Лекция: Хеширование

*Хеширование* - одна из классических задач компьютерных наук: его различные алгоритмы подробно исследованы и находят широкое применение.

Х*еширование* (*hashing*) это расширенный вариант распределяющего поиска, применяемый в более типичных приложениях поиска, где ключи не обладают столь удобными свойствами. Конечный результат применения данного подхода совершенно не похож на методы, основанные на сравнении - вместо перемещения по структурам данных словаря с помощью сравнения ключей поиска с ключами в элементах, мы пытаемся обратиться к элементам в таблице непосредственно, выполняя арифметическое преобразование ключей в адреса таблицы.

Алгоритмы поиска, использующие *хеширование*, состоят из двух отдельных частей. Первый шаг - *вычисление* хеш-функции (*hash function*), которая преобразует *ключ* поиска в *адрес* в таблице. В идеале различные ключи должны были бы отображаться на различные адреса, но часто два или более различных ключа могут дать один и тот же *адрес* в таблице. Поэтому вторая часть поиска методом хеширования - процесс разрешения коллизий (*collision* *resolution*), который обрабатывает такие ключи. В одном из методов разрешения конфликтов, используются связные списки, поэтому он находит непосредственное применение в динамических ситуациях, когда трудно заранее предугадать количество ключей поиска. В других двух методах разрешения коллизий достигается высокая *производительность* поиска, поскольку элементы хранятся в фиксированном массиве. Мы рассмотрим способ усовершенствования этих методов, позволяющий использовать их и в тех случаях, когда нельзя заранее предсказать размеры таблицы.

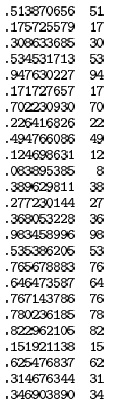
*Хеширование* - хороший пример баланса между временем и объемом памяти. Если бы не было ограничения на объем используемой памяти, любой *поиск* можно было бы выполнить с помощью всего лишь одного обращения к памяти, просто используя *ключ* в качестве адреса памяти, как при распределяющем поиске. Однако обычно этот идеальный случай недостижим, поскольку для длинных ключей может потребоваться огромный объем памяти. С другой стороны, если бы не было ограничений на *время выполнения*, можно было бы обойтись минимальным объемом памяти, пользуясь методом последовательного поиска. *Хеширование* представляет собой способ использования приемлемого объема как памяти, так и времени, и достижения баланса между этими двумя крайними требованиями. В частности, можно поддерживать любой баланс, просто меняя размер таблицы, а не переписывая код и не выбирая другие алгоритмы.

#### Хеш-функции

Прежде всего необходимо решить задачу вычисления хеш-функции, преобразующей ключи в адреса таблицы. Обычно реализация этого арифметического вычисления не представляет сложности, но все же необходимо соблюдать осторожность, чтобы не нарваться на различные малозаметные подводные камни. При наличии таблицы, которая может содержать M элементов, нужна функция, преобразующая ключи в целые числа в диапазоне [0, M - 1]. Идеальная хеш-функция должна легко вычисляться и быть похожей на случайную функцию: для любых аргументов результаты в некотором смысле должны быть равновероятными.

Хеш-функция зависит от типа ключа. Строго говоря, для каждого возможного вида ключей требуется отдельная хеш-функция. Для повышения эффективности обычно желательно избегать явного преобразования типов, обратившись вместо этого к идее рассмотрения двоичного представления ключей в машинном слове в виде целого числа, которое можно использовать в арифметических вычислениях. Хеширование появилось до языков высокого уровня - на ранних компьютерах было обычным делом рассматривать какое-либо значение то как строковый ключ, то как целое число. В некоторых языках высокого уровня затруднительно создавать программы, которые зависят от представления ключей в конкретном компьютере, поскольку такие программы, по сути, являются машинно-зависимыми, и поэтому их трудно перенести на другой компьютер. Обычно хеш-функции зависят от процесса преобразования ключей в целые числа, поэтому в реализациях хеширования бывает трудно одновременно обеспечить и машинную независимость, и эффективность. Как правило, простые целочисленные ключи или ключи типа с плавающей точкой можно преобразовать с помощью всего одной машинной операции, но строковые ключи и другие типы составных ключей требуют больших затрат и большего внимания к эффективности.

Простейшей является ситуация, когда ключами являются числа с плавающей точкой из фиксированного диапазона. Например, если ключи - числа, большие 0 и меньшие 1, их можно просто умножить на M, округлить результат до меньшего целого числа и получить адрес в диапазоне между 0 и M - 1; такой пример показан на рис. 1. Если ключи больше s и меньше t, их можно масштабировать, вычтя s и разделив на t-s, в результате чего они попадут в диапазон значений между 0 и 1, а затем умножить на M и получить адрес в таблице.

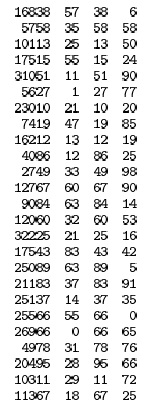


**Рис. 1.** Мультипликативная хеш-функция для ключей с плавающей точкой

Для преобразования чисел с плавающей точкой в диапазоне между 0 и 1 в индексы таблицы, размер которой равен 97, выполняется умножение этих чисел на 97. В данном примере произошло три коллизии: для индексов, равных 17, 53 и 76. Хеш-значения определяются старшими разрядами ключа, младшие разряды не играют никакой роли. Одна из целей разработки хеш-функции - устранение такого дисбаланса, чтобы во время вычисления учитывался каждый разряд.

Если ключи являются w-разрядными целыми числами, их можно преобразовать в числа с плавающей точкой и разделить на 2w для получения чисел с плавающей точкой в диапазоне между 0 и 1, а затем умножить на M, как в предыдущем абзаце. Если операции с плавающей точкой занимают много времени, а числа не столь велики, чтобы привести к переполнению, этот же результат может быть получен с помощью целочисленных арифметических операций: нужно ключ умножить на M, а затем выполнить сдвиг вправо на w разрядов для деления на 2w (или, если умножение приводит к переполнению, выполнить сдвиг, а затем умножение). Такие методы бесполезны для хеширования, если только ключи не распределены по диапазону равномерно, поскольку хеш-значение определяется только ведущими цифрами ключа.

