25.11.2024

Хэш-функции и хеш-таблицы. Указатель на функцию.

Филиппов Михаил Витальевич

m.filippov@g.nsu.ru

89232283872

Императивное программирование, 2024-2025

Новосибирский государственный университет

*НАСТОЯЩАЯ НАУКА

Давайте познакомимся

Филиппов Михаил Витальевич

- Окончил магистратуру ФФ НГУ
- Окончил аспирантуру ИТ СО РАН
- Являюсь м.н.с. ИТ СО РАН
- 7+ лет опыт в программировании C/C++

Agenda

Хеш-функции и хеш-таблицы

75 минут

Указатель на функцию

15 минут

Agenda

Хеш-функции и хеш-таблицы

75 минут

Указатель на функцию

15 минут

Введение

Мы знаем два алгоритма поиска: линейный поиск и бинарный поиск. Линейный поиск имеет время выполнения, пропорциональное O(n), в то время как бинарный поиск занимает время, пропорциональное O(log n), где n — количество элементов в массиве. Двоичный поиск и деревья бинарного поиска являются эффективными алгоритмами для поиска элемента. Но что, если мы хотим выполнить операцию поиска за время, пропорциональное O(1)? Другими словами, есть ли способ выполнить поиск в массиве за постоянное время, независимо от его размера?

Ключ	Массив записей сотрудников
Ключ 0 -> [0]	Запись сотрудника с Emp_ID 0
Ключ 1 -> [1]	Запись сотрудника с Emp_ID 1
Ключ 2 -> [2]	Запись сотрудника с Emp_ID 2
……	……
……	……
Ключ 98 -> [98]	Запись сотрудника с Emp_ID 98
Ключ 99 -> [99]	Запись сотрудника с Emp_ID 99

Введение

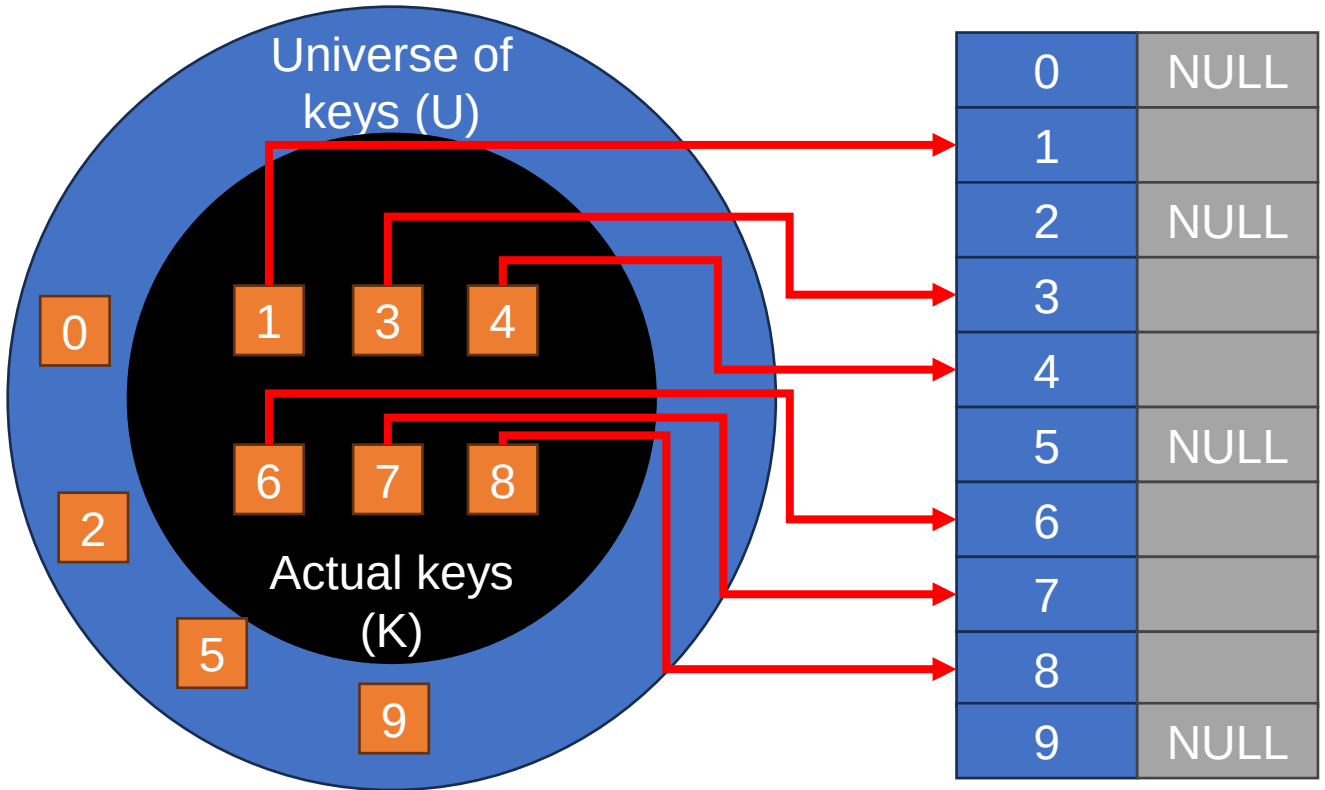
Предположим, что та же компания использует пятизначный Emp_ID в качестве первичного ключа. В этом случае значения ключей будут находиться в диапазоне от 00000 до 99999. Если мы хотим использовать ту же технику, что и выше, нам понадобится массив размером 100 000, из которых будут использоваться только 100 элементов.

Ключ	Массив записей сотрудников
Ключ 00000 -> [0]	Запись сотрудника с Emp_ID 0
……..	…….
Ключ n -> [n]	Запись сотрудника с Emp_ID n
……..	…….
Ключ 98 -> [99998]	Запись сотрудника с Emp_ID 98998
Ключ 99 -> [99999]	Запись сотрудника с Emp_ID 99999

Нецелесообразно тратить так много места для хранения только для того, чтобы гарантировать, что запись каждого сотрудника находится в уникальном и предсказуемом месте.

Хэш-таблицы

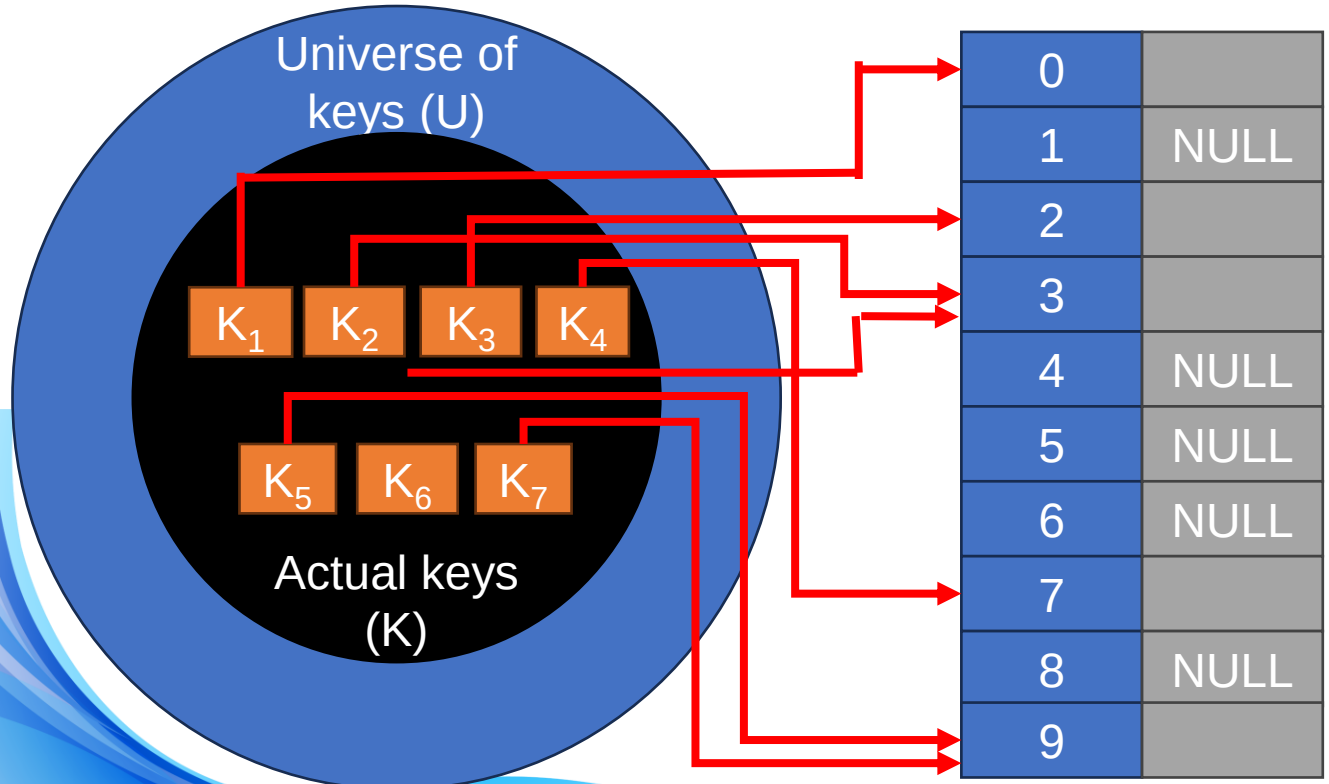
Хэш-таблица — это структура данных, в которой ключи сопоставляются с позициями массива с помощью хэш-функции. В обсуждаемом здесь примере мы будем использовать хэш-функцию, которая извлекает последние две цифры ключа. Поэтому мы сопоставляем ключи с позициями массива или индексами массива. Значение, хранящееся в хэш-таблице, можно найти за время O(1) с помощью хэш-функции, которая генерирует адрес из ключа (создавая индекс массива, в котором хранится значение).



Эта концепция полезна, когда общая вселенная ключей невелика и когда большинство ключей фактически используются из всего набора ключей. Это эквивалентно нашему первому примеру, где есть 100 ключей для 100 сотрудников. Однако, когда набор K фактически используемых ключей меньше, чем вселенная ключей (U), хэш-таблица потребляет меньше места для хранения. Требование к хранению для хэш-таблицы составляет O(k), где k — это количество фактически используемых ключей.

Хэш-таблицы

В хэш-таблице элемент с ключом k хранится в индексе h(k), а не k. Это означает, что хэш-функция h используется для вычисления индекса, в котором будет храниться элемент с ключом k. Этот процесс сопоставления ключей с соответствующими позициями (или индексами) в хэш-таблице называется хэшированием.



Обратите внимание, что ключи k2 и k6 указывают на одну и ту же ячейку памяти. Это известно как коллизия. То есть, когда два или более ключей сопоставлены с одной и той же ячейкой памяти, говорят, что происходит коллизия. Аналогично, ключи k5 и k7 также сталкиваются. Основная цель использования хэш-функции — сократить диапазон индексов массива, которые необходимо обработать. Таким образом, вместо значений U нам нужны только значения K, тем самым уменьшая объем требуемого пространства для хранения.

Хеш-функции

Хеш-функция — это математическая формула, которая при применении к ключу создает целое число, которое можно использовать в качестве индекса для ключа в хеш-таблице. Основная цель хеш-функции заключается в том, чтобы элементы были распределены относительно, случайно и равномерно. Она создает уникальный набор целых чисел в некотором подходящем диапазоне, чтобы уменьшить количество коллизий. На практике не существует хеш-функции, которая полностью устраняет коллизии. Хорошая хеш-функция может только минимизировать количество коллизий, равномерно распределяя элементы по всему массиву.

Свойства хорошей хеш-функции

Низкая стоимость Стоимость выполнения хеш-функции должна быть небольшой, чтобы использование метода хеширования стало предпочтительным по сравнению с другими подходами. Например, если алгоритм бинарного поиска может выполнять поиск элемента в отсортированной таблице из n элементов с помощью $\log_2 n$ сравнений ключей, то хеш-функция должна обходиться дешевле, чем выполнение $\log_2 n$ сравнений ключей.

Детерминизм Процедура хеширования должна быть детерминированной. Это означает, что для заданного входного значения должно быть сгенерировано одно и то же значение хеширования. Однако этот критерий исключает функции хеширования, которые зависят от внешних переменных параметров (таких как время суток) и от адреса памяти хешируемого объекта (поскольку адрес объекта может измениться во время обработки).

Равномерность Хорошая функция хеширования должна отображать ключи как можно более равномерно по всему диапазону выходных данных. Это означает, что вероятность генерации каждого значения хеширования в диапазоне выходных данных должна быть примерно одинаковой. Свойство равномерности также минимизирует количество коллизий.

Различные функции хеширования - метод деления

Это самый простой метод хэширования целого числа x. Этот метод делит x на M, а затем использует полученный остаток. В этом случае хэш-функция может быть задана как

h(x) = x mod M

Метод деления довольно хорош практически для любого значения M, и поскольку он требует только одной операции деления, метод работает очень быстро. Однако следует проявить особую осторожность, чтобы выбрать подходящее значение для M.

Пример кода

```
int const M = 97; // простое число
int h (int x) {
   return (x % M);
}
```

Пример: хеш-значения ключей 1234 и 5462.

Решение

При M = 97 хеш-значения можно вычислить следующим образом:

h(1234) = 1234 % 97 = 70

h(5642) = 5642 % 97 = 16

Различные функции хеширования - метод умножения

Шаги, используемые в методе умножения, следующие:

Шаг 1: Выберите константу A, такую, что 0 < A < 1.

Шаг 2: Умножьте ключ k на A.

Шаг 3: Извлеките дробную часть kA.

Шаг 4: Умножьте результат шага 3 на размер хеш-таблицы (m).

Следовательно, хэш-функция может быть задана как:

h(k) = m (kA mod 1)

Кнут предположил, что лучшим выбором A является (sqrt5 – 1) /2 = 0,6180339887

пример

Учитывая хеш-таблицу размером 1000, сопоставьте ключ 12345 с соответствующим местом в хеш-таблице.

