**Лабораторная работа 6****. Хеширование.**

**Цель работы: Закрепление теоретических знаний и получение практических навыков создания программ для хеширования данных.**

**Краткие теоретические сведения**

1. **Общие сведения и определения**

Алгоритмы поиска предполагают, что прежде, чем найти требуемую запись, необходимо организовать просмотр и сравнение с аргументом поиска некоторого количества ключей. Организация данных (массив, таблица, куча, бинарное дерево и т. д.) и порядок, в котором вставляются ключи, определяют то число ключей, которое должно быть проверено до получения нужного ключа. Очевидно, что эффективными методами поиска являются те методы, которые минимизируют число этих проверок. Было показано, что наиболее эффективным методом поиска является бинарный поиск, требующий в наихудшем случае выполнения числа сравнений, приблизительно равного log2N. Однако бинарный поиск не может быть применен для таблиц с произвольным расположением элементов: элементы должны быть расположены в отсортированном порядке.

В идеале мы бы хотели иметь такую организацию таблицы, при которой не было бы ненужных сравнений. Посмотрим, возможна ли такая организация. Если каждый ключ должен быть извлечен за один доступ, то положение записи внутри такой таблицы может зависеть только от данного ключа. Оно не может зависеть от расположения других ключей, как это имеет место в дереве. Наиболее эффективным способом организации такой таблицы является одномерный массив, т. е. доступ к каждой записи обеспечивается с помощью ее уникального целочисленного индекса, который определяет позицию элемента в общей таблице. Если ключи записей являются целыми числами, то *сами ключи могут использоваться как индексы в массиве*. Рассмотрим пример такой системы.

Предположим, что некоторая фирма-производитель имеет таблицу с перечнем производимых изделий, состоящую из 1000 наименований изделий, причем каждое изделие имеет уникальный номенклатурный номер из трех цифр. Тогда обычным способом хранения этой таблицы является описание некоторого массива:

a[0], a[1], a[2], …, a[HighIndex].

Общее число элементов массива обозначаем N, тогда HighIndex = N - 1.

В нашем случае можно задать N = 1000, и если номера изделий, составляющие содержимое поля Key, являются ключами, то целесообразно определить, что эти номера используются как индексы в данном массиве. Тогда запись с ключом, например, 67 при вставке размещается в элементе a[67]. В такой таблице легко организовать поиск: элемент, соответствующий аргументу поиска ArgSearch, - это элемент a[ArgSearch]. Мы видим, что при такой организации таблицы при поиске не нужно производить ни одного сравнения. Сложность поиска соcтавляет O(1).

Та же самая структура может использоваться для организации файла производимых изделий, даже если на складах фирмы накопилось до, например, 200 наименований изделий (при условии что ключи по-прежнему состоят из трех цифр). Хотя многие ячейки в массиве a тогда бы соответствовали несуществующим ключам, эти потери компенсируются преимуществом прямого доступа к записи с информацией о каждом из существующих изделий.

К сожалению, такая система не всегда имеет практический смысл. Например, предположим, что фирма имеет некоторую таблицу производимых изделий, состоящую из более 1000 пунктов, и ключ каждой записи является номером изделия из семи цифр. Для применения прямой индексации с использованием полного семизначного ключа потребовался бы массив из 10000000 (10 млн.) элементов. Ясно, что это привело бы к потере неприемлемо большого пространства памяти, поскольку совершенно невероятно, что какая-либо фирма может иметь больше чем несколько тысяч наименований изделий.

Пример: оплата коммунальных услуг

Квартира, вода и пр. – требуется номер ЕРИП – 10 цифр

Электроэнергия – требуется номер абонента – 9 цифр

Оплата Интернет – требуется номер договора -13 цифр

Каждый житель Беларуси может быть идентифицирован ключом из 7 цифр!!!

Поэтому необходим некоторый метод преобразования ключа в какое-либо целое число внутри ограниченного диапазона. В идеале в одно и то же число не должны преобразовываться два различных ключа. К сожалению, такого идеального метода не существует. Попытаемся разработать методы, которые приближаются к идеальным, и определить, какие действия надо предпринять, когда идеальный случай не достигается.

Рассмотрим опять пример с таблицей наименований изделий фирмы, в которой каждая запись задается ключом из семизначного номера изделия. Предположим, что фирма имеет не более 1000 наименований изделий и что для каждого изделия используется только одна запись. Тогда для хранения всего файла будет достаточно массива из 1000 элементов. Этот массив индексируется целым числом в диапазоне от 0 до 999 включительно. В качестве индекса записи об изделии в этом массиве используются три последние цифры номера изделия. Это показано на рисунке 1. Отметим, что два ключа, которые близки друг к другу как числа (такие как 4618396 и 4618996), могут располагаться дальше друг от друга в этой таблице, чем два ключа, которые значительно различаются как числа (такие как 0000991 и 9846995). Это происходит из-за того, что для определения позиции записи используются только три последние цифры ключа.