Более простой и эффективный метод для w-разрядных целых чисел - один из, пожалуй, наиболее часто используемых методов хеширования - выбор в качестве размера M таблицы простого числа и вычисление остатка от деления к на M, т.е. h(k) = k mod M для любого целочисленного ключа k. Такая функция называется модульной хеш-функцией. Ее очень просто вычислить (k % M в языке C++), и она эффективна для достижения равномерного распределения значений ключей между значениями, меньшими M. Небольшой пример показан на рис. 2.



**Рис. 2.** Модульная хеш-функция для целочисленных ключей

В трех правых столбцах показан результат хеширования 16-разрядных ключей, приведенных слева, с помощью следующих функций:

v % 97 (слева)

v % 100 (в центре) и

(int) (a \* v) % 100 (справа),

где a = .618033. Размеры таблицы для этих функций соответственно равны 97, 100 и 100. Значения выглядят случайными (поскольку случайны ключи). Вторая функция (v % 100) использует лишь две крайние правые цифры ключей и поэтому для неслучайных ключей может показывать низкую производительность.

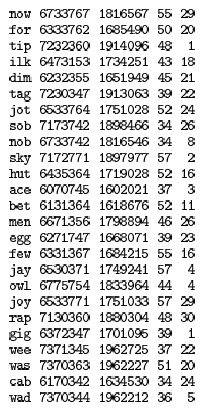
Модульное хеширование применимо и к ключам с плавающей точкой. Если ключи принадлежат небольшому диапазону, можно масштабировать их в числа из диапазона между 0 и 1, 2w для получения w-разрядных целочисленных значений, а затем использовать модульную хеш-функцию. Другой вариант - просто использовать в качестве операнда модульной хеш-функции двоичное представление ключа (если оно доступно).

Модульное хеширование применяется во всех случаях, когда имеется доступ к битам, из которых состоят ключи, независимо от того, являются ли они целыми числами, представленными машинным словом, последовательностью символов, упакованных в машинное слово, или представлены любым другим возможным вариантом. Последовательность случайных символов, упакованная в машинное слово - не совсем то же, что случайные целочисленные ключи, поскольку не все разряды используются для кодирования. Но оба эти типа (и любой другой тип ключа, закодированный так, чтобы уместиться в машинном слове) можно заставить выглядеть случайными индексами в небольшой таблице.

Основная причина выбора в качестве размера M хеш-таблицы простого числа для модульного хеширования показана на рис. 13. В этом примере символьных данных с 7-разрядным кодированием ключ трактуется как число с основанием 128 - по одной цифре для каждого символа в ключе. Слово now соответствует числу 1816567, которое может быть также записано как

$$110 \cdot 128^{2} + 111 \cdot 128^{1} + 119 \cdot 128^{0}$$

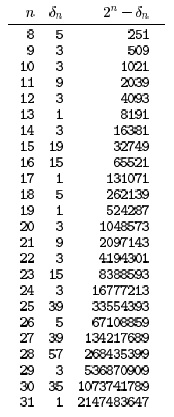
поскольку в ASCII-коде символам n, o и w соответствуют числа 1568 = 110, 1578 = 111 и 1678 = 119. Выбор размера таблицы M = 64 для этого типа ключа неудачен, поскольку добавление к х значений, кратных 64 (или 128), не меняет значение х mod 64 - для любого ключа значением хеш-функции является значение последних 6 разрядов этого ключа. Безусловно, хорошая хеш-функция должна учитывать все разряды ключа, особенно для символьных ключей. Аналогичные ситуации могут возникать, когда M содержит множитель, являющийся степенью 2. Простейший способ избежать этого - выбрать в качестве M простое число.



**Рис. 3.** Модульные хеш-функции для кодированных символов

В каждой строке этой таблицы приведены: 3-буквенное слово, представление этого слова в ASCII-коде как 21-битовое число в восьмеричной и десятичной формах и стандартные модульные хеш-функции для размеров таблиц 64 и 31 (два крайних справа столбца). Размер таблицы 64 приводит к нежелательным результатам, поскольку для получения хеш-значения используются только самые правые разряды ключа, а буквы в словах обычного языка распределены неравномерно. Например, всем словам, оканчивающимся на букву у, соответствует хеш-значение 57. И, напротив, простое значение 31 вызывает меньше коллизий в таблице более чем вдвое меньшего размера.

Модульное хеширование очень просто реализовать, за исключением того, что размер таблицы должен быть простым числом. Для некоторых приложений можно довольствоваться небольшим известным простым числом или же поискать в списке известных простых чисел такое, которое близко к требуемому размеру таблицы. Например, числа равные 2t - 1, являются простыми при t = 2, 3, 5, 7, 13, 17, 19 и 31 (и ни при каких других значениях t < 31): это известные простые числа Мерсенна. Чтобы динамически распределить таблицу нужного размера, нужно вычислить простое число, близкое к этому значению. Такое вычисление нетривиально (хотя для этого и существует остроумный алгоритм, который будет рассмотрен в части 5), поэтому на практике обычно используют таблицу заранее вычисленных значений (см. рис. 4). Использование модульного хеширования - не единственная причина, по которой размер таблицы стоит сделать простым числом; еще одна причина рассматривается в разделе 14.4.



**Рис. 4.** Простые числа для хеш-таблиц

Эта таблица наибольших простых чисел, меньших 2n, для $8 \leq n\leq 32$, может использоваться для динамического распределения хеш-таблицы, когда нужно, чтобы размер таблицы был простым числом. Для любого данного положительного значения в охваченном диапазоне эту таблицу можно использовать для определения простого числа, отличающегося от него менее чем в 2 раза.

Другой вариант обработки целочисленных ключей - объединение мультипликативного и модульного методов: нужно умножить ключ на константу в диапазоне между 0 и 1, а затем выполнить деление по модулю M.