Решение

Мы будем использовать:

A = 0,618033, m = 1000 и k = 12345

h(12345) = 1000 (12345 × 0,618033 mod 1)

h(12345) = 1000 (7629,617385 mod 1)

h(12345) = 1000 (0,617385)

h(12345) = 617,385

h(12345) = 617

Различные функции хеширования - метод середины квадрата

Метод середины квадрата — это хорошая хеш-функция, которая работает в два этапа:

Шаг 1: Возвести в квадрат значение ключа. То есть найти k^2.

Шаг 2: Извлечь средние r цифр результата, полученного на Шаге 1.

Алгоритм работает хорошо, потому что большинство или все цифры значения ключа вносят вклад в результат. Хеш-функция может быть задана как:

$h(k) = s$

где s получается путем выбора r цифр из k^2.

пример

Рассчитайте хеш-значение для ключей 1234 и 5642 с помощью метода середины квадрата. Хеш-таблица имеет 100 ячеек памяти.

Решение

Обратите внимание, что хеш-таблица имеет 100 ячеек памяти, индексы которых варьируются от 0 до 99. Это означает, что для сопоставления ключа с ячейкой в хеш-таблице требуется всего две цифры, поэтому r = 2.

Когда k = 1234, k^2 = 1522756, h (1234) = 27

Когда k = 5642, k^2 = 31832164, h (5642) = 21

Обратите внимание, что выбраны 3-я и 4-я цифры, начиная справа.

Различные функции хеширования - метод сворачивания

Метод сворачивания работает в следующие два шага:

Шаг 1: Разделите значение ключа на несколько частей. То есть, разделите k на части k_1, k_2, ..., k_n, где каждая часть имеет одинаковое количество цифр, за исключением последней части, которая может иметь меньше цифр, чем другие части.

Шаг 2: Сложите отдельные части. То есть, получите сумму $k_1 + k_2 + ... + k_n$. Значение хэша получается путем игнорирования последнего переноса, если таковой имеется.

пример

Учитывая хэш-таблицу из 100 ячеек, вычислите хэш-значение с помощью метода свертывания для ключей 5678, 321 и 34567.

Решение

Поскольку необходимо адресовать 100 ячеек памяти, мы разобьем ключ на части, каждая из которых (кроме последней) будет содержать две цифры. Значения хэш-функции можно получить следующим образом:

Ключ	5678	321	34567
Части	56 и 78	32 и 1	34, 56 и 7
Сумма	134	33	97
Хэш-значение	34 (134 % 100)	33	97

Универсальное хеширование

Если недоброжелатель будет умышленно выбирать ключи для хеширования с использованием конкретной хеш-функции, то он сможет подобрать n значений, которые будут хешироваться в одну и ту же ячейку таблицы, приводя к среднему времени выборки Θ(n). Таким образом, любая фиксированная хеш-функция становится уязвимой, и единственный эффективный выход из ситуации – случайный выбор хеш-функции, не зависящий от того, с какими именно ключами ей предстоит работать. Такой подход, который называется ***универсальным хешированием***, гарантирует хорошую производительность в среднем, независимо от того, какие данные будут выбраны упомянутым недоброжелателем.

Главная идея универсального хеширования состоит в случайном выборе хеш-функции из некоторого тщательно отобранного класса функций в начале работы программы. Как и в случае быстрой сортировки, рандомизация гарантирует, что одни и те же входные данные не могут постоянно давать наихудшее поведение алгоритма. В силу рандомизации алгоритм будет работать всякий раз по-разному, даже для одних и тех же входных данных, что гарантирует высокую среднюю производительность для любых входных данных.

Универсальное хеширование

Пусть $\mathcal{H}$ - конечное множество хеш-функций, которые отображают данную совокупность ключей U в диапазон {0, 1, …, m - 1}. Такое множество называется универсальным, если для каждой пары различных ключей k, l ∈ U количество хеш-функций h ∈ $\mathcal{H}$, для которых h(k) = h(l), не превышает |$\mathcal{H}$|/m. Другими словами, при случайном выборе хеш-функции из $\mathcal{H}$ вероятность коллизии между различными ключами k и не превышает вероятности совпадения двух случайным образом выбранных хеш-значений из множества {0, 1, …,m - 1}, которая равна 1/m.

Следующая теорема показывает, что универсальный класс хеш-функций обеспечивает хорошую среднюю производительность. В приведенной теореме n_i, как уже упоминалось, обозначает длину списка T[i].

Теорема

Пусть хеш-функция h, случайным образом выбранная из универсального множества хеш-функций, применяется для хеширования n ключей в таблицу T размером m с использованием для разрешения коллизий метода цепочек. Если ключ k отсутствует в таблице, то математическое ожидание $E[n_{h(k)}]$ длины списка, в который хешируется ключ k, не превышает коэффициента заполнения α = n/m.

Если ключ k находится в таблице, то математическое ожидание $E[n_{h(k)}]$ длины списка, в котором находится ключ к, не превышает 1 + α.

Универсальное хеширование

Доказательство

Заметим, что математическое ожидание вычисляется на множестве выборов функций и не зависит от каких бы то ни было предположений о распределении ключей. Определим для каждой пары различных ключей k и l индикаторную случайную величину $X_{kl} = I\{h(k) = h(l)\}$. Поскольку по определению универсального множества пара ключей вызывает коллизию с вероятностью не выше 1/m, получаем, что $Pr\{h(k) = h(l)\} \leq 1/m$. Следовательно, математическое ожидание $E[X_{kl}] \leq 1/m$. Далее для каждого ключа k определим случайную величину Y_k, которая равна количеству ключей, отличающихся от k и хешируемых в ту же ячейку, что и ключ k, так что

$$Y_k = \sum_{\substack{l \in T \\ l \neq k}} X_{kl}$$

Таким образом, мы имеем

$$E[Y_k] = E\left[\sum_{\substack{l \in T \\ l \neq k}} X_{kl}\right] = \sum_{\substack{l \in T \\ l \neq k}} E[X_{kl}] \leq \sum_{\substack{l \in T \\ l \neq k}} \frac{1}{m}$$

(из линейности математического ожидания)

Оставшаяся часть доказательства зависит от того, находится ли ключ k в таблице T.

- Если $k \notin T$, то $n_{h(k)} = Y_k$ и $|\{l : l \in T \text{ и } l \neq k\}| = n$. Таким образом, $E[n_{h(k)}] = E[Y_k] \leq n/m = \alpha$.
- Если $k \in T$, то, поскольку ключ k находится в списке $T[h(k)]$ и количество Y_k не включает ключ k, мы имеем $n_{h(k)} = Y_k + 1$ и $|\{l : l \in T \text{ и } l \neq k\}| = n - 1$.

  Таким образом, $E[n_{h(k)}] = E[Y_k]+1 \leq (n-1)/m+1 = 1 + \alpha - 1/m < 1 + \alpha$.

Построение универсального класса хеш-функций

Построить такое множество довольно просто, что следует из теории чисел. Начнем с выбора простого числа p, достаточно большого, чтобы все возможные ключи находились в диапазоне от 0 до p - 1 включительно. Пусть $\mathbb{Z}_p$, обозначает множество {0, 1, …, p - 1}, а $\mathbb{Z}_p^*$ - множество {1, 2, …, p - 1}. Поскольку p - простое число, мы можем решать уравнения по модулю p с помощью методов. Из предположения о том, что пространство ключей больше, чем количество ячеек в хеш-таблице, следует, что p > m.

Теперь определим хеш-функцию h_{ab} для любых a ∈ $\mathbb{Z}_p^*$, и b ∈ $\mathbb{Z}_p$, используя линейное преобразование с последующим приведением по модулю p, а затем по модулю m:

$h_{ab}(k) = ((ak + b) \bmod p) \bmod m$.

Например, при p = 17 и m = 6 мы имеем $h_{3,4}(8) = 5$. Семейство всех таких функций образует множество

$\mathcal{H}_{pm} = \{h_{ab}: a \in \mathbb{Z}_p^*, b \in \mathbb{Z}_p\}$.

Каждая хеш-функция h_{ab} отображает $\mathbb{Z}_p$, в $\mathbb{Z}_m$. Этот класс хеш-функций обладает тем свойством, что размер m выходного диапазона произволен и не обязательно представляет собой простое число. Поскольку число a можно выбрать p - 1 способом, а число b – p способами, всего во множестве $\mathcal{H}_{pm}$ содержится p(p - 1) хеш-функций.

Построение универсального класса хеш-функций

Теорема

Класс $\mathcal{H}_{pm}$ хеш-функций, определяемых уравнениями, является универсальным.

Доказательство.

Рассмотрим два различных ключа k и l из $\mathbb{Z}_p$, так что k ≠ l.

Пусть для данной хеш-функции h_{ab}

r = (ak + b) mod p,

s = (al + b) mod p.

Заметим сначала, что r ≠ s. Почему? Обратите внимание, что

r - s ≡ a(k - l) (mod p) .

Отсюда следует, что r ≠ s, поскольку p - простое число, и как a, так и (k - l) ненулевые по модулю p, так что и их произведение должно быть ненулевым по модулю p. Следовательно, вычисление любой хеш-функции $h_{ab} \in \mathcal{H}_{pm}$ отображает различные входные значения k и на различные значения r и s по модулю p; так что коллизии "по модулю p" отсутствуют. Более того, каждый из p(p - 1) возможных выборов пар (a,b) с a ≠ 0 дает различные пары (r, s) с r ≠ s, поскольку по заданным т и s можно найти a и b:

$a = ((r - s)((k - l)^{-1} \bmod p)) \bmod p$,

b = (r - ak) mod p,

где $((k - l)^{-1} \bmod p)$ обозначает единственное мультипликативное обратное по модулю p значения k - l.

Построение универсального класса хеш-функций

Поскольку имеется только p(p - 1) возможных пар (т, s), таких что m ≠ s, то имеется взаимно однозначное соответствие между парами (a, b) с a ≠ 0 и парами (r, s), в которых r ≠ s. Таким образом, для любой данной
пары входных значений к и при равномерном случайном выборе пары (a, b) из $\mathbb{Z}_p^* \times \mathbb{Z}_p$, получаемая в результате пара (r, s), может быть, с равной вероятностью любой из пар с отличающимися по модулю p значениями.

Отсюда можно заключить, что вероятность того, что различные ключи и l приводят к коллизии, равна вероятности того, что m ≡ s (mod m) при случайном выборе отличающихся по модулю p значений т и s. Для данного значения т имеется p - 1 возможных значений s. При этом число значений s, таких, что s ≠ r и s = r (mod m), не превышает
[p/m] - 1 < ((p+ m - 1)/m) – 1 = (p - 1)/m .

Вероятность того, что s приводит к коллизии с m при приведении по модулю m, не превышает ((p - 1)/m)/(p - 1) = 1/m.

Следовательно, для любой пары различных значений k,l ∈ $\mathbb{Z}_p$ Pr{$h_{ab}(k) = h_{ab}(l)$} 1/m, так что множество хеш-функций $\mathcal{H}_{pm}$ является универсальным.

Коллизии

Коллизии возникают, когда хэш-функция сопоставляет два разных ключа с одним и тем же местом. Очевидно, что две записи не могут храниться в одном и том же месте. Поэтому применяется метод, используемый для решения проблемы коллизий, также называемый методом разрешения коллизий.

Два самых популярных метода разрешения коллизий:

1. Открытая адресация
2. Цепочка

Разрешение коллизий с помощью открытой адресации

После возникновения коллизии открытая адресация или закрытое хеширование вычисляет новые позиции с помощью последовательности зондов, и следующая запись сохраняется в этой позиции. В этом методе все значения сохраняются в хеш-таблице. Хеш-таблица содержит два типа значений: контрольные значения (например, –1) и значения данных. Наличие контрольного значения указывает на то, что местоположение в данный момент не содержит значения данных, но может использоваться для хранения значения. Когда ключ сопоставляется с определенным местоположением памяти, то проверяется хранящееся в нем значение. Если он содержит контрольное значение, то местоположение свободно, и в нем можно сохранить значение данных. Однако если местоположение уже имеет некоторое сохраненное значение данных, то другие слоты систематически проверяются в прямом направлении, чтобы найти свободный слот. Если не найдено ни одного свободного места, то возникает состояние OVERFLOW. Процесс проверки мест памяти в хэш-таблице называется зондированием. Метод открытой адресации может быть реализован с использованием линейного зондирования, квадратичного зондирования, двойного хеширования и повторного хеширования.

Линейное зондирование

Самый простой подход к разрешению коллизии — линейное зондирование. В этом методе, если значение уже сохранено в месте, сгенерированном h(k), то для разрешения коллизии используется следующая хеш-функция:
h(k, i) = [h’(k) + i] mod m
Где m — размер хэш-таблицы, h’(k) = (k mod m), а i — номер зонда, который изменяется от 0 до m – 1. Следовательно, для заданного ключа k сначала зондируется место, сгенерированное [h’(k) mod m], поскольку в первый раз i = 0. Если ячейка свободна, значение сохраняется в ней, в противном случае второй зонд генерирует адрес ячейки, заданный как [h’(k) + 1]mod m. Аналогично, если ячейка занята, то последующие зонды генерируют адрес как [h’(k) + 2] mod m, [h’(k) + 3] mod m, [h’(k) + 4] mod m, [h’(k) + 5] mod m и так далее, пока не будет найдена свободная ячейка.

Примечание.

Линейное зондирование известно своей простотой. Когда нам нужно сохранить значение, мы пробуем слоты: [h’(k)] mod m, [h’(k) + 1]mod m, [h’(k) + 2]mod m, [h’(k) + 3]mod m, [h’(k) + 4]mod m, [h’(k) + 5]mod m и так далее, пока не будет найдено свободное место.

Линейное зондирование - пример

пример

Рассмотрим хеш-таблицу размером 10. Используя линейное зондирование, вставьте ключи 72, 27, 36, 24, 63, 81, 92 и 101 в таблицу. Пусть h’(k) = k mod m, m = 10

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Шаг 1: Ключ = 72

h(72, 0) = (72 % 10 + 0) % 10 = 2 % 10 = 2

Так как T[2] свободен, вставьте ключ 72 в это место.