Функция, которая трансформирует ключ в некоторый индекс в таблице, называется *функцией хеширования*(hash function). Другие названия функции хеширования: *хеш-функция*, *функция перемешивания*. Если h является некоторой хеш-функцией, а Кey некоторый ключ, то h(Кey) называется *значением хеш-функции* от ключа Кey и является индексом, по которому должна быть помещена запись с ключом Кey. Преобразование ключа элемента в значение индекса называется *хешированием* (hashing). Массив, используемый для хранения элементов, в котором индексы определяются с помощью хеширования ключей, называется *хеш-таблицей* (hash table).

  Если мы обозначим остаток от деления X на Y как X % Y, то хеш‑функция для вышеприведенного примера есть

h(Кey) = Кey % 1000.

Значения, которые выдает функция h, должны покрывать все множество индексов в таблице. Например, функция Кey % 1000 может дать любое целое число в диапазоне от 0 до 999 в зависимости от значения Кey. Хорошей идеей является таблица, размер которой немного больше, чем число вставляемых записей. Это иллюстрируется на рисунке 1, где несколько позиций таблицы не используются.

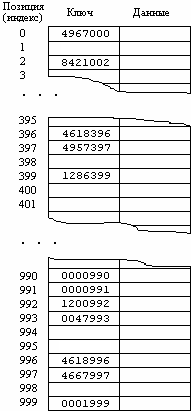


Рисунок 1 - Размещение записей в таблице, когда позиция записи определяется по трем последним цифрам ключа

**2. Коллизии при хешировании**

При получении таблицы с помощью преобразования ключей имеет место один недостаток. Предположим, что существуют два различных ключа k1 и k2 такие, что h(k1) = h(k2) Например для ключей 496843 и 557843 при использовании хеш-функции h(Кey) = Кey % 1000 получим h(k1)=h(k2)=843 Когда запись с ключом k1 вводится в таблицу, она вставляется в позицию с индексом h(k1). Но когда хешируется ключ k2, получаемая позиция является той же позицией, в которой хранится запись с ключом k1. Ясно, что две записи не могут занимать одну и ту же позицию. Такая ситуация называется *коллизией*(collision)*при хешировании* или *столкновением*. Иногда коллизию называют*конфликтом*.

В примере с изделиями на рисунке 1 коллизия при хешировании произойдет, если в таблицу будет добавлена, например, запись с ключом 0596993. Далее мы будем исследовать возможности, как найти решение в такой ситуации. Следует отметить, что хорошей хеш-функцией является такая функция, которая минимизирует коллизии и распределяет записи равномерно по всей таблице. Поэтому и желательно иметь массив с размером больше, чем число реальных записей. Чем больше диапазон хеш-функции, тем менее вероятно, что два ключа дадут одинаковое значение хеш-функции. Конечно, при этом возникает компромисс между временем и пространством. Наличие пустых мест в массиве неэффективно с точки зрения использования пространства, но при этом уменьшается необходимость разрешения коллизий при хешировании, что, следовательно, является более эффективным в смысле временных затрат.

Алгоритм, который позволяет распределять в таблице записи, конкурирующие с другими записями в одну ячейку хеш-таблицы, называется методом *разрешения коллизий* (collision resolution).

Может показаться, что из-за возможности коллизий вся схема хеширования теряет смысл, но у проблемы существует несколько обходных решений. При определении массива количество зарезервированных ячеек может вдвое превышать количество элементов данных. Таким образом, приблизительно половина ячеек остается пустой. Одно из возможных решений при возникновении коллизии заключается в систематизированном поиске пустой ячейки и вставке нового элемента в нее (вместо индекса, полученного в результате применения хеш-функции). Такое решение называется *открытой адресацией*.

Во втором решении создается массив, содержащий связанные списки данных вместо самих данных. При возникновении коллизии новый элемент просто вставляется в список с соответствующим индексом. Этот метод называется *методом цепочек*.

1. **Методы разрешения коллизий**

Если элемент данных не удается разместить в ячейке с индексом, вычисленным посредством хеш-функции, метод открытой адресации ищет в массиве другую ячейку.

Существует три разновидности открытой адресации, различающихся способом поиска следующей свободной ячейки:

* *линейное пробирование (linear probing)*,
* *квадратичное пробирование (quadric probing),*
* *двойное хеширование (double hashing)*.

