```
for i:=1 to m do str[i]:=chr(random(n2-n1)+n1);
```

На практике возможны варианты, когда символы в одних позициях ключа могут принадлежать к разным диапазонам кодов, причем между этими диапазонами может существовать разрыв.

Пример 3: *Генерация ключа из m символов с кодами в диапазоне от n1 до n4 (диапазон имеет разрыв от n2 до n3)*




```
for i:=1 to m do
begin
        x:=random((n4 - n3) + (n2 – n1));
        if x<=(n2 - n1)  then str[i]:=chr(x + n1)
                         else str[i]:=chr(x + n1 + n3 – n2)
end;
```

Рассмотрим еще один конкретный пример. Допустим известно, что ключ состоит из 7 символов. Из них три первые символа – большие латинские буквы, далее идут две цифры, остальные – малые латинские.

Пример 4: длина ключа 7 символов: 3 большие латинские (коды 65-90); 2 цифры (коды 48-57); 2 малые латинские (коды 97-122)

```
var key: string[7];
begin
        for i:=1 to 3 do key[i]:=chr(random(90-65)+65);
        for i:=4 to 5 do key[i]:=chr(random(57-48)+57);
        for i:=6 to 7 do key[i]:=chr(random(122-97)+97);
end;
```

В рассматриваемых примерах мы исходили из предположения, что хеширование будет реализовано на языке Turbo Pascal, а коды символов соответствуют альтернативной кодировке.

Организация данных для ускорения поиска по вторичным ключам

До сих пор рассматривались способы поиска в таблице по ключам, позволяющим однозначно идентифицировать запись. Мы будем называть такие ключи первичными ключами. Возможен вариант организации таблицы, при котором отдельный ключ не позволяет однозначно идентифицировать запись. Такая ситуация часто встречается в базах данных. Идентификация записи осуществляется по некоторой совокупности ключей. Ключи, не позволяющие однозначно идентифицировать запись в таблице, называются вторичными ключами.

Даже при наличии первичного ключа, для поиска записи могут быть использованы вторичные. Например, поисковые системы internet часто организованы как наборы записей, соответствующих Web-страницам. В качестве вторичных ключей для поиска выступают ключевые слова, а сама задача поиска сводится к выборке из таблицы некоторого множества записей, содержащих требуемые вторичные ключи.

Инвертированные индексы

Рассмотрим метод организации таблицы с инвертированными индексами. Для таблицы строится отдельный набор данных, содержащий так называемые инвертированные индексы. Вспомогательный набор содержит для каждого значения вторичного ключа отсортированный список адресов записей таблицы, которые содержат данный ключ.

Поиск осуществляется по вспомогательной структуре достаточно быстро, так как фактически отсутствует необходимость обращения к основной структуре данных. Область памяти, используемая для индексов, является относительно небольшой по сравнению с другими методами организации таблиц.

Метод организации таблицы с инвертированными индексами

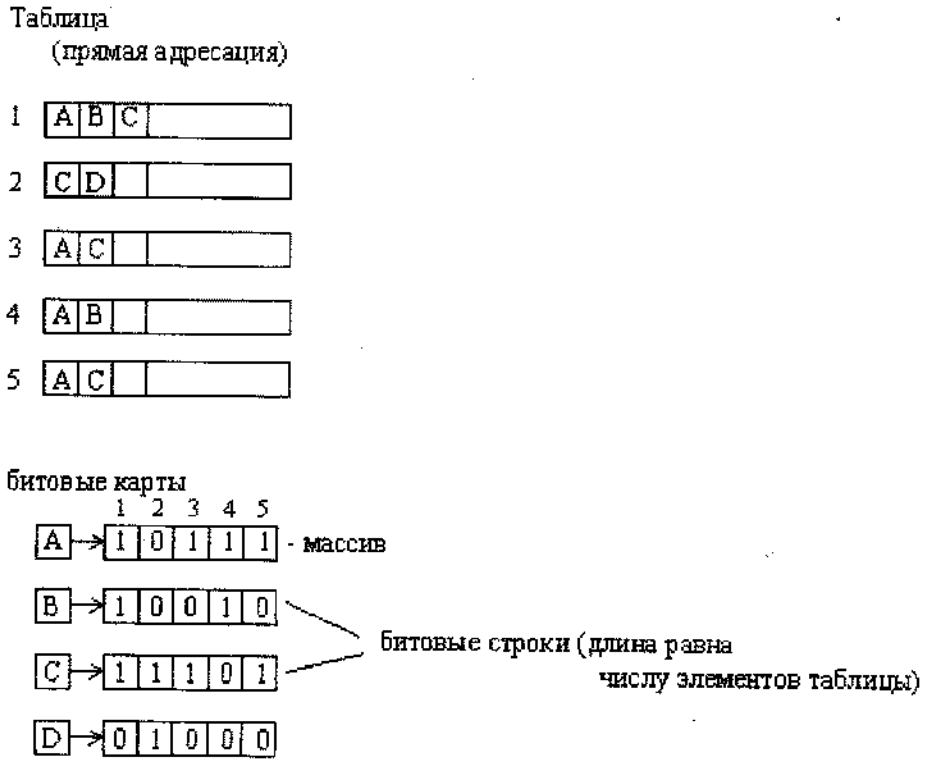
Недостатками данной системы являются большие затраты времени на составление вспомогательной структуры данных и ее обновление. Причем эти затраты возрастают с увеличение объема базы данных. Система инвертированных индексов является чрезвычайно удобной и эффективной при организации поиска в больших таблицах.