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	-1	-1	-1	-1

Шаг 2: Ключ = 27

h(27, 0) = (27 % 10 + 0) % 10 = 7 % 10 = 7

Так как T[7] свободен, вставьте ключ 27 в это место.

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	-1	27	-1	-1

Линейное зондирование - пример

пример

Рассмотрим хеш-таблицу размером 10. Используя линейное зондирование, вставьте ключи 72, 27, 36, 24, 63, 81, 92 и 101 в таблицу. Пусть h’(k) = k mod m, m = 10

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	-1	27	-1	-1

Шаг 3: Ключ = 36

h(36, 0) = (36 % 10 + 0) % 10 = 6 % 10 = 6

Так как T[6] свободен, вставьте ключ 36 в это место.

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	36	27	-1	-1

Шаг 4: Ключ = 24

h(24, 0) = (24 % 10 + 0) % 10 = 4 % 10 = 4

Так как T[4] свободен, вставьте ключ 24 в это место.

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	24	-1	36	27	-1	-1

Линейное зондирование - пример

пример

Рассмотрим хеш-таблицу размером 10. Используя линейное зондирование, вставьте ключи 72, 27, 36, 24, 63, 81, 92 и 101 в таблицу. Пусть h’(k) = k mod m, m = 10

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	24	-1	36	27	-1	-1

Шаг 5: Ключ = 63

h(63, 0) = (63 % 10 + 0) % 10 = 3 % 10 = 3

Так как T[3] свободен, вставьте ключ 63 в это место.

0	1	2	3	4	5	6	7	8	9
-1	-1	72	63	24	-1	36	27	-1	-1

Шаг 6: Ключ = 81

h(81, 0) = (81 % 10 + 0) % 10 = 1 % 10 = 1

Так как T[1] свободен, вставьте ключ 81 в это место.

0	1	2	3	4	5	6	7	8	9
-1	81	72	63	24	-1	36	27	-1	-1

Линейное зондирование - пример

пример

Рассмотрим хеш-таблицу размером 10. Используя линейное зондирование, вставьте ключи 72, 27, 36, 24, 63, 81, 92 и 101 в таблицу. Пусть h’(k) = k mod m, m = 10

0	1	2	3	4	5	6	7	8	9
-1	81	72	63	24	-1	36	27	-1	-1

Шаг 7: Ключ = 92

h(92, 0) = (92 % 10 + 0) % 10 = 2 % 10 = 2

Теперь T[2] занято, поэтому мы не можем сохранить ключ 92 в T[2]. Поэтому попробуйте еще раз для следующего местоположения. Таким образом, на этот раз зонд, i = 1.

h(92, 1) = (92 % 10 + 1) % 10 = (2 + 1) % 10 = 3

Теперь T[3] занято, поэтому мы не можем сохранить ключ 92 в T[3]. Поэтому попробуйте еще раз для следующего местоположения. Таким образом, на этот раз зонд, i = 2.

h(92, 2) = (92 % 10 + 2) % 10 = (2 + 2) % 10 = 4

Теперь T[4] занято, поэтому мы не можем сохранить ключ 92 в T[4]. Поэтому попробуем снова для следующего местоположения. Таким образом, на этот раз зондируем i = 3.

h(92, 2) = (92 % 10 + 3) % 10 = (2 + 3) % 10 = 5

Так как T[5] свободен, вставьте ключ 92 в это место.

0	1	2	3	4	5	6	7	8	9
-1	81	72	63	24	92	36	27	-1	-1

Линейное зондирование - пример

пример

Рассмотрим хеш-таблицу размером 10. Используя линейное зондирование, вставьте ключи 72, 27, 36, 24, 63, 81, 92 и 101 в таблицу. Пусть h’(k) = k mod m, m = 10

0	1	2	3	4	5	6	7	8	9
-1	81	72	63	24	-1	36	27	-1	-1

Шаг 8: Ключ = 101

h(101, 0) = (101 % 10 + 0) % 10 = 1 % 10 = 1

Теперь T[1] занято, поэтому мы не можем сохранить ключ 101 в T[1]. Поэтому попробуйте еще раз для следующего местоположения. Таким образом, на этот раз зонд, i = 1.

h(101, 1) = (101 % 10 + 1) % 10 = (2 + 1) % 10 = 2

T[2] также занято, поэтому мы не можем сохранить ключ в этом расположении. Процедура будет повторяться до тех пор, пока хэш-функция не сгенерирует адрес ячейки 8, которая является свободной и может использоваться для хранения в ней значения.

0	1	2	3	4	5	6	7	8	9
-1	81	72	63	24	92	36	27	101	-1

Поиск значения с использованием линейного зондирования

Процедура поиска значения в хэш-таблице такая же, как и для сохранения значения в хэш-таблице. При поиске значения в хэш-таблице индекс массива пересчитывается, а ключ элемента, хранящегося в этой ячейке, сравнивается со значением, которое необходимо найти. Если совпадение найдено, то операция поиска считается успешной. Время поиска в этом случае задается как O(1). Если ключ не совпадает, то функция поиска начинает последовательный поиск массива, который продолжается до тех пор, пока:

- значение не будет найдено, или
- функция поиска не встретит свободную ячейку в массиве, что указывает на то, что значение отсутствует, или
- функция поиска завершается, поскольку достигает конца таблицы, а значение отсутствует.

В худшем случае операция поиска может потребовать выполнения n–1 сравнений, а время выполнения алгоритма поиска может занять O(n) времени. Худший случай возникнет, когда после сканирования всех n–1 элементов значение либо присутствует в последнем месте, либо отсутствует в таблице. Таким образом, мы видим, что с увеличением числа коллизий расстояние между индексом массива, вычисленным хэш-функцией, и фактическим местом расположения элемента увеличивается, тем самым увеличивая время поиска.

Плюсы и минусы

Линейное зондирование находит пустое место, выполняя линейный поиск в массиве, начиная с позиции h(k). Хотя алгоритм обеспечивает хорошее кэширование памяти за счет хорошей локальности ссылок, недостатком этого алгоритма является то, что он приводит к кластеризации, и, таким образом, существует более высокий риск большего количества коллизий, где одно столкновение уже произошло. Производительность линейного зондирования чувствительна к распределению входных значений. По мере заполнения хэш-таблицы формируются кластеры последовательных ячеек, и время, необходимое для поиска, увеличивается с размером кластера. В дополнение к этому, когда новое значение должно быть вставлено в таблицу в позицию, которая уже занята, это значение вставляется в конец кластера, что снова увеличивает длину кластера. Обычно вставка выполняется между двумя кластерами, которые разделены одним свободным местом. Но при линейном зондировании больше шансов, что последующие вставки также окажутся в одном из кластеров, тем самым потенциально увеличивая длину кластера на величину, намного большую, чем единица. Чем больше число столкновений, тем больше зондов, необходимых для поиска свободного места, и тем ниже производительность. Это явление называется первичной кластеризацией. Чтобы избежать первичной кластеризации, используются другие методы, такие как квадратичное зондирование и двойное хеширование.

Квадратичное зондирование

В этой технике, если значение уже сохранено в месте, сгенерированном h(k), то для разрешения коллизии используется следующая хэш-функция:

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

где m — размер хэш-таблицы, h’(k) = (k mod m), i — номер зонда, который изменяется от 0 до m–1, а c_1 и c_2 — константы, такие, что c_1 и $c_2 \neq 0$. Квадратичное зондирование устраняет первичное явление кластеризации линейного зондирования, поскольку вместо выполнения линейного поиска выполняется квадратичный поиск. Для заданного ключа k сначала зондируется место, сгенерированное h’(k) mod m. Если место свободно, значение сохраняется в нем, в противном случае последующие зондируемые места смещаются на коэффициенты, которые зависят квадратичным образом от номера зонда i. Хотя квадратичное зондирование работает лучше, чем линейное, для максимального использования хэш-таблицы необходимо ограничить значения c_1, c_2 и m.

Квадратичное зондирование - пример

пример

Рассмотрим хеш-таблицу размером 10. Используя квадратичное зондирование, вставьте ключи 72, 27, 36, 24, 63, 81 и 101 в таблицу. Возьмем $c_1 = 1$ и $c_2 = 3$. Решение Пусть $h'(k) = k \bmod m$, $m = 10$ Первоначально хеш-таблица может быть задана как:

Имеем,

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

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

Шаг 1: Ключ = 72

h(72, 0) = (72 % 10 + 1 × 0 + 3 × 0) % 10 = 2 % 10 = 2

Так как T[2] свободен, вставьте ключ 72 в это место.

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	-1	-1	-1	-1

Шаг 2: Ключ = 27

h(27, 0) = (27 % 10 + 1 × 0 + 3 × 0) % 10 = 7 % 10 = 7

Так как T[7] свободен, вставьте ключ 27 в это место.

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	-1	27	-1	-1

Квадратичное зондирование - пример

пример

Рассмотрим хеш-таблицу размером 10. Используя квадратичное зондирование, вставьте ключи 72, 27, 36, 24, 63, 81 и 101 в таблицу. Возьмем $c_1 = 1$ и $c_2 = 3$.
Решение Пусть h'(k) = k mod m, m = 10

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	-1	27	-1	-1

Шаг 3: Ключ = 36
h(36, 0) = (36 % 10 + 1 × 0 + 3 × 0) % 10 = 6 % 10 = 6
Так как T[6] свободен, вставьте ключ 36 в это место.

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	36	27	-1	-1

Шаг 4: Ключ = 24
h(24, 0) = (24 % 10 + 1 × 0 + 3 × 0) % 10 = 4 % 10 = 4
Так как T[4] свободен, вставьте ключ 24 в это место.

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	24	-1	36	27	-1	-1

Квадратичное зондирование - пример

пример

Рассмотрим хеш-таблицу размером 10. Используя квадратичное зондирование, вставьте ключи 72, 27, 36, 24, 63, 81 и 101 в таблицу. Возьмем $c_1 = 1$ и $c_2 = 3$.

Решение Пусть h'(k) = k mod m, m = 10

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	24	-1	36	27	-1	-1

Шаг 5: Ключ = 63

h(63, 0) = (63 % 10 + 1 × 0 + 3 × 0) % 10 = 3 % 10 = 3

Так как T[3] свободен, вставьте ключ 63 в это место.

0	1	2	3	4	5	6	7	8	9
-1	-1	72	63	24	-1	36	27	-1	-1

Шаг 6: Ключ = 81

h(81, 0) = (81 % 10 + 1 × 0 + 3 × 0) % 10 = 1 % 10 = 1

Так как T[1] свободен, вставьте ключ 81 в это место.

0	1	2	3	4	5	6	7	8	9
-1	81	72	63	24	-1	36	27	-1	-1

Квадратичное зондирование - пример

пример

Рассмотрим хеш-таблицу размером 10. Используя квадратичное зондирование, вставьте ключи 72, 27, 36, 24, 63, 81 и 101 в таблицу. Возьмем $c_1 = 1$ и $c_2 = 3$.
Решение Пусть h'(k) = k mod m, m = 10

0	1	2	3	4	5	6	7	8	9
-1	81	72	63	24	-1	36	27	-1	-1

Шаг 7: Ключ = 101
h(101, 0) = (101 % 10 + 1 × 0 + 3 × 0) % 10 = 1 % 10 = 1
Теперь T[1] занято, поэтому мы не можем сохранить ключ 92 в T[2]. Поэтому попробуйте еще раз для следующего местоположения. Таким образом, на этот раз зонд, i = 1.
h(101, 1) = (101 % 10 + 1 × 1 + 3 × 1) % 10 = (1 + 1 + 3) % 10 = 5
Поскольку T[5] пуст, вставьте ключ 101 в T[5].

0	1	2	3	4	5	6	7	8	9
-1	81	72	63	24	101	36	27	-1	-1

Поиск значения с использованием квадратичного зондирования

При поиске значения с использованием техники квадратичного зондирования индекс массива пересчитывается, и ключ элемента, сохраненного в этом месте, сравнивается со значением, которое необходимо найти. Если желаемое значение ключа совпадает со значением ключа в этом месте, то элемент присутствует в хэш-таблице, и поиск считается успешным. В этом случае время поиска задается как O(1). Однако если значение не совпадает, то функция поиска начинает последовательный поиск в массиве, который продолжается до тех пор, пока:

- значение не будет найдено, или
- функция поиска не встретит пустое место в массиве, что указывает на то, что значение отсутствует, или
- функция поиска завершает работу, поскольку достигает конца таблицы, а значение отсутствует.

В худшем случае операция поиска может потребовать n–1 сравнений, а время выполнения алгоритма поиска может быть O(n). Худший случай возникнет, когда после сканирования всех n–1 элементов значение либо присутствует в последнем месте, либо отсутствует в таблице. Таким образом, мы видим, что с увеличением числа коллизий расстояние между индексом массива, вычисленным хэш-функцией, и фактическим местом элемента увеличивается, тем самым увеличивая время поиска.

Плюсы и минусы

Квадратичный зонд решает основную проблему кластеризации, которая существует в методе линейного зондирования. Квадратичный зонд обеспечивает хорошее кэширование памяти, поскольку сохраняет некоторую локальность ссылок. Но линейное зондирование лучше справляется с этой задачей и обеспечивает лучшую производительность кэша. Одним из основных недостатков квадратичного зондирования является то, что последовательность последовательных зондирований может исследовать только часть таблицы, и эта часть может быть довольно маленькой. Если это произойдет, то мы не сможем найти пустое место в таблице несмотря на то, что таблица никоим образом не заполнена. В примере попробуйте вставить ключ 92, и вы столкнетесь с этой проблемой. Хотя квадратичное зондирование свободно от первичной кластеризации, оно все равно подвержено тому, что известно как вторичная кластеризация. Это означает, что если между двумя ключами происходит столкновение, то для обоих будет использоваться одна и та же последовательность зондирования. При квадратичном зондировании вероятность множественных столкновений увеличивается по мере заполнения таблицы. Такая ситуация обычно возникает, когда хэш-таблица более чем заполнена. Квадратичное зондирование широко применяется в Berkeley Fast File System для выделения свободных блоков.