Другими словами, необходимо использовать функцию $h(k)= \lfloor{k \alpha}\rfloor \mod {M}$. Между значениями $\alpha$, M и эффективным основанием системы счисления ключа существует взаимосвязь, которая теоретически могла бы привести к аномальному поведению, но если использовать произвольное значение a, в реальном приложении вряд ли возникнет какая-либо проблема. Часто в качестве a выбирают значение α = 0,618033... (золотое сечение).

Изучено множество других вариаций на эту тему, в частности, хеш-функции, которые могут быть реализованы с помощью таких эффективных машинных инструкций, как сдвиг и выделение по маске .

Во многих приложениях, в которых используются таблицы символов, ключи не являются числами и не обязательно являются короткими; чаще это алфавитно-цифровые строки, которые могут быть весьма длинными. Ну и как вычислить хеш-функцию для такого слова, как averylongkey?

В 7-разрядном ASCII-коде этому слову соответствует 84-разрядное число \begin{align\*} 97 \cdot 128^{11} &+ 118 \cdot 128^{10} + 101 \cdot 128^{9} + 114 \cdot 128^{8} + 121 \cdot 128^{7}\\ &+ 108 \cdot 128^{6} + 111 \cdot 128^{5} + 110 \cdot 128^{4} + 103 \cdot 128^{3}\\ &+ 107 \cdot 128^{2} + 101 \cdot 128^{1} + 121 \cdot 128^{0}, \end{align\*},

которое слишком велико, чтобы с ним можно было выполнять обычные арифметические функции в большинстве компьютеров. А зачастую требуется обрабатывать и гораздо более длинные ключи.

Чтобы вычислить модульную хеш-функцию для длинных ключей, они преобразуются фрагмент за фрагментом. Можно воспользоваться арифметическими свойствами функции модуля и использовать алгоритм Горнера. Этот метод основан на еще одном способе записи чисел, соответствующих ключам. Для рассматриваемого примера запишем следующее выражение: \begin{align\*} ((((((((((97 \cdot 128^{11} &+ 118) \cdot 128^{10} + 101) \cdot 128^{9} + 114) \cdot 128^{8} + 121) \cdot 128^{7}\\ &+ 108) \cdot 128^{6} + 111) \cdot 128^{5} + 110) \cdot 128^{4} + 103) \cdot 128^{3}\\ &+ 107) \cdot 128^{2} + 101) \cdot 128^{1} + 121. \end{align\*}

То есть десятичное число, соответствующее символьной кодировке строки, можно вычислить при просмотре ее слева направо, умножая накопленное значение на 128, а затем добавляя кодовое значение следующего символа. В случае длинной строки этот способ вычисления в конце концов приведет к числу, большему того, которое вообще можно представить в компьютере. Однако это число и не нужно, поскольку требуется только (небольшой) остаток от его деления на M. Результат можно получить, даже не сохраняя большое накопленное значение, т.к. в любой момент вычисления можно отбросить число, кратное M - при каждом выполнении умножения и сложения нужно хранить только остаток от деления по модулю M. Результат будет таким же, как если бы у нас имелась возможность вычислить длинное число, а затем выполнять деление (см. упражнение 14.10). Это наблюдение ведет к непосредственному арифметическому способу вычисления модульных хеш-функций для длинных строк - см. программу 14.1. В этой программе используется еще одно, последнее ухищрение: вместо основания 128 в ней используется простое число 127. Причина этого изменения рассматривается в следующем абзаце.

Существует множество способов вычисления хеш-функций приблизительно с теми же затратами, что и для модульного хеширования с использованием метода Горнера (одна-две арифметические операции для каждого символа в ключе). Для случайных ключей эти методы практически не отличаются друг от друга, но реальные ключи редко бывают случайными. Возможность ценой небольших затрат придать реальным ключам случайный вид приводит к рассмотрению рандомизированных алгоритмов хеширования, поскольку нам требуются хеш-функции, которые создают случайные индексы таблицы независимо от распределения ключей. Рандомизацию организовать нетрудно, поскольку вовсе не требуется буквально придерживаться определения модульного хеширования - нужно всего лишь, чтобы в вычислении целого числа, меньшего M, использовались все разряды ключа.

*Пример 1. Хеш-функция для строковых ключей*

Данная реализация хеш-функции для строковых ключей использует одно умножение и одно сложение для каждого символа в ключе. Если константу 127 заменить на 128, программа просто вычисляла бы методом Горнера остаток от деления числа, соответствующего 7-разрядному ASCII-представлению ключа, на размер таблицы. Простое основание, равное 127, помогает избежать аномалий, которые возникают, если размер таблицы является степенью 2 или кратным 2.

int hash(char \*v, int M)

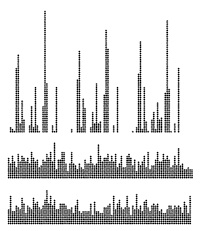
{ int h = 0, a = 127;

for (; \*v != 0; v++)

h = (a\*h + \*v) % M;

return h;

}



**Рис. 5.** Хеш-функции для символьных строк

На этих диаграммах показано распределение для набора английских слов (первые 1000 различных слов романа Мел-вилла " Моби Дик ") при использовании примере 1 с

M = 96 и a = 128 (вверху),

M = 97 и a = 128 (в центре) и

M = 96 и a = 127 (внизу)

Неравномерное распределение в первом случае является результатом неравномерного употребления букв и сохранения неравномерности из-за того, что и размер таблицы, и множитель кратны 32. Два других примера выглядят случайными, поскольку размер таблицы и множитель являются взаимно простыми числами.

В примере 1 показан один из способов сделать это: использование простого основания вместо степени 2 и целого числа, соответствующего ASCII-представлению строки. На рис. 5 показано, как это изменение улучшает распределение для типичных строковых ключей. Теоретически хеш-значения, созданные примере 1, могут давать плохие результаты для размеров таблицы, которые кратны 127 (хотя на практике это, скорее всего, будет почти незаметно); для создания рандомизированного алгоритма можно было бы выбрать значение множителя наугад. Еще более эффективный подход - использование случайных значений коэффициентов в вычислении и различных случайных значений для каждой цифры ключа. Такой подход дает рандомизированный алгоритм, называемый универсальным хешированием (universal hashing).