Согласно методу открытой адресации, если ключ k отображается в табличный индекс i, а он уже занят, в таблице просматриваются другие индексы до тех пор, пока не будет найдено свободное место для объекта, приведшего к коллизии (G(i) – некоторая функция):

h = h(k) + G(i);

В качестве функции G(i) можно использовать

* 1. линейную функцию G(i)= n, где n- некоторое целое число (linear probing),
  2. квадратичную функцию G(i)= n2 , (quadratic probing),
  3. повторное хеширование (rehashing), называемое также двойным хешированием (double hashing).

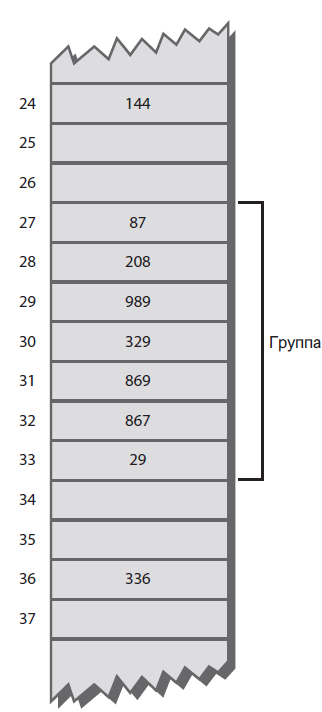
При ***линейном пробировании*** используется следующая последовательность индексов в предположении, что таблица круговая:

i, i+1, …, m-1, 0, 1, 2, …, i-1.

Такая последовательность получается, если функция G(i) = i, индексы hi, используемые для поиска, имеют вид:

|  |  |
| --- | --- |
| h0 = h(k),  hi = hi-1 + n i = 1, …, m-1. | (3) |

Линейное пробирование является простым в реализации, но не является хорошей стратегией разрешения коллизий, т.к. ведет к группировке (кластеризации), при которой вокруг первичных ключей (ключей, для которых конфликта при занесении в таблицу не было) возникают все более длинные последовательности занятых подряд индексов.



**Рисунок 2.** Пример группировки

Например, пусть надо добавить в таблицу несколько объектов. При этом используется хеш-функция вида

h(Key)=Key%100 (последние две цифры ключа)

и функция G(i)=1;

Объект 1 с ключом 1324, Объект 2 с ключом 1627, Объект 3 с ключом 6827, Объект 4 с ключом 3227.

Объект 1 будет размещен в ячейке с индексом 24. Объект 2 будет размещен в ячейке с индексом 27. Объект 3 должен быть размещен в ячейке с индексом 27, но эта ячейка уже занята, поэтому Объект 3 занимает следующий свободный элемент с индексом 28. Объект 4 должен быть добавлен в ячейку с индексом 27, но этот индекс уже занят, поэтому алгоритм пытается разместить объект 4 в ячейке 28, но и она занята, тогда Объект 4 размещается в ячейке 29 и так далее. При наличии коллизий возникает так называемая группировка объектов в хеш-таблице.

Для уменьшения группировки можно в линейном пробировании использовать функцию G(i) вида

G(i)=i\*n, где n – целое число. (Например G(i)=(i\*n) mod 7). В этом случае после каждого шага нахождения занятой ячейки поиск пустой ячейки будет происходить все дальше и дальше.

Коллизии усложняют не только процесс вставки объекта в хеш-таблицу, но также и процесс поиска объекта по ключу. Например, если необходимо найти информацию об Объекте 4, то по его ключу 3227 вычисляется значение хеш-функции, равное 27. Но в элементе с индексом 27 находится Объект 2. Линейный поиск будет продолжаться до тех пор, пока не будет найден объект с заданным ключом или пустой элемент таблицы. Если в таблице найден пустой элемент, это означает, что в ней не содержится объект с заданным ключом.

Итак, при открытой адресации с линейным пробированием возникает проблема группировки. Образовавшиеся группы начинают расширяться. Элементы, хешируемые в пределах группы, добавляются в конец группы, в результате чего группа становится еще больше. Чем больше размер группы, тем быстрее она растет. Явление напоминает собирающуюся толпу зевак: первые зрители подходят посмотреть, что случилось, а остальные хотят узнать, на что все смотрят. Чем больше толпа, тем больше людей она привлекает.

Отношение количества элементов в таблице к размеру таблицы называется *коэффициентом заполнения*. Таблица с 10 000 ячеек, содержащая 6667 элементов, имеет коэффициент заполнения 2/3.

коэффициент\_заполнения = количество\_элементов / размер\_массива;

Группы могут образовываться даже при относительно небольшом коэффициенте заполнения. Одни части хеш-таблицы могут быть заполнены большими группами, другие почти не содержать элементов. Группировка снижает быстродействие таблицы.