Двойное хеширование

Для начала, двойное хеширование использует одно хеш-значение, а затем многократно переходит на интервал вперед, пока не будет достигнуто пустое место. Интервал определяется с помощью второй, независимой хеш-функции, отсюда и название двойное хеширование. В двойном хешировании мы используем две хеш-функции, а не одну. Хэш-функция в случае двойного хеширования может быть задана как:

$h(k, i) = [h_1(k) + ih_2(k)] \bmod m$

где m — размер хеш-таблицы, $h_1(k)$ и $h_2(k)$ — две хеш-функции, заданные как $h_1(k) = k \bmod m$, $h_2(k) = k \bmod m'$, i — номер пробы, который изменяется от 0 до m–1, а m' выбирается меньше m. Мы можем выбрать m' = m–1 или m–2. Когда нам нужно вставить ключ k в хеш-таблицу, мы сначала проверяем местоположение, заданное применением $[h_1(k) \bmod m]$, потому что во время первого зондирования i = 0. Если местоположение свободно, ключ вставляется в него, в противном случае последующие зондирования генерируют местоположения, которые находятся со смещением $[h_2(k) \bmod m]$ от предыдущего местоположения. Поскольку смещение может меняться с каждым зондом в зависимости от значения, сгенерированного второй хеш-функцией, производительность двойного хеширования очень близка к производительности идеальной схемы равномерного хеширования.

Двойное хеширование - пример

пример

Рассмотрим хеш-таблицу размером = 10. Используя двойное хеширование, вставьте ключи 72, 27, 36, 24, 63, 81, 92 и 101 в таблицу. Возьмем $h_1 = (k \bmod 10)$ и $h_2 = (k \bmod 8)$. Пусть m = 10

Изначально хеш-таблица может быть задана как:

Имеем,

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

$h(k, i) = [h_1(k) + ih_2(k)] \bmod m$

Шаг 1: Ключ = 72

h(72, 0) = (72 % 10 + 0 × 72 % 8) % 10 = 2 % 10 = 2

Так как T[2] свободен, вставьте ключ 72 в это место.

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	-1	-1	-1	-1

Шаг 2: Ключ = 27

h(27, 0) = (27 % 10 + 1 × 0 + 0 × 27 % 8) % 10 = 7 % 10 = 7

Так как T[7] свободен, вставьте ключ 27 в это место.

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	-1	27	-1	-1

Двойное хеширование - пример

пример

Рассмотрим хеш-таблицу размером = 10. Используя двойное хеширование, вставьте ключи 72, 27, 36, 24, 63, 81, 92 и 101 в таблицу. Возьмем h_1 = (k mod 10) и h_2 = (k mod 8). Пусть m = 10

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	-1	27	-1	-1

Шаг 3: Ключ = 36

h(36, 0) = (36 % 10 + 0 × 36 % 8) % 10 = 6 % 10 = 6

Так как T[6] свободен, вставьте ключ 36 в это место.

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	36	27	-1	-1

Шаг 4: Ключ = 24

h(24, 0) = (24 % 10 + 0 × 24 % 8) % 10 = 4 % 10 = 4

Так как T[4] свободен, вставьте ключ 24 в это место.

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	24	-1	36	27	-1	-1

Двойное хеширование - пример

пример

Рассмотрим хеш-таблицу размером = 10. Используя двойное хеширование, вставьте ключи 72, 27, 36, 24, 63, 81, 92 и 101 в таблицу. Возьмем h_1 = (k mod 10) и h_2 = (k mod 8). Пусть m = 10

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	24	-1	36	27	-1	-1

Шаг 5: Ключ = 63

h(63, 0) = (63 % 10 + 0 × 63 % 8) % 10 = 3 % 10 = 3

Так как T[3] свободен, вставьте ключ 63 в это место.

0	1	2	3	4	5	6	7	8	9
-1	-1	72	63	24	-1	36	27	-1	-1

Шаг 6: Ключ = 81

h(81, 0) = (81 % 10 + 0 × 81 % 8) % 10 = 1 % 10 = 1

Так как T[1] свободен, вставьте ключ 81 в это место.

0	1	2	3	4	5	6	7	8	9
-1	81	72	63	24	-1	36	27	-1	-1

Двойное хеширование - пример

пример

Рассмотрим хеш-таблицу размером = 10. Используя двойное хеширование, вставьте ключи 72, 27, 36, 24, 63, 81, 92 и 101 в таблицу. Возьмем h_1 = (k mod 10) и h_2 = (k mod 8). Пусть m = 10

0	1	2	3	4	5	6	7	8	9
-1	81	72	63	24	-1	36	27	-1	-1

Шаг 7: Ключ = 92

h(92, 0) = (92 % 10 + 0 × 92 % 8) % 10 = 2 % 10 = 2

Теперь T[2] занято, поэтому мы не можем сохранить ключ 92 в T[2]. Поэтому попробуйте еще раз для следующего местоположения. Таким образом, на этот раз зонд, i = 1.

h(92, 1) = (92 % 10 + 1 × 92 % 8) % 10 = (2 + 4) % 10 = 6

Теперь T[6] занято, поэтому мы не можем сохранить ключ 92 в T[6]. Поэтому попробуйте еще раз для следующего местоположения. Таким образом, на этот раз зонд, i = 2.

Так как T[0] свободен, вставьте ключ 92 в это место.

0	1	2	3	4	5	6	7	8	9
92	81	72	63	24	-1	36	27	-1	-1

Двойное хеширование - пример

пример

Рассмотрим хеш-таблицу размером = 10. Используя двойное хеширование, вставьте ключи 72, 27, 36, 24, 63, 81, 92 и 101 в таблицу. Возьмем h_1 = (k mod 10) и h_2 = (k mod 8). Пусть m = 10

0	1	2	3	4	5	6	7	8	9
92	81	72	63	24	-1	36	27	-1	-1

Шаг 8: Ключ = 101

h(101, 0) = (101 % 10 + 0 × 101 % 8) % 10 = 1 % 10 = 1

Теперь T[1] занято, поэтому мы не можем сохранить ключ 101 в T[1]. Поэтому попробуйте еще раз для следующего местоположения. Таким образом, на этот раз зонд, i = 1.

h(101, 1) = (101 % 10 + 1 × 101 % 8) % 10 = (1 + 5) % 10 = 6

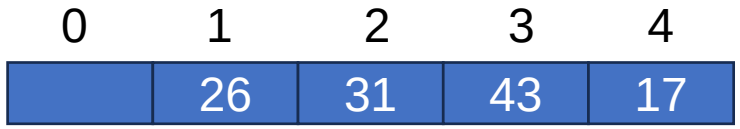
T[6] также занято, поэтому мы не можем сохранить ключ в этом расположении. Вы увидите, что нам нужно выполнить зондирование много раз, чтобы вставить ключ 101 в хэш-таблицу. Хотя двойное хэширование является очень эффективным алгоритмом, оно всегда требует, чтобы m было простым числом. В нашем случае m=10, что не является простым числом, отсюда и ухудшение производительности. Если бы m было равно 11, алгоритм работал бы очень эффективно. Таким образом, можно сказать, что производительность метода чувствительна к значению m.

Плюсы и минусы

Двойное хеширование минимизирует повторяющиеся коллизии и эффекты кластеризации. То есть двойное хеширование свободно от проблем, связанных как с первичной кластеризацией, так и со вторичной кластеризацией.

Перехеширование

Когда хеш-таблица почти заполнена, количество коллизий увеличивается, тем самым снижая производительность операций вставки и поиска. В таких случаях лучшим вариантом будет создание новой хеш-таблицы с размером, вдвое превышающим размер исходной хеш-таблицы. Затем все записи в исходной хеш-таблице придется переместить в новую хеш-таблицу. Это делается путем взятия каждой записи, вычисления ее нового значения хеш-функции и последующей вставки ее в новую хеш-таблицу. Хотя перехеширование кажется простым процессом, оно довольно затратно и поэтому не должно выполняться часто. Рассмотрим хеш-таблицу размером 5, приведенную ниже. Используемая хеш-функция — h(x) = x % 5. Перехешируем записи в новую хеш-таблицу.



Обратите внимание, что новая хеш-таблица состоит из 10 ячеек, что вдвое больше размера исходной таблицы.

Теперь перехешируем ключевые значения из старой хеш-таблицы в новую, используя хеш-функцию — h(x) = x % 10

Анализ хеширования с открытой адресацией

Анализ открытой адресации, как и анализ метода цепочек, выполняется с использованием коэффициента заполнения $\alpha = n/m$. Само собой разумеется, при использовании открытой адресации может быть не более одного элемента на ячейку таблицы, так что $n \le m$ и, следовательно, $\alpha \le 1$.

Будем считать, что используется равномерное хеширование. При такой идеализированной схеме последовательность исследований $\langle h(k, 0), h(k, 1), \ldots, h(k, m - 1)\rangle$, используемая для вставки или поиска каждого ключа к, с равной вероятностью является одной из возможных перестановок $(0, 1, \ldots, m - 1)$. Разумеется, с каждым конкретным ключом связана единственная фиксированная последовательность исследований, так что при рассмотрении распределения вероятностей ключей и хеш-функций все последовательности исследований оказываются равновероятными.

Мы проанализируем математическое ожидание количества исследований для хеширования с открытой адресацией в предположении равномерного хеширования и начнем с анализа количества исследований в случае неудачного поиска.

Анализ хеширования с открытой адресацией

Теорема

Математическое ожидание количества исследований при неудачном поиске в хеш-таблице с открытой адресацией и коэффициентом заполнения $\alpha = n/m < 1$ в предположении равномерного хеширования не превышает $1/(1 - \alpha)$.

Доказательство

При неудачном поиске каждая последовательность исследований завершается пустой ячейкой. Определим случайную величину X как количество исследований, выполненных при неудачном поиске, и определим также события ($i = 1, 2, \ldots$), заключающиеся в том, что было выполнено i-е исследование, и оно пришлось на занятую ячейку. Тогда событие $\{X \geq i\}$ представляет собой пересечение событий $A_1 \cap A_2 \cap \ldots \cap A_{i-1}$. Ограничим вероятность $Pr\{X \geq i\}$ путем ограничения вероятности $Pr\{ A_1 \cap A_2 \cap \ldots \cap A_{i-1}\}$.

$Pr\{ A_1 \cap A_2 \cap \ldots \cap A_{i-1}\} = Pr\{A_1\} \cdot Pr\{A_2 \mid A_1\} \cdot Pr\{A_3 \mid A_1 \cap A_2\} \cdot \ldots \cdot$
$Pr\{A_{i-1} \mid A_1 \cap A_2 \cap \ldots \cap A_{i-2}\}$.

Анализ хеширования с открытой адресацией

Доказательство

Поскольку всего имеется n элементов и m ячеек, $Pr\{A_1\}$ = n/m. Вероятность того, что будет выполнено j-е исследование (j > 1) и что оно будет проведено над заполненной ячейкой (при этом первые j - 1 исследований проведены над заполненными ячейками), равна (m – j + 1)/(m - j + 1). Эта вероятность определяется следующим образом: мы должны проверить один из оставшихся (n - (j - 1)) элементов в одной из оставшихся к этому времени (m - (j - 1)) неисследованных ячеек. В соответствии с предположением о равномерном хешировании искомая вероятность равна отношению этих величин. Воспользовавшись тем фактом, что из n < m для всех 0 ≤ j < m следует соотношение (n - j)/(m - j) ≤ n/m, для всех 1 ≤ i ≤ m получаем

$$\Pr\{X \geq i\} = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdots \frac{n-i-2}{m-i-2} \leq \left(\frac{n}{m}\right)^{i-1} = \alpha^{i-1}$$

Анализ хеширования с открытой адресацией

Доказательство

Теперь воспользуемся уравнением для получения границы ожидаемого количества исследований:

$$E[X] = \sum_{i=1}^{\infty} Pr\{X \geq i\} \leq \sum_{i=1}^{\infty} \alpha^{i-1} = \sum_{i=0}^{\infty} \alpha^{i} = \frac{1}{1-\alpha}$$

Полученная граница 1/(1 - α) = 1 + α + $α^2$ + имеет интуитивную интерпретацию. Одно исследование выполняется всегда. С вероятностью, приблизительно равной α, первое исследование проводится над заполненной ячейкой, и требуется выполнение второго исследования. С вероятностью, приблизительно равной $α^2$, две первые ячейки оказываются заполненными, и требуется проведение третьего исследования, и т.д.

Если α - константа, то теорема 1.2 предсказывает, что неудачный поиск выполняется за время O(1). Например, если хеш-таблица заполнена наполовину, то среднее количество исследований при неудачном поиске не превышает 1/(1 - .5) = 2. При заполненности хеш-таблицы на 90% среднее количество исследований не превышает 1/(1 - .9) = 10.

Теорема практически непосредственно дает оценку производительности процедуры Hash-Insert.

Анализ хеширования с открытой адресацией

Следствие 1.2

Вставка элемента в хеш-таблицу с открытой адресацией и коэффициентом заполнения в предположении равномерного хеширования требует в среднем не более 1/(1 - α) исследований.

Доказательство.

Элемент может быть вставлен в хеш-таблицу только в том случае, если в ней есть свободное место, так что α < 1. Вставка ключа требует проведения неудачного поиска, за которым следует размещение ключа в найденной пустой ячейке. Следовательно, математическое ожидание количества исследований не превышает 1/(1 - α).

Вычисление математического ожидания количества исследований при успешном поиске требует немного больше усилий.