Теоретически идеальная универсальная хеш-функция - это функция, для которой вероятность коллизии между двумя различными ключами в таблице размером M в точности равна 1/M. Можно доказать, что использование в качестве коэффициента а в примере 1 не фиксированного произвольного значения, а последовательности случайных различных значений преобразует модульное хеширование в универсальную хеш-функцию. Однако затраты на генерирование нового случайного числа для каждого символа в ключе обычно неприемлемы. На практике можно достичь компромисса, показанного в примере 1, не храня массив различных случайных чисел для каждого символа ключа, а варьируя коэффициенты с помощью генерации простой псевдослучайной последовательности.

**ПОДВЕДЕМ ИТОГИ:**

чтобы для реализации абстрактной таблицы символов использовать хеширование, сначала необходимо расширить интерфейс абстрактного типа, включив в него операцию hash, которая отображает ключи на неотрицательные целые числа, меньшие размера таблицы M.

Непосредственная реализация

inline int hash(Key v, int M)

{ return (int) M\*(v-s)/(t-s); }

выполняет эту задачу для ключей с плавающей точкой со значениями между s и t; для целочисленных ключей можно просто вернуть значение v % M. Если M не является простым числом, хеш-функция может возвращать

(int) (.616161 \* (float) v) % M

или результат аналогичного целочисленного выражения, вроде

(16161 \* (unsigned) v) % M

Все эти функции, включая пример 1 для работы со строковыми ключами, проверены временем; они равномерно распределяют ключи и служат программистам в течение многих лет. Универсальный метод, представленный в примере 2 - заметное усовершенствование для строковых ключей, которое обеспечивает случайные хеш-значения при небольших дополнительных затратах; аналогичные рандомизированные методы можно применять и к целочисленным ключам.

В конкретном приложении универсальное хеширование может работать значительно медленнее, чем более простые методы, поскольку при наличии длинных ключей выполнение двух арифметических операций для каждого символа в ключе может потребовать слишком много времени. Для обхода этого ограничения ключи можно обрабатывать большими фрагментами. Действительно, можно использовать наибольшие фрагменты, которые помещаются в машинное слово, так же, как и в случае с элементарным модульным хешированием. Во многих ситуациях эти факторы важно учитывать, поскольку вычисление хеш-функции может выполняться во внутреннем цикле - следовательно, ускорив хеш-функцию, можно ускорить все вычисление.

Несмотря на очевидные преимущества рассмотренных методов, их реализация требует внимания по двум причинам. Во-первых, необходимо быть внимательным во избежание ошибок при преобразовании типов и использовании арифметических функций для различных машинных представлений ключей.

Пример *2. Универсальная хеш-функция (для строковых ключей)*

Эта программа выполняет те же вычисления, что и примере 1, однако для аппроксимации вероятности возникновения конфликтов для двух несовпадающих ключей до значения 1/M вместо фиксированных оснований системы счисления применяются псевдослучайные значения коэффициентов. Для минимизации нежелательных затрат времени при вычислении хеш-функции используется грубый генератор случайных чисел.

int hashU(char \*v, int M)

{ int h, a = 31415, b = 27183;

for (h = 0; \*v != 0; v++, a = a\*b % (M-1))

h = (a\*h + \*v) % M;

return (h < 0) ? (h + M) : h;

}

Такие операции часто являются источниками ошибок, особенно при переносе программы со старого компьютера на новый, с другим количеством разрядов в слове или прочими отличиями в точности выполнения операций. Во-вторых, весьма вероятно, что во многих приложениях вычисление хеш-функции будет выполняться во внутреннем цикле, и время ее выполнения может в значительной степени определять общее время выполнения. В подобных случаях важно убедиться, что функция сводится к эффективному машинному коду. Подобные операции - известные источники неэффективности; например, разница во времени выполнения простого модульного метода для целых чисел и версии, в которой вначале выполняется умножение ключа на 0,61616, может быть существенной. Наиболее быстрый метод для многих компьютеров - принять M равным степени 2 и воспользоваться хеш-функцией

inline int hash(Key v, int M)

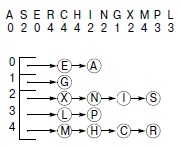
{ return v & (M-1); }

Эта функция использует только lg M - 1 младших разрядов ключей. Для устранения нежелательных эффектов плохого распределения ключей можно применять операцию побитового " И ", которая выполняется существенно быстрее и проще, чем другие операции.

Типичная ошибка в реализациях хеширования заключается в том, что хеш-функция всегда возвращает одно и то же значение - возможно, из-за неправильного выполнения требуемого преобразования типов. Такая ошибка называется ошибкой производительности, поскольку использующая такой метод хеширования программа вполне может выполняться корректно, но крайне медленно (т.к. ее эффективная работа возможна только при равномерном распределении хеш-значений). Однострочные реализации этих функций очень легко тестировать, поэтому настоятельно рекомендуется проверять, насколько успешно они работают для типов ключей, которые ожидаются в любой конкретной реализации таблицы символов.

Рассмотрим метод, называемый цепочками переполнения (separate chaining), поскольку конфликтующие элементы объединяются в отдельные связные списки-цепочки. Пример таких цепочек приведен на рис. 6. Как и в случае элементарного последовательного поиска, эти списки можно хранить упорядоченными или оставить неупорядоченными. Здесь присутствует тот же основной компромисс: для цепочек переполнения более важна не экономия времени (поскольку списки невелики), а экономия памяти (поскольку списков много).

Для упрощения кода вставки в упорядоченный список можно было бы использовать ведущий узел, но применение M ведущих узлов для отдельных цепочек переполнения не всегда удобно.



**Рис. 6.** Хеширование с цепочками переполнения

Здесь показан результат вставки ключей A S E R C H I N G X M P L в первоначально пустую хеш-таблицу с цепочками переполнения (неупорядоченные списки); используются хеш-значения, приведенные вверху. A попадает в список 0, затем S попадает в список 2, E - в список 0 (в его начало, чтобы время вставки было постоянным), R - в список 4 и т.д.