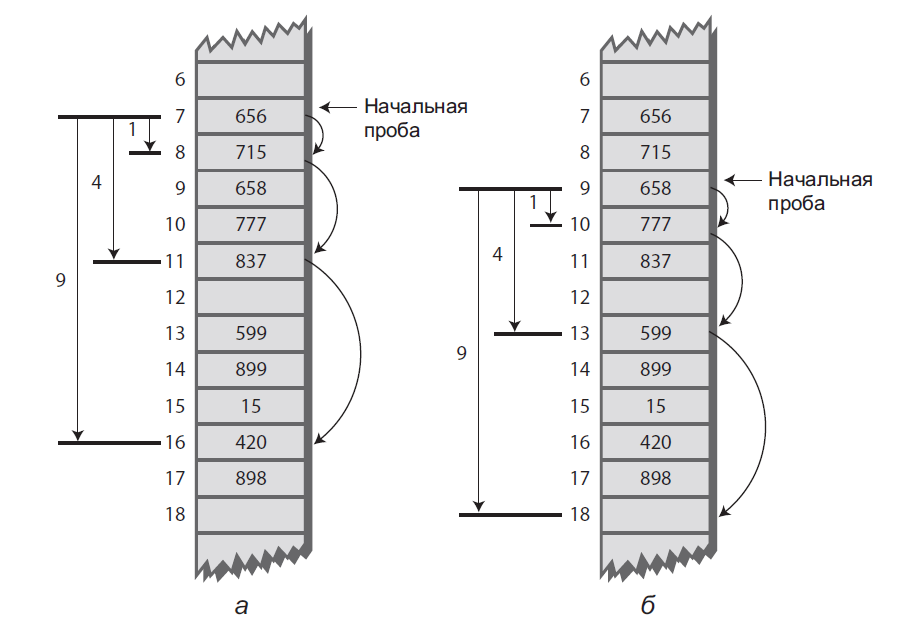
***Квадратичное пробирование*** пытается избежать образования групп. Его идея заключается в том, чтобы проверять ячейки, находящиеся на больших расстояниях (вместо ячеек, находящихся вблизи от исходной позиции хеширования).

При квадратичном пробировании выполняется просмотр индексов таблицы на квадратичном расстоянии (функция G(i) = i2n)

**Вычисление шага**

При линейном пробировании, если первичный индекс хеширования равен *x*, то последующие пробы проверяют позиции *x* + 1, *x* + 2, *x* + 3 и т. д. При квадратичном пробировании проверяются позиции *x* + 1, *x* + 4, *x* + 9, *x* + 16, *x* + 25 и т. д. Расстояние от исходной позиции вычисляется как квадрат номера шага.

Пример квадратичного пробирования представлен на рис. 3.

****

**Рис. 3.** Квадратичное пробирование: *а* — успешный поиск элемента 420;

*б* — безуспешный поиск элемента 481

**Проблемы с квадратичным пробированием**

Квадратичное пробирование решает проблему группировки, присущую линейной группировке; эта разновидность группировки называется *первичной*. Однако при квадратичном пробировании возникает другая, более тонкая проблема группировки. Дело в том, что все ключи, хешируемые в конкретный индекс, ищут свободную ячейку в одной и той же последовательности.

Допустим, элементы 184, 302, 420 и 544 хешируются в индекс 7 и вставляются в хеш-таблицу в указанном порядке. В этом случае индекс 302 потребует смещения на одну ячейку, индекс 420 — смещения на 4 ячейки, а индекс 544 — на 9 ячеек. У каждого дополнительного элемента, хешируемого в индекс 7, смещение будет еще большим. Это явление называется *вторичной группировкой*.

Вторичная группировка не создает серьезных проблем. И все же квадратичное пробирование на практике применяется нечасто, потому что существует другое, более удачное решение.

Однако даже квадратичная проверка может привести к группировке (кластеризации). Вторичная группировка возникает из-за того, что алгоритм, генерирующий последовательность смещений для квадратичного пробирования, всегда генерирует одни и те же смещения: 1, 4, 9, 16 и т. д.

Кроме того, недостаток этой стратегии заключается в том, что при добавлении объекта можно пропустить свободный элемент. Фактически при квадратичной проверке используется половина таблицы.

**Двойное хеширование**

Для устранения как первичной, так и вторичной группировки применяется алгоритм *двойного хеширования*.

В идеале последовательность проб должна генерироваться в зависимости от ключа (вместо использования набора одинаковых смещений для всех ключей).

В этом случае числа с разными ключами, хешируемые в один индекс, будут использовать разные последовательности смещений. Задача решается повторным хешированием ключа с другой хеш-функцией и использованием результата в качестве смещения. Для заданного ключа размер

смещения остается постоянным при пробировании, но для разных ключей используются разные размеры.

Практический опыт показал, что вторичная хеш-функция должна обладать некоторыми характеристиками:

* Она не должна совпадать с первичной хеш-функцией.
* Ее результат никогда не должен быть равен 0 (в противном случае смещения не будет, все пробы будут приходиться на одну ячейку, а алгоритм войдет в бесконечный цикл).