Анализ хеширования с открытой адресацией

Теорема

Математическое ожидание количества исследований при удачном поиске в хештаблице с открытой адресацией и коэффициентом заполнения α < 1, в предположении равномерного хеширования и равновероятного поиска любого из ключей, не превышает

$$\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$$

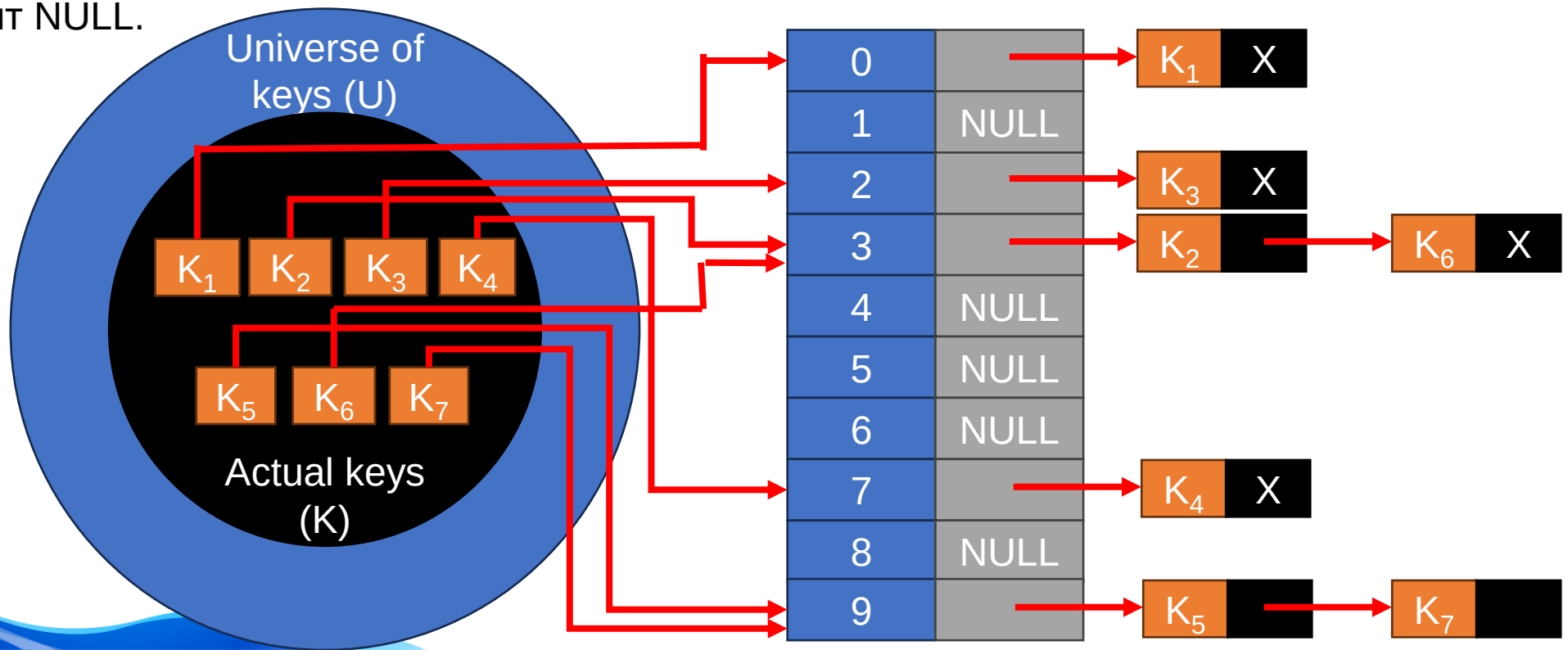
Доказательство. Поиск ключа k выполняется той же последовательностью исследований, что и его вставка. В соответствии со следствием 1.2, если k был (i + 1)-м ключом, вставленным в хеш-таблицу, то математическое ожидание количества проб при поиске k не превышает 1/(1 - i/m) = m/(m - i). Усреднение по всем n ключам в хеш-таблице дает нам среднее количество исследований при удачном поиске:

$$\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i}=\frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i}=\frac{1}{\alpha}\sum_{k=m-n+1}^{m}\frac{1}{k}\le\frac{1}{\alpha}\int_{m-n}^{m}\frac{1}{x}dx=\frac{1}{\alpha}\ln\frac{m}{m-n}=\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$$

В наполовину заполненной хеш-таблице ожидаемое количество исследований при удачном поиске оказывается меньше 1.387. Если хеш-таблица заполнена на 90%, ожидаемое количество исследований меньше 2.559.

Разрешение коллизий путем объединения в цепочку

При объединении в цепочку каждое местоположение в хэш-таблице хранит указатель на связанный список, содержащий все значения ключей, которые были хэшированы в это местоположение. То есть местоположение I в хэш-таблице указывает на заголовок связанного списка всех значений ключей, которые хэшированы в I. Однако если ни одно значение ключа не хэшируется в I, то местоположение I в хэш-таблице содержит NULL.



Разрешение коллизий путем объединения в цепочку

Поиск значения в цепочке хеш-таблиц так же прост, как сканирование связанного списка для записи с заданным ключом. Операция вставки добавляет ключ в конец связанного списка, на который указывает хешированное местоположение. Удаление ключа требует поиска в списке и удаления элемента. Связанные хэш-таблицы со связанными списками широко используются из-за простоты алгоритмов вставки, удаления и поиска ключа. Код для этих алгоритмов точно такой же, как для вставки, удаления и поиска значения в одном связанном списке. В то время как стоимость вставки ключа в связанную хэш-таблицу составляет O(1), стоимость удаления и поиска значения определяется как O(m), где m — количество элементов в списке этого местоположения. Поиск и удаление занимают больше времени, поскольку эти операции сканируют записи выбранного местоположения на предмет нужного ключа.

В худшем случае поиск значения может занять время выполнения O(n), где n — количество значений ключей, хранящихся в связанной хэш-таблице. Этот случай возникает, когда все значения ключей вставляются в связанный список того же местоположения (хэш-таблицы). В этом случае хэш-таблица неэффективна.

```
//Struture of the node
typedef struct node_HT
{
    int value;
    struct node *next;
} node;
```

Код для инициализации цепной хэш-таблицы

```
/* Initializes m location in the chained hash table.
The operation takes a running time of O(m) */
void initializeHashTable(node *hash_table[],
                         int m)
{
    int i;
    for (i = 0; i <= m; i++)
        hash_table[i] = NULL;
}
```

Код для вставки значения

```
/* The element is inserted at the beginning of
the linked list whose pointer to its head is
stored in the location given by h(k). The running
time of the insert operation is O(1), as
the new key value is always added as the first
element of the list irrespective of the size of
the linked list as well as that of the chained
hash table. */
node *insert_value(node *hash_table[], int val)
{
    node *new_node;
    new_node = (node *)malloc(sizeof(node));
    new_node->value = val;
    new_node->next = hash_table[h(val)];
    hash_table[h(val)] = new_node;
}
```

Код для поиска значения

```
/* The element is searched in the linked
list whose pointer to its head is stored
in the location given by h(k). If search is
successful, the function returns a pointer
to the node in the linked list; otherwise
it returns NULL. The worst case running
time of the search operation is given as
order of size of the linked list. */
node *search_value(node *hash_table[], int val)
{
    node *ptr;
    ptr = hash_table[h(val)];
    while ((ptr != NULL) && (ptr->value != val))
        ptr = ptr->next;
    if (ptr->value == val)
        return ptr;
    else
        return NULL;
}
```

Код для поиска значения

```
/* To delete a node from the linked list whose head is stored at the location given by
h(k) in the hash table, we need to know the address of the node's predecessor. We do
this using a pointer save. The running time complexity of the delete operation is same
as that of the search operation because we need to search the predecessor of the node
so that the node can be removed without affecting other nodes in the list. */
void delete_value(node *hash_table[], int val) {
    node *save, *ptr;
    save = NULL;
    ptr = hash_table[h(val)];
    while ((ptr != NULL) && (ptr->value != val)) {
        save = ptr;
        ptr = ptr->next;
    }
    if (ptr != NULL) {
        save->next = ptr->next;
        free(ptr);
    }
    else
        printf("\n VALUE NOT FOUND");
}
```

Разрешение коллизий путем объединения в цепочку - пример

пример

Вставьте ключи 7, 24, 18, 52, 36, 54, 11 и 23 в цепочку хэш-таблицыиз 9 ячеек памяти. Используйте h(k) = k mod m. В этом случае m=9. Изначально хэш-таблица может быть задана как:

0	NULL
1	NULL
2	NULL
3	NULL
4	NULL
5	NULL
6	NULL
7	NULL
8	NULL

Шаг 1
Ключ = 7
h(k) = 7 mod 9 = 7
Создайте связанный список для ячейки 7 и сохраните в нем ключевое значение 7 как его единственный узел.

0	NULL
1	NULL
2	NULL
3	NULL
4	NULL
5	NULL
6	NULL
7	→ 7 X
8	NULL

Шаг 2
Ключ = 24
h(k) = 24 mod 9 = 6
Создайте связанный список для ячейки 6 и сохраните в нем ключевое значение 24 как его единственный узел.

0	NULL
1	NULL
2	NULL
3	NULL
4	NULL
5	NULL
6	→ 24 X
7	→ 7 X
8	NULL

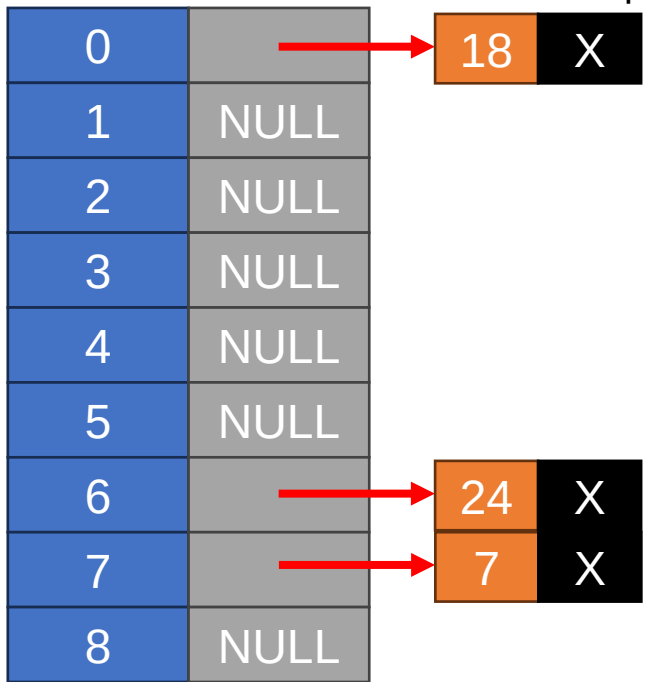
Разрешение коллизий путем объединения в цепочку - пример

пример 7, 24, 18, 52, 36, 54, 11 и 23

Шаг 3

Ключ = 18 h(k) = 18 mod 9 = 0

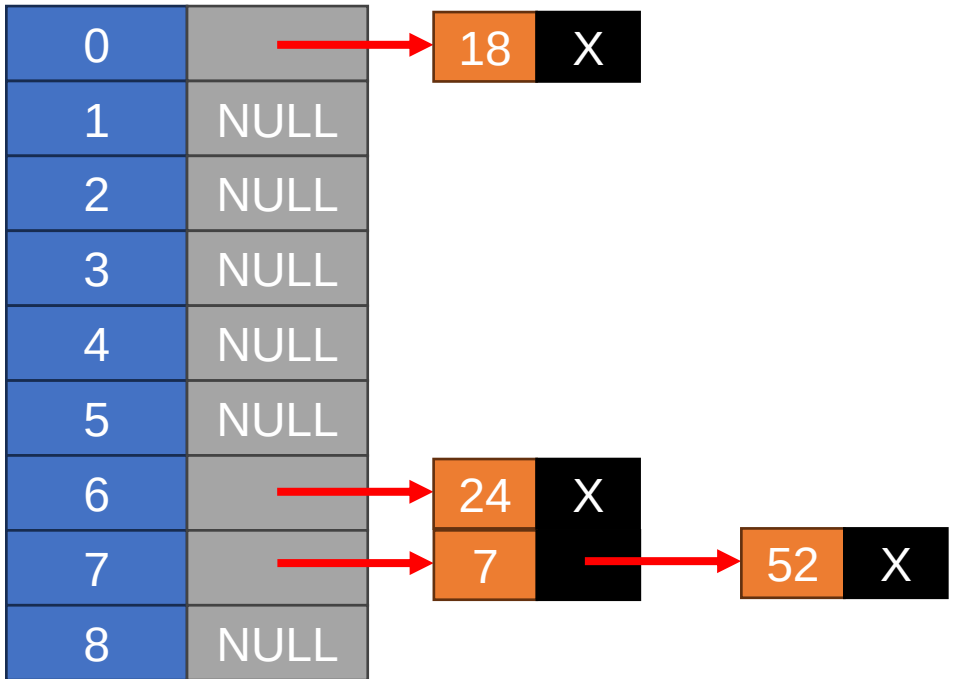
Создайте связанный список для местоположения 0 и сохраните в нем значение ключа 18 как его единственный узел.



Шаг 4

Ключ = 52 h(k) = 52 mod 9 = 7

Вставьте 52 в конец связанного списка местоположения 7.