**Пример *3. Хеширование с цепочками переполнения***

Данная реализация таблицы символов основана на замене конструктора ST и функций search и insert в таблице символов с применением связных списков из программы 12.6 на приведенные здесь функции, а также на замене ссылки head на массив ссылок heads. Здесь используются рекурсивные функции поиска и удаления в списке, и при этом используются M списков с ведущими ссылками в heads, с использованием хеш-функции для выбора одного из списков. Конструктор устанавливает M так, что каждый список будет содержать около пяти элементов; поэтому для выполнения остальных операций требуется всего несколько проверок.

private:

link\* heads;

int N, M;

public:

ST(int maxN)

{ N = 0; M = maxN/5;

heads = new link[M];

for (int i = 0; i < M; i++) heads[i] = 0;

}

Item search(Key v)

{ return searchR(heads[hash(v, M)], v); }

void insert(Item item)

{ int i = hash(item.key(), M);

heads[i] = new node(item, heads[i]); N++;

}

Более того, для элементов примитивного типа можно было бы даже исключить M ссылок на эти списки, поместив первые узлы списков в саму таблицу. Для неудачных поисков можно считать, что хеш-функция достаточно равномерно перемешивает значения ключей, чтобы поиск в каждом из M списков был равновероятным.

***Лемма 1.*** Цепочки переполнения уменьшают количество сравнений, выполняемых при последовательном поиске, в M раз (в среднем) и используют дополнительный объем памяти для M ссылок.

Чаще всего для цепочек переполнения используются неупорядоченные списки, поскольку этот подход прост в реализации и эффективен: операция вставить выполняется за постоянное время, а операция найти - за время, пропорциональное N/M. Если ожидается очень большое количество неудачных поисков, их обнаружение можно ускорить в два раза, храня списки в упорядоченном виде, но за счет замедления операции вставить.

***Лемма 2.*** В хеш-таблице с цепочками переполнения, содержащей M списков и N ключей, вероятность того, что количество ключей в каждом списке отличается от N/M на небольшой постоянный коэффициент, очень близка к 1.

Замечание: Обычно в реализациях цепочек переполнения значение M выбирают достаточно малым, чтобы не тратить понапрасну большие непрерывные участки памяти с пустыми ссылками, но достаточно большим, чтобы последовательный поиск в списках был наиболее эффективным методом. Гибридные методы (вроде использования бинарных деревьев вместо связных списков), вряд ли стоят рассмотрения. Как правило, можно выбирать значение M равным приблизительно одной пятой или одной десятой от ожидаемого количества ключей в таблице, чтобы каждый из списков в среднем содержал порядка 5-10 ключей. Одно из достоинств цепочек переполнения состоит в том, что этот выбор не критичен: при наличии большего, чем ожидалось, количества ключей поиски будут требовать несколько больше времени, чем если бы заранее был выбран больший размер таблицы; при наличии в таблице меньшего количества ключей поиск будет сверхбыстрым при (скорее всего) небольшом дополнительном расходе памяти. Если памяти хватает, значение M можно выбрать достаточно большим, чтобы время поиска было постоянным; если же объем памяти критичен, все-таки можно повысить производительность в M раз, выбрав максимально возможное значение M.

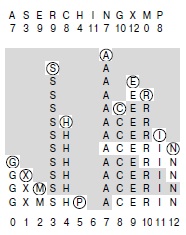
На практике для цепочек переполнения обычно применяются неупорядоченные списки, по двум основным причинам. Во-первых, исключительно быстро выполняется операция вставить: вычисляем хеш-функцию, выделяем память для узла и связываем его с началом соответствующего списка. Во многих приложениях шаг распределения памяти не требуется (поскольку элементами, вставляемыми в таблицу символов, могут быть существующие записи с доступными полями ссылок), и для выполнения операции вставить остается выполнить всего три-четыре машинные инструкции. Второе важное преимущество использования в примере 3 реализации с использованием неупорядоченных списков заключается в том, что все списки работают подобно стекам, поэтому можно легко удалить последние вставленные элементы, которые размещаются в начале списков. Эта операция важна при реализации таблицы символов с вложенными областями видимости (например, в компиляторе).

В общем случае хеширование не подходит для использования в приложениях, в которых требуются реализации операций АТД сортировать и выбрать. Однако хеширование часто используется в типичных ситуациях, когда необходимо использовать таблицу символов с потенциально большим количеством операций найти, вставить и удалить с последующим однократным выводом элементов в порядке их ключей. Одним из примеров такого приложения является таблица символов в компиляторе; другой пример - программа удаления повторяющихся ключей. В реализации с использованием упорядоченных списков сортировку можно выполнить слиянием всех за время, пропорциональное Nlg M.

#### **ЛИНЕЙНОЕ ОПРОБОВАНИЕ**

Если можно заранее оценить количество элементов, которые должны быть помещены в хеш-таблицу, и имеется достаточно большая непрерывная область памяти, в которой можно поместить все ключи и оставить еще свободные участки, то тогда в хеш-таблице, вероятно, вообще не стоит использовать какие-либо ссылки. Существует несколько методов хранения N элементов в таблице размером M > N, при которых разрешение коллизий основано на использовании пустых мест в таблице. Такие методы называются методами хеширования с открытой адресацией (open-addressing).

Простейший метод открытой адресации называется линейным опробованием (linear probing): при возникновении коллизии (когда хеширование дает адрес в таблице, который уже занят элементом с ключом, не совпадающим с ключом поиска) мы просто проверяем следующую позицию в таблице. Обычно такую проверку (определяющую, содержит ли данная позиция таблицы элемент с ключом, равным ключу поиска) называют пробой (probe). При линейном опробовании определяется один из трех возможных исходов пробы: если позиция таблицы содержит элемент, ключ которого совпадает с искомым, то поиск завершился успешно; в противном случае (если позиция таблицы содержит элемент, ключ которого не совпадает с искомым) мы просто проверяем позицию таблицы с большим индексом, продолжая этот процесс (с возвратом к началу таблицы при достижении ее конца) до тех пор, пока не будет найден искомый ключ или пустая позиция таблицы. Если элемент, содержащий искомый ключ, должен быть вставлен после неудачного поиска, он помещается в пустое место таблицы, где был завершен поиск. Примере 4 является реализацией АТД таблицы символов, использующей этот метод. Процесс построения хеш-таблицы с использованием линейного опробования для некоторого набора ключей показан на рис. 7.