Эксперты обнаружили, что для решения этой задачи хорошо подходят функции вида

***смещение = константа - (ключ % константа);***

где *константа* — простое число, меньшее размера массива. Пример функции:

stepSize = 5 - (key % 5);

Разные ключи могут хешироваться в один индекс, но для них (вероятно) будут сгенерированы разные смещения.

**Выбор простого размера таблицы**

При использовании двойного хеширования размер таблицы должен быть простым числом. Чтобы понять смысл этого требования, представьте ситуацию, в которой размер таблицы простым числом не является. Предположим, размер массива равен 15 (индексы от 0 до 14), а конкретный ключ хешируется в исходный индекс 0 со смещением 5. Пробы будут выполняться в последовательности 0, 5, 10, 0, 5, 10 и т. д. до бесконечности. Проверяются только эти три ячейки, поэтому алгоритм «не увидит» пустые ячейки 1, 2, 3 и т. д., а из попытки выполнения операции ничего не выйдет.

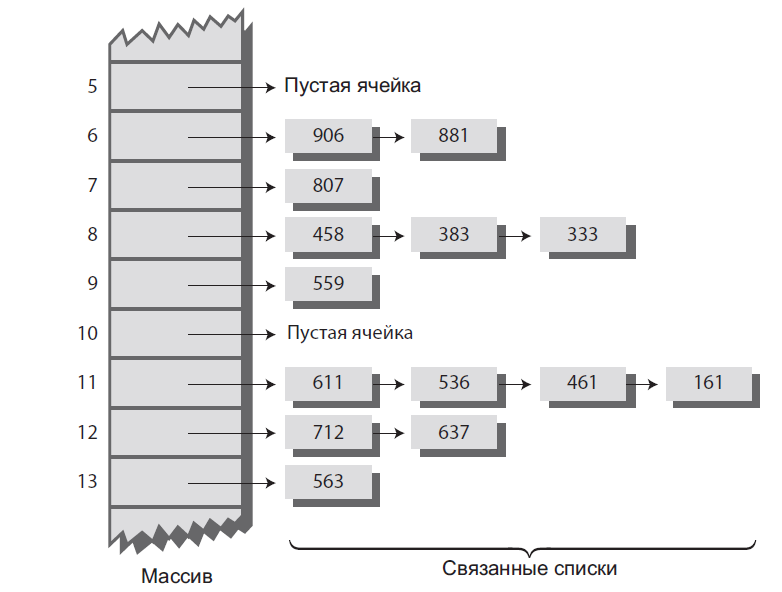
Если бы размер массива был равен 13 (простое число), то в процессе пробирования в конечном итоге была бы проверена каждая ячейка: 0, 5, 10, 2, 7, 12, 4, 9, 1, 6, 11, 3 и т. д. Даже если в массиве имеется всего одна пустая ячейка, она будет успешно обнаружена. Простой размер массива не делится нацело ни на какое число, поэтому последовательность проб рано или поздно проверит каждую ячейку.

Аналогичный эффект проявляется с квадратичным пробированием. Однако в этом случае смещение увеличивается с каждым шагом, что в конечном итоге приведет к переполнению переменной, в которой оно хранится (хотя это тоже приведет к прерыванию бесконечного цикла).

В общем случае при использовании открытой адресации предпочтение следует отдавать двойному хешированию.

**Метод цепочек**

При открытой адресации коллизии разрешаются поиском свободных ячеек в хеш-таблице. Другое возможное решение основано на ведении отдельного связанного списка по каждому индексу в хеш-таблице. Ключ элемента данных хешируется в индекс обычным способом, а полученный элемент вставляется в связанный список по этому индексу. Другие элементы, хешируемые в тот же индекс, просто добавляются в связанный список; искать пустые ячейки в первичном массиве просто не нужно. Принцип работы метода цепочек показан на рис. 5.



**Рис. 5.** Метод цепочек

На концептуальном уровне метод цепочек реализуется проще, чем различные схемы пробирования, используемые при открытой адресации. С другой стороны, объем кода увеличивается, так как программа должна реализовать механизм ведения связанных списков — как правило, для их представления в программу вводятся дополнительные классы.

**Коэффициенты заполнения**

Коэффициент заполнения (отношение количества элементов в хеш-таблице к ее размеру) при использовании метода цепочек обычно отличается от коэффициента заполнения при открытой адресации. В методе цепочек массив из *N* ячеек вполне может содержать более *N* элементов; таким образом, коэффициент заполнения может быть равен единице и более. Это абсолютно нормально; коэффициент, больший единицы, попросту означает, что списки некоторых ячеек состоят из двух и более элементов.