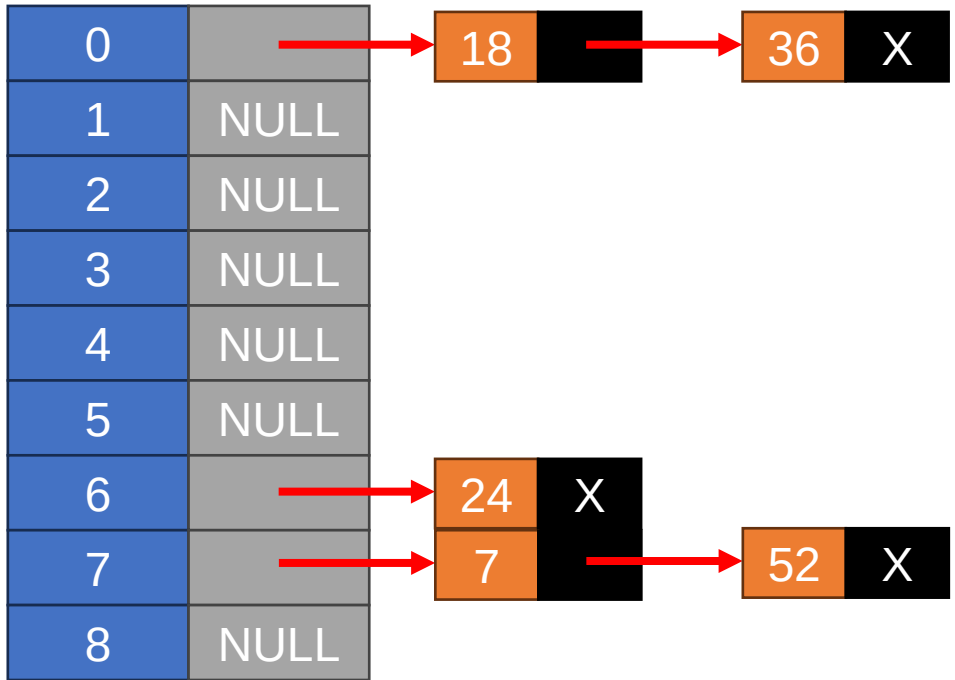
Разрешение коллизий путем объединения в цепочку - пример

пример 7, 24, 18, 52, 36, 54, 11 и 23

Шаг 5:

Ключ = 36 h(k) = 36 mod 9 = 0

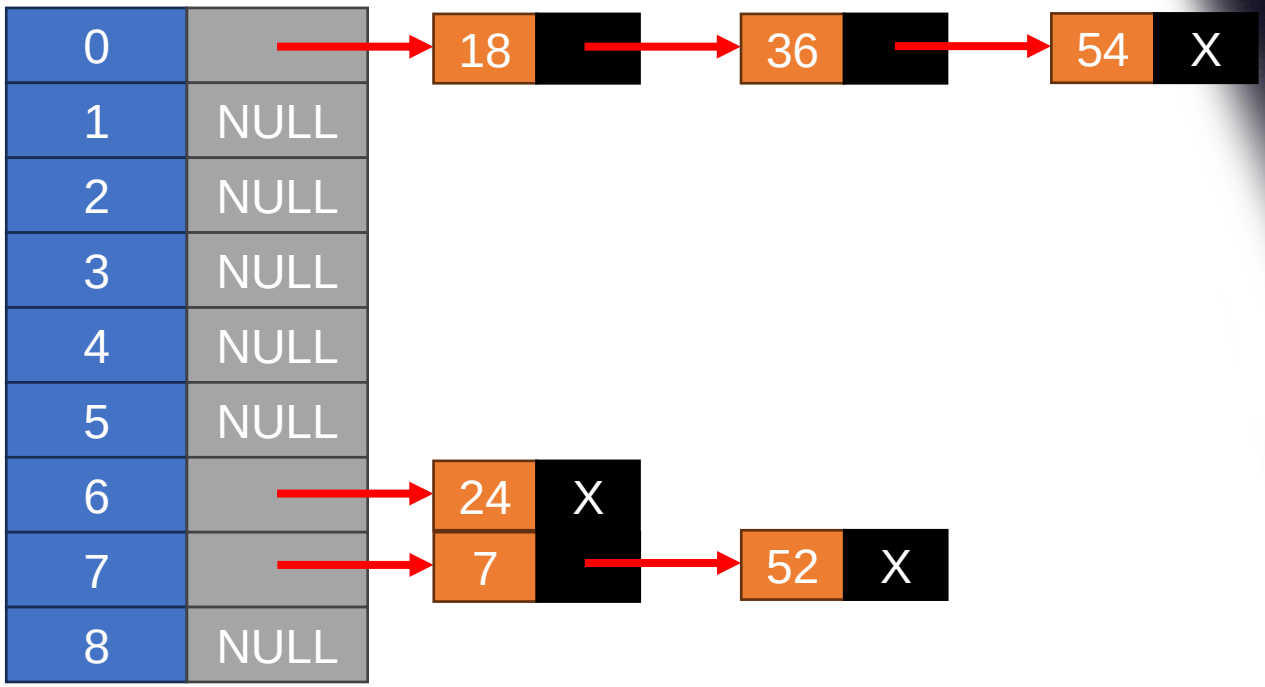
Вставьте 36 в конец связанного списка местоположения 0.



Шаг 6:

Ключ = 54 h(k) = 54 mod 9 = 0

Вставьте 54 в конец связанного списка местоположения



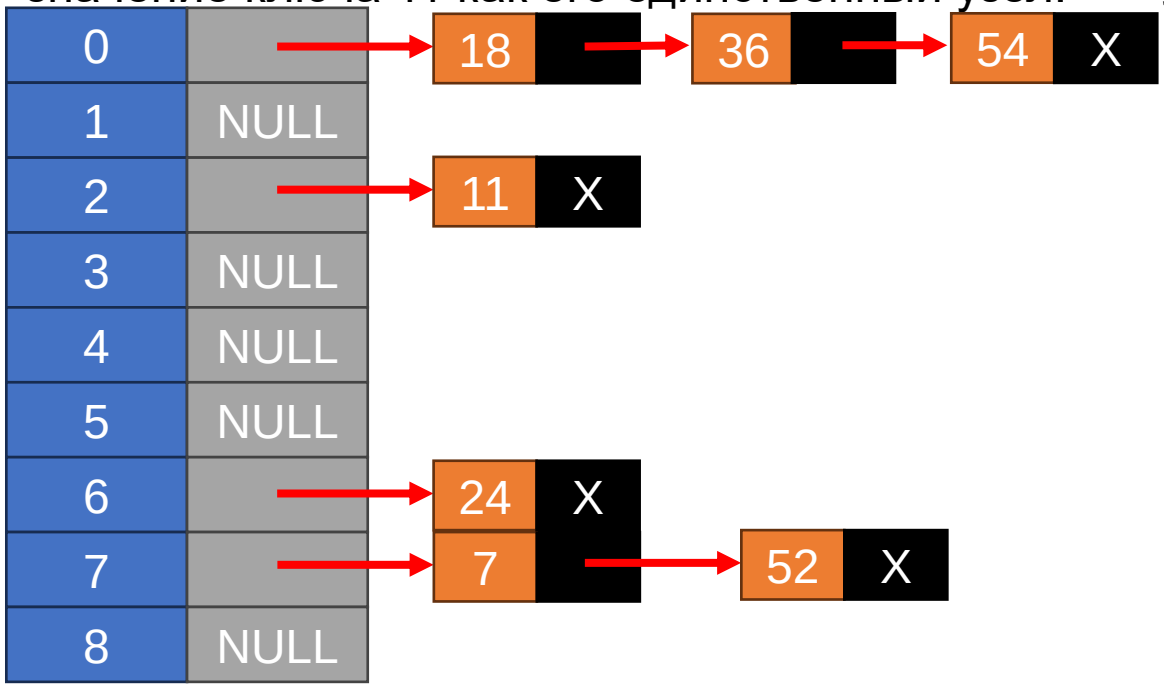
Разрешение коллизий путем объединения в цепочку - пример

пример 7, 24, 18, 52, 36, 54, 11 и 23

Шаг 7:

Ключ = 11 h(k) = 11 mod 9 = 2

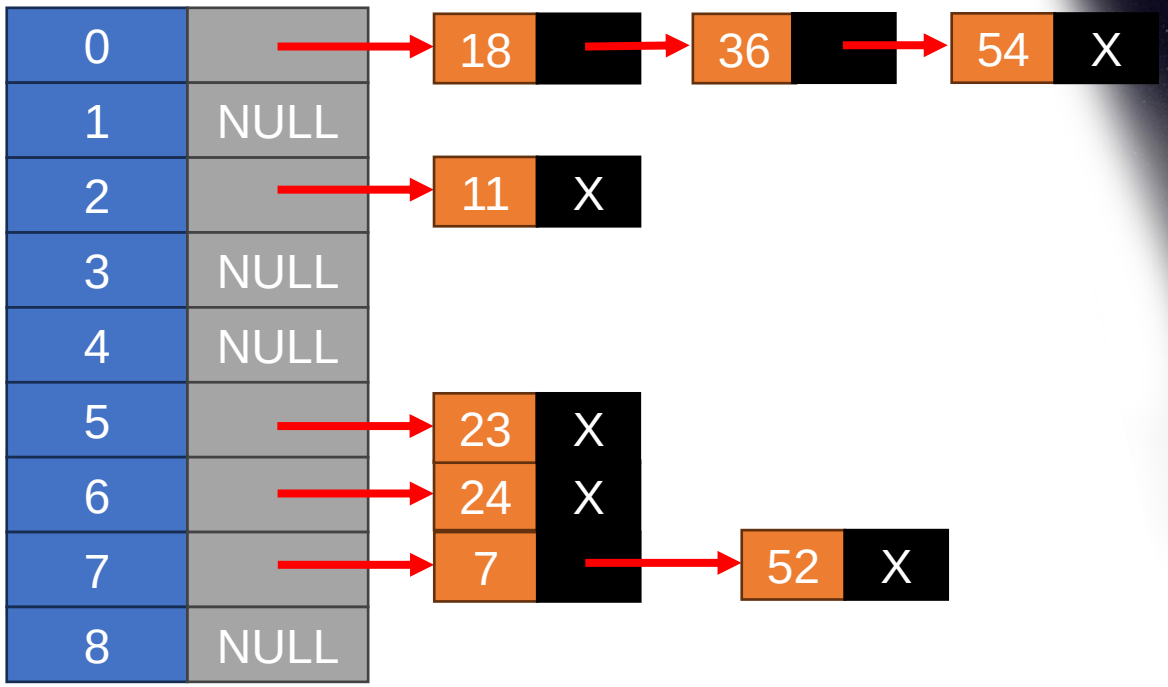
Создайте связанный список для местоположения 2 и сохраните в нем значение ключа 11 как его единственный узел.



Шаг 8:

Ключ = 23 h(k) = 23 mod 9 = 5

Создайте связанный список для местоположения 5 и сохраните в нем значение ключа 23 как его единственный узел.



Анализ хеширования с цепочками

Насколько высока производительность хеширования с цепочками? В частности, сколько времени требуется для поиска элемента с заданным ключом?

Пусть у нас есть хеш-таблица T с m ячейками, в которых хранятся n элементов. Определим коэффициент заполнения таблицы T как n/m, т.е. как среднее количество элементов, хранящихся в одной цепочке. Наш анализ будет опираться на значение величины α, которая может быть меньше, равна или больше единицы.

В наихудшем случае хеширование с цепочками ведет себя крайне неприятно:

все n ключей хешированы в одну и ту же ячейку, создав список длиной n. Таким образом, время поиска в наихудшем случае равно Θ(n) плюс время вычисления хеш-функции, что ничуть не лучше, чем в случае использования связанного списка для хранения всех n элементов. Понятно, что использование хеш-таблиц в наихудшем случае совершенно бессмысленно. (Идеальное хеширование, применимое в случае статического множества ключей обеспечивает высокую производительность даже в наихудшем случае.)

Производительность хеширования в среднем случае зависит от того, насколько хорошо хеш-функция h распределяет множество сохраняемых ключей по m ячейкам в среднем. Будем полагать, что все элементы хешируются по ячейкам равномерно и независимо, и назовем данное предположение ***простым равномерным хешированием*** (simple uniform hashing).

Анализ хеширования с цепочками

Обозначим длины списков T[j] для j = 0, 1, …,m - 1 как n_j, так что

$n = n_0 + n_1 + + n_{m-1}$

а ожидаемое значение n, равно $E[n_3] = \alpha = n/m$.

Мы полагаем, что для вычисления хеш-значения h(k) достаточно времени O(1), так что время, необходимое для поиска элемента с ключом k, линейно зависит от длины $n_{h(k)}$ списка T[h(k)]. Не учитывая время O(1), требующееся для вычисления хеш-функции и доступа к ячейке h(k), рассмотрим математическое ожидание количества элементов, которое должно быть проверено алгоритмом поиска (т.е. количество элементов в списке T[h(k)], которые проверяются на равенство их ключей величине k). Мы должны рассмотреть два случая: во-первых, когда поиск неудачен и в таблице нет элементов с ключом к и, во-вторых, когда поиск заканчивается успешно и в таблице определяется элемент с ключом k.

Анализ хеширования с цепочками

Теорема

В хеш-таблице с разрешением коллизий методом цепочек время неудачного поиска в среднем случае в предположении простого равномерного хеширования составляет Θ(1 + α).

Доказательство.

В предположении простого равномерного хеширования любой ключ k, который еще не находится в таблице, может быть помещен с равной вероятностью в любую из m ячеек. Математическое ожидание времени неудачного поиска ключа k равно времени поиска до конца списка T[h(k)], ожидаемая длина которого – $E[n_{h(k)}] = \alpha$. Таким образом, при неудачном поиске математическое ожидание количества проверяемых элементов равно α, а общее время, необходимое для поиска, включая время вычисления хеш-функции h(k), равно 0(1 + α).

Успешный поиск несколько отличается от неудачного, поскольку вероятность поиска в списке различна для разных списков и пропорциональна количеству содержащихся в нем элементов. Тем не менее и в этом случае математическое ожидание времени поиска остается равным Θ(1 + α).

Анализ хеширования с цепочками

Теорема

В хеш-таблице с разрешением коллизий методом цепочек время успешного поиска в среднем случае в предположении простого равномерного хеширования в среднем равно $\Theta(1 + \alpha)$.

Доказательство.