**Рис. 7.** Хеширование методом линейного опробования

На этой диаграмме показан процесс вставки ключей A S E R C H I N G X M P в первоначально пустую хеш-таблицу с открытой адресацией, размер которой равен 13. Используются показанные вверху хеш-значения и разрешение коллизий методом линейного опробования. Вначале A попадает в позицию 7, затем S попадает в позицию 3, E - в позицию 9. Потом, после коллизии в позиции 9, R попадает в позицию 10 и т.д. При достижении правого конца таблицы опробование продолжается с левого конца: например, последний вставленный ключ P хешируется в позицию 8, но после коллизий в позициях 8-12 и 0-4 попадает в позицию 5. Неопробованные позиции таблицы затенены.

Как и в случае цепочек переполнения, производительность методов с открытой адресацией зависит от коэффициента $\alpha = N/M$, но при этом он интерпретируется иначе.

*Пример 4. Линейное опробование*

Данная реализация таблицы символов хранит элементы в таблице, размер которой вдвое превышает максимально ожидаемое количество элементов и которая первоначально содержит значения nullItem. Таблица содержит сами элементы, но если элементы велики, тип элемента можно изменить, чтобы таблица содержала ссылки на элементы.

Для вставки нового элемента выполняется хеширование в позицию таблицы и ее просмотр вправо (чтобы найти незанятую позицию), используя пустые элементы в незанятых позициях в качестве сигнальных, точно так же, как и при поиске с индексацией по ключам. Для поиска элемента с данным ключом мы начинаем с хеш-позиции ключа в таблице и просматриваем ее на совпадение, завершая процесс при обнаружении незанятой позиции.

Конструктор устанавливает M таким образом, чтобы таблица была заполнена менее чем наполовину, поэтому, если хеш-функция выдает значения, похожие на случайные, для выполнения остальных операций потребуется всего несколько проб.

private:

Item \*st;

int N, M;

Item nullItem;

public:

ST(int maxN)

{ N = 0; M = 2\*maxN;

st = new Item[M];

for (int i = 0; i < M; i++) st[i] = nullItem;

}

int count() const

{ return N; }

void insert(Item item)

{ int i = hash(item.key(), M);

while (!st[i].null()) i = (i+1) % M;

st[i] = item; N++;

}

Item search(Key v)

{ int i = hash(v, M);

while (!st[i].null())

if (v == st[i].key())

return st[i];

else

i = (i+1) % M;

return nullItem;

}

В случае цепочек переполнения - среднее количество элементов в одном списке, обычно большее 1. В случае открытой адресации а - доля занятых позиций таблицы; она должна быть меньше 1. Иногда а называют коэффициентом загрузки хеш-таблицы.

В случае разреженной таблицы (значение $\alpha$ мало) очевидно, что для большинства операций поиска пустая позиция будет найдена после всего нескольких проб. В случае почти полной таблицы (значение близко $\alpha$ к 1) для выполнения поиска может потребоваться очень большое количество проб, а при полностью заполненной таблице поиск может даже привести к бесконечному циклу. Как правило, чтобы время поиска не было слишком большим, при использовании линейного опробования нужно не допускать заполнения таблицы. То есть вместо того, чтобы использовать дополнительную память для ссылок, она используется для создания дополнительного места в хеш-таблице, что позволяет сократить последовательности проб. При использовании линейного опробования размер таблицы больше, чем с цепочками переполнения, т.к. необходимо соблюдение условия M > N, но общий объем используемой памяти может быть меньше, поскольку не используются ссылки.

Средние затраты на выполнение линейного опробования зависят от того, как элементы при их вставке объединяются в непрерывные группы занятых ячеек таблицы, называемые кластерами (cluster). Рассмотрим следующие два крайних случая заполненной наполовину (M = 2N) таблицы линейного опробования. В лучшем случае позиции таблицы с четными индексами будут пустыми, а с нечетными - занятыми (или наоборот - прим. перев.). В худшем случае первая (вообще-то любая непрерывная - прим. перев.) половина позиций таблицы будет пустой, а вторая - заполненной. Средняя длина кластеров в обоих случаях равна N/(2N) = 1/2, но среднее количество проб при неудачном поиске равно 1 (нужна по меньшей мере одна проба) плюс $$(0 + 1 + 0 + 1 + ...) / (2N ) = 1/2$$

в лучшем случае и 1 плюс $$(N + (N - 1) + (N - 2) + ...) / (2N) \approx N/4$$

в худшем случае.

Обобщая эти рассуждения, приходим к выводу, что среднее количество проб при неудачном поиске пропорционально квадратам длин кластеров. Среднее значение рассчитывается путем вычисления затрат при неудачном поиске, начиная с каждой позиции таблицы, и деления суммы на M. Для неудачного поиска требуется не менее 1 пробы, поэтому будем считать количество проб, следующих за первой. Если кластер имеет дину t, то вклад этого кластера в общую сумму определяется выражением

$$(t + (t - 1) + ... + 2 + 1) / М = t (t + 1) / (2М )$$

Сумма длин кластеров равна N, поэтому, суммируя эти затраты для всех ячеек в таблице, находим, что общие средние затраты при неудачном поиске равны 1 + N / (2M) плюс сумма квадратов длин кластеров, деленная на 2M. Имея заданную таблицу, можно быстро вычислить средние затраты на неудачный поиск в этой таблице (см. упражнение 14.28), но общую аналитическую формулу дать трудно, поскольку кластеры образуются в результате сложного динамического процесса (алгоритма линейного опробования).