Конечно, большое количество элементов в списках замедляет обращение к данным, потому что для обращения к заданному элементу приходится просматривать в среднем половину элементов списка. Исходная ячейка находится быстро за время *O*(1), но поиск по списку выполняется за время, пропорциональное *M* — среднему количеству элементов в списке. Следовательно, поиск выполняется за время *O*(*M*), а чрезмерное заполнение списков нежелательно.

Хеш-таблицы с коэффициентом заполнения 1 встречаются достаточно часто. При таком коэффициенте заполнения примерно треть ячеек остается пустой, треть содержит один элемент и еще треть —два и более элементов.

При открытой адресации коэффициент заполнения, превышающий порог в ½ или 2/3, приводит к резкому снижению быстродействия. При реализации метода цепочек коэффициент заполнения может превысить единицу без сколько-нибудь значительного ущерба для быстродействия. Таким образом, метод цепочек обладает значительно большей стабильностью, особенно если объем данных в хеш-таблице трудно предсказать заранее.

**Размер таблицы**

С методом цепочек выбор простых чисел в качестве размера таблицы уже не играет такой важной роли, как при квадратичном пробировании и двойном хешировании. Метод цепочек обходится без проб, поэтому беспокоиться о возможном зацикливании процесса пробирования (из-за размера массива, кратного смещению) уже не нужно.

С другой стороны, некоторые виды распределения ключей могут привести к группировке данных, когда размер массива не является простым числом.

**Гнезда**

Другое решение, концептуально сходное с методом цепочек, заключается в создании массива (вместо связанного списка) в каждой ячейке хеш-таблицы. Такие массивы иногда называются *гнездами* (buckets). Однако гнездовые решения по эффективности уступают решениям на базе связанных списков из-за проблемы выбора размера гнездовых массивов. Слишком малые размеры массивов могут привести к переполнению, а слишком большие — к неэффективному расходованию памяти. У связанных списков с динамическим выделением памяти такой проблемы

не существует.

1. **Хеш-функции**

Принято считать, что хорошей, с точки зрения практического применения, является такая *хеш-функция*, которая удовлетворяет следующим условиям:

* функция должна быть простой с вычислительной точки зрения;
* функция должна распределять ключи в хеш-таблице наиболее равномерно;
* функция не должна отображать какую-либо связь между значениями ключей в связь между значениями адресов;
* функция должна минимизировать число *коллизий* – то есть ситуаций, когда разным ключам соответствует одно значение *хеш-функции* (ключи в этом случае называются *синонимами* ).

Основной целью хеш-функции является преобразование диапазона ключей в диапазон индексов, обеспечивающее равномерное распределение ключей по индексам хеш-таблицы. Ключи могут быть как полностью случайными, так и частично детерминированными.

**Случайные ключи**

*Идеальная* хеш-функция ставит в соответствие каждому значению ключа уникальный индекс. Такое поведение возможно только при нетипично «хорошем» поведении ключей с диапазоном, достаточно узким для прямого использования в качестве индексов (как в примере с табельными номерами работников).

На практике обычно ни одно из этих условий не выполняется, и хеш-функции приходится сжимать широкий диапазон ключей в меньший диапазон индексов.

Желательное поведение хеш-функции зависит от распределения значений ключей в конкретной базе данных. Ранее мы предполагали, что данные равномерно распределены по всему диапазону. Для такой ситуации хеш-функция вида

***индекс = ключ % размер\_массива;***

достаточно хорошо работает. В ней используется только одна математическая операция, поэтому для действительно случайных ключей полученные индексы тоже будут случайными, а следовательно, будут иметь хорошее распределение.

**Неслучайные ключи**

Однако на практике данные часто распределяются не полностью случайным образом. Представьте базу данных, в которой в качестве ключей используются номера деталей машин — допустим, числа в формате

033-400-03-94-05-0-535

Формат номера интерпретируется следующим образом:

Цифры 0–2: код поставщика (от 1 до 999, в настоящее время до 70).

Цифры 3–5: код категории (100, 150, 200, 250 и т. д. до 850).

Цифры 6–7: месяц запуска в производство (от 1 до 12).

Цифры 8–9: год запуска в производство (от 00 до 99).

Цифры 10–11: серийный номер (от 1 до 99).

Цифра 12: признак токсичности (0 или 1).

Цифры 13–15: контрольная сумма (остаток от деления суммы других полей на

100).

Ключ для приведенного номера детали будет иметь вид 0 334 000 394 050 535.

Однако такие ключи распределены неравномерно. Большинство чисел из диапазона 0 до 9 999 999 999 999 999 не может использоваться в качестве ключа (например, числа с кодами поставщиков выше 70, коды категорий, не кратные 50, или месяцы от 13 до 99). Кроме того, контрольная сумма вычисляется в зависимости от других чисел. Чтобы эти числа образовали более случайный диапазон, их необходимо подвергнуть дополнительной обработке.