Битовые карты

Для таблиц небольшого объема используют организацию вспомогательной структуры данных в виде битовых карт. Для каждого значения вторичного ключа записей основного набора данных записывается последовательность битов. Длина последовательности битов равна числу записей. Каждый бит в битовой карте соответствует одному значению вторичного ключа и одной записи. Единица означает наличие ключа в записи, а ноль –отсутствие.

Основным преимуществом такой организации является очень простая и эффективная организация обработки сложных запросов, которые могут объединять значения ключей различными логическими предикатами. В этом случае поиск сводится к выполнению логических операций запроса непосредственно над битовыми строками и интерпретации результирующей битовой строки. Другим преимуществом является простота обновления карты при добавлении записей.

К недостаткам битовых карт следует отнести увеличение длины строки пропорционально длине файла. При этом заполненность карты единицами уменьшается с увеличением длины файла. Для большой длине таблицы и редко встречающихся ключах битовая карта превращается в большую разреженную матрицу, состоящую в основном из одних нулей.



Организация вспомогательной структуры данных в виде битовых карт

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.

Разумеется, методы и сферы применения хеширования не ограничиваются тем, что было нами рассмотрено. Не вдаваясь в строгий анализ эффективности, мы рассматривали только базовые, наиболее известные методы. Помимо них можно отметить полиномиальное хеширование (М. Ханан и др., 1963), упорядоченное хеширование (О. Амбль, 1973), мультипликативное хеширование (Р. Флойд), хеширование Фибоначчи (Я. Одерфельд), расширяемое хеширование (Ю. Нивергельт и др., 1979), линейное хеширование (В. Литвин, 1980). Подробнее о методах хеширования см. [3, 6, 8—11].

Литература

1. Hellerman H. Digital Computer System Principles. McGraw-Hill, 1967.
2. Ершов А.П. Избранные труды. Новосибирск: ВО «Наука», 1994.
3. Кнут Д. Искусство программирования, т.3. М.: Вильямс, 2000.
4. Peterson W.W. Addressing for Random-Access Storage // IBM Journal of Research and Development. 1957. V.1, N2. P.130—146.
5. Morris R. Scatter Storage Techniques // Communications of the ACM, 1968. V.11, N1. P.38—44.
6. Ахо А., Хопкрофт Дж., Ульман Дж. Структуры данных и алгоритмы. М.: Вильямс, 2000.
7. Чмора А. Современная прикладная криптография. М.: Гелиос АРВ, 2001.
8. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ. М.: МЦНМО, 2001.
9. Вирт Н. Алгоритмы + структуры данных = программы. М.: Мир, 1985.
10. Керниган Б., Пайк Р. Практика программирования. СПб.: Невский диалект, 2001.
11. Шень А. Программирование: теоремы и задачи. М.: МЦНМО, 1995.

Адреса сайтов, посвящённых вопросам хеширования.

http://k46.aanet.ru/textbooks/str_alg/index0.htm – на данном сайте опубликован конспект лекций по курсу «Структуры и алгоритмы обработки данных».

http://www.optim.ru/cs/2000/4/bintree_htm/beentree_main.asp – прекраснейшая статья, посвящённая прикладным вопросам хеширования. Даются «исходники» описания классов различных типов хеширования.

Если вы прочтёте «Уроки хакера Windows 2000 Часть 1» (http://oleg-koenig.hotbox.ru/files/page6_1.html#1), то вы получите некоторое представление о том, на сколько актуальны вопросы хеширования.

По адресу http://www.osp.ru/pcworld/2001/06/152.htm опубликована статья «Расстановка, или Схемы хеширования» Руслана Богатырева и Андрея Шилова.

На страницах сайта http://www.mpei.ac.ru/homepages/mm/Lab/A1496/uov/main.htm вы сможете ознакомиться с алгоритмом хеширования АВЛ-деревьями, который удобно использовать при работе в реальном времени.

В статье Геннадия Баранова «Обзор методов сжатия данных» (http://www.izip.nm.ru/media/articles/art2.htm) разбираются вопросы использования хеширования при сжатии информации.

Использование алгоритмов хеширования для организации работы кеш-буферов раскрывается на сайте http://users.podolsk.ru/alex/html/doc/Novell/kayak/ru411ref/cncptrus/DOCMODUL/ch30s40.htm