*Лемма 3.* При разрешении коллизий с помощью линейного опробования среднее количество проб, нужных для поиска в хеш-таблице размером M, которая содержит $N = \alpha М$ ключей, приблизительно равно

$\dfrac{1}{2}\left(1+\dfrac{1}{1-\alpha }\right)$ и

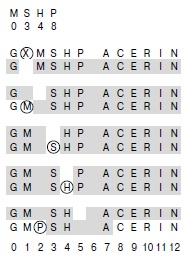
$\dfrac{1}{2}\left(1+\dfrac{1}{(1-\alpha)^{2}}\right)$

соответственно для успешного и неудачного поиска.

Точность этих выражений уменьшается с приближением значения $\alpha$ к 1, но в данном случае это не важно, поскольку в любом случае линейное опробование не следует использовать в почти заполненной таблице. Для меньших значений а равенства достаточно точны. Ниже приведена таблица, в которую сведены ожидаемые количества проб при успешном и неудачном поиске с использованием линейного опробования:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **коэффициент загрузки ($\alpha$ )** | **1/2** | **2/3** | **3/4** | **9/10** |
| успешный поиск | 1,5 | 2,0 | 3,0 | 5,5 |
| неудачный поиск | 2,5 | 5,0 | 8,5 | 55,5 |

Для неудачного поиска всегда требуются большие затраты, чем для успешного, и можно ожидать, что в обоих случаях в заполненной менее чем наполовину таблице в среднем требуется лишь несколько проб.



**Рис. 8.** Удаление в хеш-таблице с линейным опробованием

На этой диаграмме показан процесс удаления ключа X из таблицы, показанной на рис. 8. Во второй строке показан результат простого удаления X из таблицы, что неприемлемо, поскольку в этом случае M и P оказываются отрезаны от своих хеш-позиций пустой позицией, оставшейся после X. Поэтому ключи M, S, H и P (справа от X в этом же кластере) повторно вставляются в указанном порядке с использованием хеш-значений, указанных сверху, и с разрешением коллизий с помощью линейного опробования. M заполняет свободное место, оставленное ключом X, потом в таблицу без коллизий вставляются S и H, а затем в позицию 2 вставляется P.

Как и в случае цепочек переполнения, выбор, хранить ли в таблице элементы с повторяющимися ключами, предоставляется клиенту. Такие элементы не обязательно размещаются в таблице с линейным опробованием в соседних позициях - среди элементов с одинаковыми ключами могут размещаться и другие элементы с таким же хеш-значением.

По самой сути построения ключи в таблице, построенной с помощью линейного опробования, размещаются в случайном порядке. В результате операции АТД сортировать и выбрать требуют выполнения заново по одному из методов, описанных в лекциях 6-10. Поэтому линейное опробование не годится для приложений, в которых эти операции выполняются часто.

А как удалить ключ из таблицы, построенной с помощью линейного опробования? Просто убрать его нельзя, поскольку элементы, которые были вставлены позже, могли перескочить через этот элемент, и поэтому их поиск будет постоянно прерываться на пустой позиции, оставшейся после удаленного элемента. Одно из решений этой проблемы заключается в повторном хешировании всех элементов, для которых эта проблема могла бы возникнуть - между удаленным элементом и следующей незанятой позицией справа от него. Пример, иллюстрирующий этой процесс, приведен на рис. 8, а пример 5 содержит реализацию этого подхода. В разреженной таблице в большинстве случаев такой процесс потребует лишь нескольких операций повторного хеширования. Другой способ реализации удаления - замена удаленного ключа сигнальным ключом, который будет служить заполнителем для поиска, но может быть повторно использован для вставок (.

*Пример 5. Удаление из хеш-таблицы с линейным опробованием*

Для удаления элемента с заданным ключом выполняется поиск этого элемента и замена его пустым элементом nullItem. Затем необходимо внести изменения на случай, если какой-либо элемент, расположенный справа от теперь незанятой позиции, хешируется в эту позицию или левее нее, т.к. свободная позиция может привести к прерыванию поиска такого элемента. Поэтому выполняется повторная вставка всех элементов, расположенных в одном кластере с удаленным элементом правее него. Поскольку таблица заполнена менее чем наполовину, в среднем количество повторно вставляемых элементов будет мало.

void remove(Item x)

{ int i = hash(x.key(), M), j;

while (!st[i].null())

if (x.key() == st[i].key())

break;

else

i = (i+1) % M;

if (st[i].null()) return;

st[i] = nullItem; N--;

for (j = i+1; !st[j].null(); j = (j + 1) % M, N--)

{ Item v = st[j]; st[j] = nullItem; insert(v); }

}

#### **ДВОЙНОЕ ХЕШИРОВАНИЕ**

Основной принцип линейного опробования (а, вообще-то, и любого метода хеширования) -гарантирование того, что при поиске конкретного ключа мы просматриваем каждый ключ, который отображается в тот же адрес в таблице (в частности, сам ключ, если он есть в таблице). Однако при использовании схемы с открытой адресацией, как правило, просматриваются и другие ключи, особенно когда заполнение таблицы велико. В примере, приведенном на рис. 7, при поиске ключа N просматриваются ключи C, E, R и I, ни один из которых не имеет такого же хеш-значения. Что еще хуже, вставка ключа с одним хеш-значением может существенно увеличить время поиска ключей с другими хеш-значениями: на рис. 7 вставка ключа M приводит к увеличению времени поиска для позиций 7-12 и 0-1. Это явление называется кластеризацией (clustering), поскольку оно связано с процессом образования кластеров. Для почти заполненных таблиц оно может значительно замедлять линейное опробование.

К счастью, существует простой способ избежать возникновения проблемы кластеризации - двойное хеширование (double hashing). Основная стратегия остается той же, что и при выполнении линейного опробования; единственное различие состоит в том, что вместо просмотра каждой позиции таблицы, следующей за коллизией, мы используем вторую хеш-функцию для получения постоянного шага, используемого для последовательных проб. Реализация этого способа приведена в примере 6.