**Исключение неинформативных частей**

Поля ключа необходимо по возможности «сжать», чтобы каждый их бит нес полезную информацию. Например, поле кода категории можно изменить так, чтобы в нем могли храниться только значения в диапазоне от 0 до 15. Кроме того, поле контрольной суммы следует исключить из ключа, так как оно не содержит дополнительной информации; это избыточные данные, намеренно введенные с целью контроля. Для сжатия полей ключа применяются различные поразрядные операции.

**Использование всех данных**

Каждая часть ключа (кроме неинформативных частей, о которых говорилось выше) должна вносить свой вклад в вычисление хеш-функции. Не ограничивайтесь первыми четырьмя цифрами или другими аналогичными схемами. Чем больше данных учитывается при вычислении ключа, тем больше вероятность хеширования ключей по всему диапазону индексов.

Иногда диапазон допустимых значений ключей оказывается настолько большим, что он приводит к переполнению переменных типа int и long. О том, как решается проблема переполнения, будет рассказано ниже, когда мы вернемся к хешированию строк.

Подведем итог: хеш-функция должна быть простой и быстрой. При этом она должна исключать из обработки неинформативные части ключа и использовать все данные.

1) Наиболее известная хеш-функция ***метод деления*,** при котором некоторый целый ключ делится на размер таблицы и остаток от деления берется в качестве значения хеш-функции. Эта хеш-функция имеет вид

***h(Кey) = Кey % N.***

Предположим, что N равно 1000 и что все ключи оканчиваются на три одинаковые цифры (например, последние три цифры номера изделия могут обозначать номер фабрики и программа пишется для этой фабрики). Тогда остаток от деления на 1000 для всех ключей будет одним и тем же, так что для всех записей, кроме первой, будет происходить коллизия при хешировании. Ясно, что при таком наборе ключей должна использоваться другая хеш-функция. Было найдено, что наилучшие результаты для метода деления, как и для большинства других методов, получаются тогда, когда размер таблицы N является простым числом (т. е. N не делится ни на какое положительное целое число, кроме 1 и N).

2) Значение хеш-функции, использующей ***метод умножения*,** формируется следующим образом. Выбирается константа А, такая, что 0<A<1. Ключ умножается на эту константу и выбирается заданное количество цифр в дробной части.

Заметим, что в методе умножения, во-первых, в качестве значения А рекомендуется использовать иррациональное число, которое близко к длине машинного слова; хорошие результаты дает значение a=(Sqrt(5)-1)/2, где Sqrt - функция вычисления квадратного корня; 3)

3)В ***методе преобразования системы* счисления** ключ представляется в некоторой р-ичной системе счисления:

Кey = d0 + d1×p + d2×p2 + …

Выбирается основание q новой системы счисления такое, что q < p. Пусть s – некоторое целое число. Тогда для метода преобразования системы счисления значение хеш-функции вычисляется в новой системе счисления со «старыми» коэффициентами:

h(Кey) = d0 + d1×q + d2×q2 + … + dS-1×qS-1.

Очевидно, s ограничивает порядок значения хеш-функции в q-ичной системе счисления. Трудоемкость этого метода больше, чем у предыдущих методов, поскольку для вычисления значения h(Кey) нужно s операций умножения и сложения.

5) При использовании метода, известного как ***метод середины квадрата*,** ключ умножается сам на себя и в качестве индекса используется несколько средних цифр этого квадрата. Если данный квадрат рассматривается как десятичное число, то размер таблицы должен быть некоторой степенью 10, а если он рассматривается как двоичное число, то размер таблицы должен быть степенью 2. Причиной возведения числа в квадрат до извлечения средних цифр является то, что все цифры первоначального числа дают свой вклад в значение средних цифр квадрата.

6) При ***методе свертки*** ключ разбивается на несколько сегментов, над которыми выполняется операция сложения для формирования хеш-функции.

Такое решение гарантирует, что хеш-код будет зависеть от каждой цифры исходных данных. Количество цифр в группе должно соответствовать размеру массива. Другими словами, для массива из 1000 элементов каждая группа должна состоять из трех цифр.

Предположим, вы хотите хешировать 9-разрядные ключи. Если размер массива равен 1000, число из 9 цифр делится на три группы из трех цифр. Так, для кода 123-45-6789 вычисляется ключ 123 + 456 + 789 = 1368. Оператор % усекает полученную сумму, чтобы максимальное значение индекса составляло 999. В нашем примере 1368%1000 = 368. Если бы размер массива был равен 100, то ключ из 9 цифр пришлось бы разделить на четыре группы из двух цифр и одну группу из одной цифры:

12 + 34 + 56 + 78 + 9 = 189, и 189%100 = 89.