Мы полагаем, что искомый элемент с равной вероятностью может быть любым элементом, хранящимся в таблице. Количество элементов, проверяемых в процессе успешного поиска элемента x, на 1 больше, чем количество элементов, находящихся в списке перед x. Элементы, находящиеся в списке до x, были вставлены в список после того, как элемент x был сохранен в таблице, так как новые элементы помещаются в начало списка. Для того чтобы найти математическое ожидание количества проверяемых элементов, мы возьмем среднее по всем n элементам x в таблице значение, которое равно 1 плюс математическое ожидание количества элементов, добавленных в список x после самого искомого элемента. Пусть x_i обозначает i-й элемент, вставленный в таблицу ($i = 1, 2, \ldots, n$), и пусть $k_i = x_i.key$. Определим для ключей k_i и k_j индикаторную случайную величину $X_{ij} = I\{h(k_i) = h(k_j)\}$. В предположении простого равномерного хеширования $Pr\{h(k_i) = h(k_j)\} = 1/m$ и, соответственно, $E[X_i] = 1/m$.

Анализ хеширования с цепочками

Доказательство.

Таким образом, математическое ожидание количества проверяемых элементов в случае успешного поиска равно

$$E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right]=\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}E\left[X_{ij}\right]\right)=\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right)=1+\frac{1}{nm}\sum_{i=1}^{n}(n-i)$$

$$=1+\frac{1}{nm}\left(\sum_{i=1}^{n}n-\sum_{i=1}^{n}i\right)=1+\frac{1}{nm}\left(n^2-\frac{n(n+1)}{2}\right)=1+\frac{n-1}{2m}=1+\frac{\alpha}{2}+\frac{\alpha}{2n}$$

(из линейности математического ожидания)

Таким образом, полное время, необходимое для проведения успешного поиска (включая время вычисления хеш-функции), составляет Θ(2 + a/2 - a/2n) = Θ(1 + a).

О чем говорит проведенный анализ? Если количество ячеек в хеш-таблице как минимум пропорционально количеству элементов, хранящихся в ней, то n = O(m) и, следовательно, α = n/m = O(m)/m = O(1). Таким образом, поиск элемента в хеш-таблице в среднем требует постоянного времени. Поскольку в худшем случае вставка элемента в хеш-таблицу занимает O(1) времени (как и удаление элемента при использовании дважды связанных списков), можно сделать вывод, что все словарные операции в хеш-таблице в среднем выполняются за время O(1).

Плюсы и минусы

Главное преимущество использования цепочечных хеш-таблиц заключается в том, что они остаются эффективными, даже когда количество хранимых значений ключей намного превышает количество ячеек в хеш-таблице. Однако с увеличением количества хранимых ключей производительность цепочечных хеш-таблиц постепенно (линейно) ухудшается. Например, цепочечная хеш-таблица с 1000 ячеек памяти и 10 000 сохраненных ключей даст в 5–10 раз меньшую производительность по сравнению с цепочечными хеш-таблицами с 10 000 ячеек. Но цепочечная хеш-таблица все равно в 1000 раз быстрее простой хеш-таблицы. Другое преимущество использования цепочки для разрешения коллизий заключается в том, что ее производительность, в отличие от квадратичного зондирования, не ухудшается, когда таблица заполнена более чем наполовину. Этот метод абсолютно свободен от проблем кластеризации и, таким образом, обеспечивает эффективный механизм обработки коллизий.

Однако цепочечные хеш-таблицы наследуют недостатки связанных списков. Во-первых, для хранения значения ключа накладные расходы пространства следующего указателя в каждой записи могут быть значительными. Во-вторых, обход связанного списка имеет низкую производительность кэша, что делает кэш процессора неэффективным.

Хеширование бакетов

В закрытом хешировании все записи напрямую хранятся в хеш-таблице. Каждая запись со значением ключа k хранится в месте, называемом ее домашней позицией. Домашняя позиция вычисляется путем применения некоторой хеш-функции. В случае, если домашняя позиция записи с ключом k уже занята другой записью, то запись будет сохранена в каком-то другом месте в хеш-таблице. Это другое место будет определяться методом, который используется для разрешения коллизий. После вставки записей тот же алгоритм снова применяется для поиска определенной записи.

Одна из реализаций закрытого хеширования группирует хеш-таблицу в блоки, где M слотов хеш-таблицы делятся на B блоков. Таким образом, каждый блок содержит M/B слотов. Теперь, когда необходимо вставить новую запись, хэш-функция вычисляет исходную позицию. Если слот свободен, запись вставляется. В противном случае слоты бакеты последовательно просматриваются, пока не будет найден открытый слот. В случае, если весь бакет заполнен, запись вставляется в переполненный бакет. Переполненный бакет имеет бесконечную емкость в конце таблицы и является общим для всех. Эффективной реализацией хэширования ведра будет использование хэш-функции, которая равномерно распределяет записи между бакетами, так что в переполненный бакет нужно вставлять очень мало записей.

Другие хеш-функции

Одна из простых хеш-функций. Разработана Гленом Фаулером, Лондоном Керт Нолом и Фогном Во для общего применения.

```
unsigned int FNV1AHash(char *input)
{
    const unsigned int FNV_prime = 0x1000193;
    const unsigned int FNV_offset_basic = 0x811C9DC5;
    unsigned int hash = FNV_offset_basic;
    for (int i = 0; i < sizeof(input); ++i)
    {
        char byte_of_data = i;
        hash ^= byte_of_data;
        hash *= FNV_prime;
    }
    return hash;
}
```

Другие хеш-функции

Одна из простых хеш-функций. Разработана Бобом Дженкинсом для общего применения.

```c
unsigned JOAATHash(char *input)
{
    unsigned int hash = 0;
    for (int i = 0; i < sizeof(input); ++i)
    {
        unsigned char byte_of_data = (unsigned char)i;
        hash += byte_of_data;
        hash += hash << 10;
        hash ^= hash >> 6;
    }
    hash += hash << 3;
    hash ^= hash >> 11;
    hash += hash << 15;
    return hash;
}
```

Другие хеш-функции

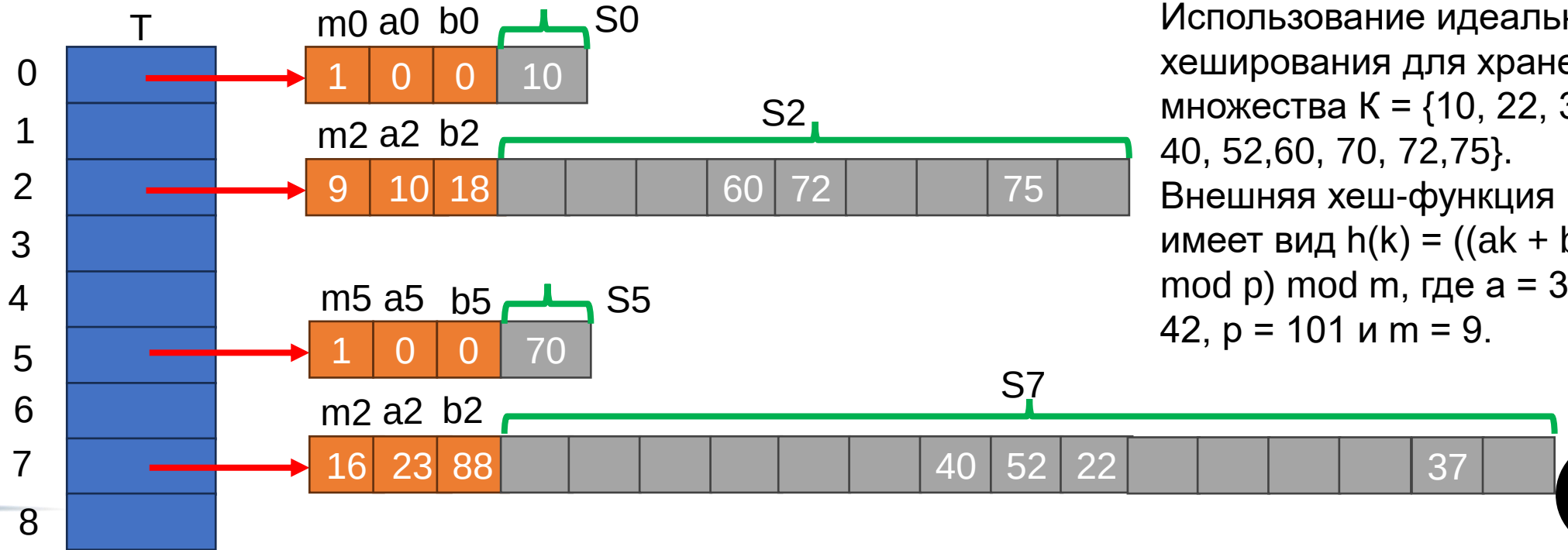
Одна из простых хеш-функций. Разработана Питером Вэйнбергером для общего применения.

```
unsigned int PJW32Hash(char *input)
{
    unsigned int hash = 0;
    for (int i = 0; i < sizeof(input); ++i)
    {
        unsigned char byte_of_data = (unsigned char)i;
        hash = (hash << 4) + byte_of_data;
        unsigned int h1 = hash & 0xf0000000;
        if (h1 != 0)
        {
            hash = ((hash ^ (h1 >> 24)) & (0xffffff));
        }
    }
    return hash;
}
```

Идеальное хеширование

Хотя чаще всего хеширование используется из-за превосходной средней производительности, возможна ситуация, когда можно обеспечить превосходную производительность хеширования в наихудшем случае. Такой ситуацией является статическое множество ключей, т.е. после того как все ключи сохранены в таблице, их множество никогда не изменяется. Ряд приложений в силу своей природы работает со статическими множествами ключей. Идеальным хешированием мы называем методику, которая выполняет поиск за O(1) обращений к памяти в наихудшем случае.

Для создания схемы идеального хеширования мы используем двухуровневую схему хеширования с универсальным хешированием на каждом уровне.



Использование идеального хеширования для хранения множества K = {10, 22, 37, 40, 52,60, 70, 72,75}. Внешняя хеш-функция имеет вид h(k) = ((ak + b) mod p) mod m, где a = 3, b = 42, p = 101 и m = 9.

Идеальное хеширование

Чтобы гарантировать отсутствие коллизий на втором уровне, требуется, чтобы размер m, хеш-таблицы S, был равен квадрату числа n, ключей, хешированных в ячейку j. Такая квадратичная зависимость m, от n, может показаться чрезмерно расточительной, однако далее мы покажем, что при должном выборе хеш-функции первого уровня ожидаемое количество требуемой для хеш-таблицы памяти можно ограничить значением O(n).

Мы выбираем хеш-функцию из универсальных классов хеш-функций. Хеш-функция первого уровня выбирается из класса $\mathcal{H}_{pm}$, где, p является простым числом, превышающим значение любого из ключей. Ключи, хешированные в ячейку j, затем повторно хешируются во вторичную хеш-таблицу S_j, размером m_j, с использованием хеш-функции h_j, выбранной из класса $\mathcal{H}_{p,m_j}{}^2$

Работа будет выполнена в два этапа. Сначала мы выясним, как гарантировать отсутствие коллизий во вторичной таблице. Затем мы покажем, что общее ожидаемое количество памяти, необходимой для первичной и всех вторичных хеш таблиц, равно O(n).

Идеальное хеширование

Теорема

Предположим, что n ключей сохраняются в хеш-таблице размером m = n^2 с использованием хеш-функции h, случайно выбранной из универсального класса хеш-функций. Тогда вероятность возникновения коллизий оказывается меньше 1/2.

Доказательство. Всего имеется $\binom{n}{2}$ пар ключей, которые могут вызвать коллизию. Если хеш-функция выбрана случайным образом из универсального семейства хеш-функций $\mathcal{H}$, то для каждой пары вероятность возникновения коллизии равна 1/m. Пусть X - случайная величина, которая подсчитывает количество коллизий. Если m = n^2, то математическое ожидание числа коллизий равно

$$E[X] = \binom{n}{2} \cdot \frac{1}{n^2} = \frac{n^2 - n}{2} \cdot \frac{1}{n^2} \leq \frac{1}{2}$$

(Применение неравенства Маркова, Pr{X ≥ t} ≤ E[X]/t, при t = 1 завершает доказательство.

Идеальное хеширование

В ситуации, описанной в теореме 11.7, когда $m = n^2$, произвольно выбранная и множества H хеш-функция с большей вероятностью не приведет к коллизиям, чем приведет к ним. Для заданного множества K, содержащего n ключей (напомним, что K - статическое множество), найти хеш-функцию h, не дающую коллизий, можно после нескольких случайных попыток.

Однако если значение n велико, таблица размером $m = n^2$ оказывается слишком большой и приводит к ненужному перерасходу памяти. Поэтому мы принимаем двухуровневую схему хеширования и используем подход из теоремы только для хеширования записей в пределах каждой ячейки. Внешняя хеш-функция h первого уровня используется для хеширования ключей в m = n ячеек.

Затем, если в ячейку j хешировано ключей, для того чтобы обеспечить отсутствие коллизий и поиск за константное время, используется вторичная хеш-таблица S_j размером $m_j = n_j^2$.

Вернемся к вопросу необходимого для описанной схемы количества памяти. Поскольку размер j-й вторичной хеш-таблицы т, растет с ростом n, квадратично, возникает риск, что в целом потребуется очень большое количество памяти.

Если хеш-таблица первого уровня имеет размер m = n, то, естественно, нам потребуется количество памяти, равное O(n), для первичной хеш-таблицы, а также для хранения размеров m, вторичных хеш-таблиц и параметров a_j и b_j, определяющих вторичные хеш-функции h_j, выбираемые из класса $\mathcal{H}_{p,m_j}$ (за исключением случая, когда $n_j = 1$; в этом случае мы просто принимаем a = b = 0).

Идеальное хеширование

Теорема

Предположим, что мы сохраняем n ключей в хеш-таблице размером m = n с использованием хеш-функции h, выбираемой случайным образом из универсального класса хеш-функций. Тогда мы имеем

$$E\left[\sum_{j=0}^{m-1} n_j^2\right] < 2n,$$

где n_j - количество ключей, хешированных в ячейку j.

Доказательство. Начнем со следующего тождества, которое справедливо для любого неотрицательного целого a:

$$a^2 = a + 2\binom{a}{2}.$$

Мы имеем

$$E\left[\sum_{j=0}^{m-1} n_j^2\right] = E\left[\sum_{j=0}^{m-1} n_j + 2\binom{n_j}{2}\right] = E\left[\sum_{j=0}^{m-1} n_j\right] + 2E\left[\sum_{j=0}^{m-1} \binom{n_j}{2}\right] = E[n] + 2E\left[\sum_{j=0}^{m-1} \binom{n_j}{2}\right]$$

$$= n + 2E\left[\sum_{j=0}^{m-1} \binom{n_j}{2}\right] \leq n + 2\frac{n-1}{2} < 2n$$

Идеальное хеширование

Следствие Если мы сохраняем n ключей в хеш-таблице размером m = n с использованием хеш-функции h, выбираемой случайным образом из универсального класса хеш-функций, и устанавливаем размер каждой вторичной хеш-таблицы равным $m_j = n_j^2$, j = 0, 1, …, m - 1, то математическое ожидание количества необходимой для вторичных хеш-таблиц в схеме идеального хеширования памяти не превышает величины 2n.

Доказательство. Поскольку $m_j = n_j^2$для j = 0, 1, …,m - 1, теорема 1.8 дает

$$E\left[\sum_{j=0}^{m-1} m_j\right] = E\left[\sum_{j=0}^{m-1} n_j^2\right] < 2n$$

что и требовалось доказать.

Идеальное хеширование

Следствие

Если мы сохраняем n ключей в хеш-таблице размером m = n с использованием хеш-функции h, выбираемой случайным образом из универсального класса хеш-функций, и устанавливаем размер каждой вторичной хеш-таблицы равным $m_j = n_j^2$, j = 0, 1, ,m - 1, то вероятность того, что общее количество необходимой для вторичных хеш-таблиц памяти не менее 4n, меньше, чем 1/2.

Доказательство. Вновь применим неравенство Маркова , Pr{X ≥ t} ≤ E[X] /t, на этот раз - к неравенству, с $X = \sum_{j=0}^{m-1} m_j$ и t = 4n:

$$Pr\left\{\sum_{j=0}^{m-1} m_j \geq 4n\right\} \leq \frac{E\left[\sum_{j=0}^{m-1} m_j\right]}{4n} < \frac{2n}{4n} = \frac{1}{2}$$

Из следствия видно, что если проверить несколько случайным образом выбранных из универсального семейства хеш-функций, то достаточно быстро будет найдена хеш-функция, обеспечивающая умеренные требования к количеству памяти

Плюсы и минусы хеширования

- ✓ Для хранения индекса не требуется дополнительного места, как в случае с другими структурами данных.
- ✓ Хеш-таблица обеспечивает быстрый доступ к данным и дополнительное преимущество быстрых обновлений.
- ✗ Для вставки и извлечения значений ей обычно не хватает локальности и последовательного извлечения по ключу. Это делает вставку и извлечение значений данных еще более случайными. Тем более, выбор эффективной хеш-функции — это скорее искусство, чем наука. Нередко (в хеш-таблицах с открытой адресацией) создается плохая хеш-функция.

Применение хеширования

Хеш-таблицы широко используются в ситуациях, когда необходимо получить доступ к огромным объемам данных для быстрого поиска и извлечения информации. Здесь приведены несколько типичных примеров использования хеширования. Хеширование используется для индексирования базы данных. Некоторые системы управления базами данных хранят отдельный файл, известный как индексный файл. Когда данные необходимо извлечь из файла, ключевая информация сначала ищется в соответствующем индексном файле, который ссылается на точное местоположение записи данных в файле базы данных. Эта ключевая информация в индексном файле часто хранится как хешированное значение.

Во многих системах баз данных хеширование файлов и каталогов используется в высокопроизводительных файловых системах. Такие системы используют два дополнительных метода для повышения производительности доступа к файлам. В то время как одним из этих методов является кэширование, которое сохраняет информацию в памяти, другим является хеширование, которое делает поиск местоположения файла в памяти намного быстрее, чем большинство других методов. Метод хеширования используется для реализации таблиц символов компилятора в C++. Компилятор использует таблицу символов для хранения записи всех пользовательских символов в программе C++. Хеширование позволяет компилятору быстро находить имена переменных и другие атрибуты, связанные с символами. Хеширование также широко используется в поисковых системах Интернета.

Реальные приложения хеширования

Базы данных компакт-дисков

Для компакт-дисков желательно иметь всемирную базу данных компакт-дисков, чтобы, когда пользователи вставляют свой диск в проигрыватель компакт-дисков, они получали полную таблицу содержания на экране своего компьютера. Эти таблицы не хранятся на самих дисках, т. е. компакт-диск не хранит никакой информации о песнях, а загружается из базы данных. Критическая проблема, которую необходимо решить здесь, заключается в том, что на компакт-дисках не хранятся идентификационные номера, так как же компьютер узнает, какой компакт-диск был вставлен в проигрыватель? Единственная информация, которую можно использовать, — это длина трека и тот факт, что каждый компакт-диск отличается. По сути, из длин треков создается большое число, также известное как «сигнатура». Эта сигнатура используется для идентификации конкретного компакт-диска. Сигнатура — это значение, полученное путем хеширования. Например, создается число длиной 8 или 10 шестнадцатеричных цифр; затем число отправляется в базу данных, и эта база данных ищет ближайшее совпадение. Причина в том, что длина дорожки может быть измерена неточно.

Водительские права/страховые карты

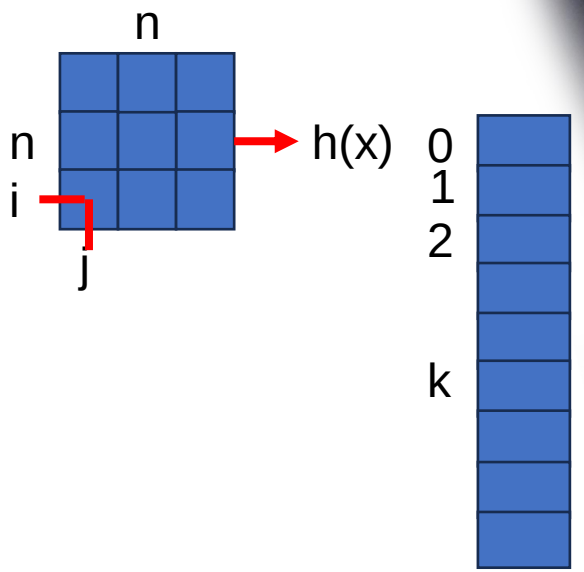
Как и в нашем примере с компакт-диском, даже номера водительских прав или страховых карт создаются с помощью хеширования из элементов данных, которые никогда не меняются: дата рождения, имя и т. д.

Реальные приложения хеширования

Разреженная матрица

Разреженная матрица — это двумерный массив, в котором большинство записей содержат 0. То есть в разреженном массиве очень мало ненулевых записей. Конечно, мы можем хранить двумерный массив как есть, но это приведет к пустой трате ценной памяти. Поэтому еще одна возможность — хранить ненулевые элементы запасной матрицы как элементы одномерного массива. То есть, используя хеширование, мы можем хранить двумерный массив в одномерном массиве. Между элементами разреженной матрицы и элементами массива существует однозначное соответствие.

Если размер разреженной матрицы равен n x n, и в ней есть N ненулевых записей, то из координат (i,j) матрицы мы определяем индекс k в массиве простым вычислением. Таким образом, мы имеем k=h(i,j) для некоторой функции h, называемой хэш-функцией. Размер одномерного массива пропорционален N. Это намного лучше размера разреженной матрицы, которая требовала хранения, пропорционального n ¥ n. Например, если у нас есть треугольная разреженная матрица A, то запись A[i, j] можно сопоставить с записью в одномерном массиве, вычислив индекс с помощью хэш-функции h(i, j) = i(i–1)/2 + j.



Реальные приложения хеширования

Подписи файлов

Подписи файлов предоставляют компактное средство идентификации файлов. Мы используем функцию h[x], подпись файла, которая является свойством файла. Хотя мы можем хранить файлы по имени, подписи предоставляют компактную идентификацию файлов. Поскольку подпись зависит от содержимого файла, если в файл вносятся какие-либо изменения, то подпись изменится. Таким образом, подпись файла можно использовать в качестве быстрой проверки, чтобы узнать, изменил ли кто-либо файл или он потерял часть информации во время передачи. Подписи широко используются для файлов, в которых хранятся оценки студентов.

Игровые доски

На игровой доске для крестиков-ноликов или шахмат, положение в игре может храниться с помощью хэш-функции.

Графика

В графике центральной проблемой является хранение объектов в сцене или представлении. Для этого мы организуем наши данные с помощью хэширования. Хэширование может использоваться для создания сетки соответствующего размера, обычной вертикально-горизонтальной сетки. (Обратите внимание, что сетка — это не что иное, как двумерный массив, и существует взаимно-однозначное соответствие при переходе от двумерного массива к одномерному массиву.) Таким образом, мы храним сетку как одномерный массив, как мы делали в случае разреженных матриц. Ключевым преимуществом этого метода хранения является быстрое выполнение операций, таких как поиск ближайшего соседа.

Agenda

Хеш-функции и хеш-таблицы

75 минут

Указатель на функцию

15 минут

Причудливые объявления

Язык C позволяет создавать сложные формы данных. Хотя мы придерживаемся простейших форм, все же имеет смысл отметить некоторые из доступных возможностей. Когда вы делаете объявление, имя (или идентификатор) можно изменить, добавив модификатор.

Модификатор	Описание
*	Обозначает указатель
()	Обозначает функцию
[]	Обозначает массив

```
int board[8][8]; // массив из массивов значений int
int ** ptr; // указатель на указатель на int
int * risks[10]; // 10-элементный массив указателей на int
int (* rusks ) [10]; // указатель на массив из 10 значений int
int * oof[3][4]; // массив размером 3 x 4 указателей на int
int (*uuf)[3][4]; // указатель на массив размером 3 x 4 значений int
int (*uof [3])[4]; // 3-элементный массив указателей на 4-элементные массивы значений int
char * fump(int); // функция, возвращающая указатель на char
char (* frump)(int); // указатель на функцию, возвращающую тип char
char (* flump[3]) (int); // массив из 3 указателей на функции, которые возвращают тип char
```

Функции и указатели

Как продемонстрировало обсуждение объявлений, допускается объявлять указатели на функции. Возможно, вас интересует, в чем они могут быть полезны.

Обычно указатель на функцию используется в качестве аргумента в другой функции, сообщая ей, какую функцию применять. Например, сортировка массива предполагает сравнение двух элементов для выяснения того, какой из них должен следовать первым. В случае числовых элементов можно использовать операцию >. Но в целом элементом может быть строка или структура, что требует вызова специальной функции для выполнения сравнения. Функция qsort () из библиотеки С спроектирована на работу с массивами любого вида при условии, что в ы уведомите ее о том, какую функцию применять для сравнения элементов. С этой целью qsort () принимает в одном из своих аргументов указатель на функцию. Затем qsort () использует указанную функцию для сортировки значений определенного типа — будь он целочисленным, строкой или структурой.

Функции и указатели

void ToUpper(char *); // преобразует строку в верхний регистр
Тип ToUpper() определен как “функция с параметром char * и возвращаемым типом void". Вот как объявить указатель на функцию такого типа по имени pf:
void (*pf)(char *); // pf - указатель на функцию
Читая это объявление, вы видите, что первая пара круглых скобок связывает операцию * с pf, т.е. pf является указателем на функцию. Это делает (*pf) функцией, а (char *) — списком ее параметров функции и void — возвращаемым типом.
void *pf(char *); // pf - функция, которая возвращает указатель

В этом контексте для представления адреса функции может применяться ее имя:
void ToUpper(char *)
void ToLower(char *)
int round(double);
void (*pf) (char *);
pf = ToUpper; // допустимо, ToUpper - адрес функции
pf = ToLower; // допустимо, ToLower - адрес функции
pf = round; // недопустимо, round - неподходящий тип функции
pf = ToLower(); // недопустимо, ToLower() не является адресом

Функции и указатели

Подобно тому, как можно применять указатель на данные с целью доступа к ним, вы можете использовать указатель на функцию для обращения к этой функции. На удивление для этого существуют два логически несогласованных синтаксических правила, как иллюстрируется в следующем фрагменте:

```
void ToUpper(char *);
void ToLower(char *);
void (*pf) (char *);
char mis [] = "Nina Metier";
pf = ToUpper;
(*pf)(mis); // применить ToUpper к mis (первый синтаксис)
pf = ToLower;
pf(mis); // применить ToLower к mis (второй синтаксис)
```

Компилятор K & R C не разрешает вторую форму, но для поддержки совместимости с существующим кодом стандарт ANSI C принимает обе формы ((*pf) (mis) и pf(mis)) как эквивалентные. Последующие стандарты сохранили такой в высшей степени двойственный подход.

Функции и указатели

Одним из наиболее распространенных случаев использования указателей на данные является аргумент функции, и то же самое относится к указателю на функцию.
Например, рассмотрим следующий прототип функции:
void show(void (* fp) (char *), char * str);
Он выглядит запутанным, но в нем объявляются два параметра, fp и str. Параметр fp — это указатель на функцию, а str — указатель на данные. Точнее, fp указывает на функцию, которая принимает параметр char * и имеет возвращаемый тип void, а str указывает на char. Таким образом, имея представленные выше объявления, можно делать вызовы функций вроде приведенных ниже:

```
 show(ToLower, mis); /* show () использует функцию ToLower(): fp = ToLower * /
show(pf, mis) ; /*show() использует функцию, указанную посредством pf: fp = pf */
```

Каким образом show() применяет переданный указатель на функцию?
Для вызова этой функции в show() используется либо синтаксис fp(), либо синтаксис (*fp) ():

```
void show(void (* fp)(char *), char * str)
{
    (*fp)(str); /* применить выбранную функцию к str */
    puts(str); /* отобразить результат */
}
```

Здесь функция show () сначала трансформирует строку str, применяя к ней функцию, на которую указывает fp, после чего отображает результирующую строку.