Когда размер массива кратен 10, работу этой схемы легко понять. Но оптимальный размер массива должен быть простым числом.

**3.2 Хеш-функции для строковых ключей**

Если ключи не являются числами, то они должны быть преобразованы в целые числа перед применением описанных выше хеш-функций. Для этого имеется несколько способов. Например, для строки символов в качестве двоичного числа может интерпретироваться внутреннее двоичное представление кода каждого символа. Недостатком этого является то, что для большинства ЭВМ двоичные представления всех букв или цифр очень похожи друг на друга.

Так, заглавная буква русского алфавита ’И’ представляется цифрами 200, а малая буква ’в’ представляется цифрами 226. Ключ «Иван» после слияния всех номеров букв представляется целым числом 200226224237. Когда существует некоторое целое представление строки символов, то для сведения его к приемлемому размеру может быть использован метод свертки или середины квадрата.

Метод ***весовых коэффициентов*** использует значение позиции каждого символа во избежание коллизий при использовании анаграмм в качестве ключей (анаграмма - это слово, полученное из другого слова перестановкой его букв). Этот метод реализуется подпрограммой SimpleHash, которая имеет следующий вид:

**Function** SimpleHash(**Const** aKey: String;   
N: Integer): Integer;

**Var** i: Integer;

Hash: LongInt;

**Begin**

Hash:= 0;

**For** i:= 1 **To** Length(aKey) **Do**

Hash:= ((Hash\*17) + Ord(aKey[i])) **Mod** N;

Result:= Hash;

**If** Result < 0 **Then** Inc(Result, N);

**End**;

Эта подпрограмма воспринимает два параметра: первый из них - хешируемая строка, второй - число ячеек в таблице. Алгоритм по циклу изменяет переменную Hash, умножая ее текущее значение на простое число 17 и прибавляя порядковый номер i-ого символа; завершается изменение делением по модулю на размер таблицы. Заключительный If требуется потому, что промежуточное значение переменной может быть отрицательным, а программа, вызывающая эту функцию, ожидает получить результат, значение которого находится в диапазоне от 0 до N-

**Использование простых чисел при вычислении остатка**

В хеш-функциях часто используется оператор вычисления остатка (%) с размером таблицы. Вы уже знаете, что выбор простого числа в качестве размера таблицы играет важную роль при квадратичном пробировании и двойном хешировании. Но если сами ключи имеют неслучайное распределение, размер таблицы должен быть простым числом независимо от выбора системы хеширования.

Дело в том, что, если многие ключи имеют общий делитель с размером массива, они часто хешируются в одну позицию, а это приводит к группировке. Простой размер таблицы исключает такую возможность. Например, если размер таблицы был бы кратен 50 в нашем примере с деталями машин, то все коды категорий хешировались бы в индексы, кратные 50. Простое число (например, 53) гарантирует, что ключи не будут делиться нацело на размер таблицы.

Итак, тщательно анализируйте ключи и адаптируйте алгоритм хеширования для устранения любых аномалий в распределении ключей.

**2. Задание на лабораторную работу.**

* + 1. Написать программу для вычисления хеш-функции строкового ключа.
    2. Построить хеш-таблицы (по варианту), внося в нее хеш-коды фамилий всех студентов группы.
    3. Определить коэффициент заполнения таблицы.

Варианты задания

|  |  |  |
| --- | --- | --- |
| Вариант | Метод разрешения коллизий | Хеш-функция |
| 1 | Линейное пробирование | Свертка |
| 2 | Квадратичное пробирование | Метод деления |
| 3 | Двойное хеширование | Середина квадрата |
| 4 | Метод цепочек | Метод умножения |
| 5 | Линейное пробирование | Метод деления |
| 6 | Квадратичное пробирование | Свертка |
| 7 | Двойное хеширование | Метод деления |
| 8 | Метод цепочек | Середина квадрата |
| 9 | Линейное пробирование | Метод умножения |
| 10 | Квадратичное пробирование | Свертка |
| 11 | Двойное хеширование | Метод умножения |
| 12 | Метод цепочек | Свертка |
| 13 | Линейное пробирование | Метод деления |
| 14 | Квадратичное пробирование | Середина квадрата |
| 15 | Двойное хеширование | Метод умножения |

**3. Содержание отчета по лабораторной работе**

- наименование лабораторной работы и ее цель;

- задание на лабораторную работу согласно варианту;

- листинг программы для вычисления хеш-функции строкового ключа;

- хеш-таблица и коэффициент ее заполнения;

- выводы.

1. Контрольные вопросы
2. Что такое хеширование?
3. Что такое коллизии при хешировании?
4. Какие существуют методы разрешения коллизий?
5. Какие требования предъявляются к хеш-функциям?
6. Какие существуют методы хеширования строковых ключей?