*Пример 6. Двойное хеширование*

Двойное хеширование аналогично линейному опробованию, за исключением того, что здесь используется вторая хеш-функция - для определения шага поиска, используемого после каждой коллизии. Шаг поиска должен быть ненулевым, а размер таблицы и шаг поиска должны быть взаимно простыми числами. Функция remove для линейного опробования не работает с двойным хешированием, поскольку любой ключ может присутствовать во многих различных последовательностях проб.

void insert(Item item)

{ Key v = item.key();

int i = hash(v, M), k = hashtwo(v, M);

while (!st[i].null()) i = (i+k) % M;

st[i] = item; N+ + ;

}

Item search(Key v)

{ int i = hash(v, M), k = hashtwo(v, M);

while (!st[i].null())

if (v == st[i].key())

return st[i];

else

i = ( i+k) % M;

return nullItem;

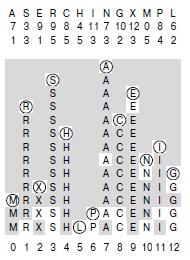
}

Выбор второй хеш-функции требует определенной осторожности, поскольку в противном случае программа может вообще не работать. Во-первых, необходимо исключить случай, когда вторая хеш-функция дает нулевое значение, поскольку при первой же коллизии это приведет к бесконечному циклу. Во-вторых, важно, чтобы значение второй хеш-функции и размер таблицы были взаимно простыми числами, т.к. иначе некоторые из последовательностей проб могут оказаться очень короткими (например, в случае, когда размер таблицы вдвое больше значения второй хеш-функции). Один из способов претворения этих правил в жизнь - выбор в качестве M простого числа и выбор второй хеш-функции, возвращающей значения, меньшие M. На практике во многих ситуациях, если размер таблицы не слишком мал, будет достаточно простой второй хеш-функции вроде

inline int hashtwo(Key v) { return (v % 97) + 1; }

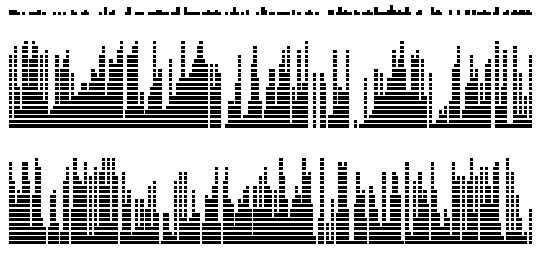
Кроме того, любое снижение эффективности (из-за уменьшения дисперсии), вызываемое данным упрощением, на практике, скорее всего, будет не только не важно, но и вовсе незаметно. Если таблица большая и разреженная, сам размер таблицы не обязательно должен быть простым числом, поскольку для каждого поиска нужно будет лишь несколько.

На рис. 9 показан процесс построения небольшой таблицы методом двойного хеширования, а из рис. 10 видно, что двойное хеширование приводит к образованию значительно меньшего количества кластеров (которые поэтому значительно короче), чем в результате линейного опробования.



**Рис. 9.** Двойное хеширование

На этой диаграмме показан процесс вставки ключей A S E R C H I N G X M P L в первоначально пустую хеш-таблицу с открытой адресацией с использованием хеш-значений, приведенных вверху, и разрешением коллизий с помощью двойного хеширования. Первое и второе хеш-значения каждого ключа приведены в двух строках под этим ключом. Как и на рис. 7, проверяемые позиции таблицы выделены белым цветом. Ключ A попадает в позицию 7, затем S попадает в позицию 3, E - в позицию 9. Но ключ R после коллизии в позиции 9 попадает в позицию 1; его второе хеш-значение, равное 5, используется в качестве шага последовательности проб после коллизии. Аналогично, ключ P окончательно попадает в позицию 6 после коллизий в позициях 8, 12, 3, 7, 11 и 2 при использовании в качестве шага его второго хеш-значения, равного 4.



**Рис. 10.** Кластеризация

На этих диаграммах показано размещение записей при их вставке в хеш-таблицу с использованием линейного опробования (в центре) и двойного хеширования (внизу), при распределении ключей, показанном на верхней диаграмме. Каждая черточка показывает результат вставки 10 записей. По мере заполнения таблицы записи объединяются в кластеры, разделенные пустыми позициями таблицы. Длинные кластеры нежелательны, т.к. средние затраты на поиск одного ключа в кластере пропорциональны длине кластера. В случае линейного опробования чем длиннее кластеры, тем более вероятно их удлинение, поэтому по мере заполнения таблицы оказываются доминирующими несколько длинных кластеров. При использовании двойного хеширования этот эффект значительно менее выражен, и кластеры остаются сравнительно короткими.

Лемма 4. При разрешении коллизий с помощью двойного хеширования среднее количество проб, необходимых для выполнения поиска в хеш-таблице размером M, содержащей $N = \alpha M$ ключей, равно $\dfrac{1}{\alpha}\ln\left(\dfrac{1}{1-\alpha}\right)$ и $\dfrac{1}{1-\alpha}$

соответственно для успешного и неудачного поиска.

Эти формулы - результат глубокого математического анализа, выполненного Гиба (Guibas) и Шемереди (Szemeredi).

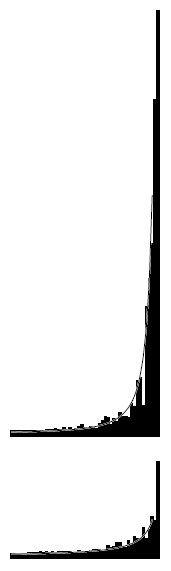
Точная природа взаимосвязи между производительностью двойного хеширования и идеальным случаем случайного хеширования, установленной Гиба и Шемереди - асимптотическое приближение, которое не обязательно должно быть справедливо для используемых на практике размеров таблиц. Кроме того, полученные результаты основываются на предположении, что хеш-функции возвращают случайные значения. Но все же асимптотические формулы из свойства 14.5 на практике позволяют достаточно точно предсказать производительность двойного хеширования, даже при использовании такой просто вычисляемой второй хеш-функции, как (v % 97)+1. Как и соответствующие формулы для линейного опробования, при приближении значения а к 1 эти формулы стремятся к бесконечности, но гораздо медленнее.

Различие между линейным опробованием и двойным хешированием хорошо видно на рис. 11. В разреженных таблицах двойное хеширование и линейное опробование имеют схожую производительность, но при использовании двойного хеширования можно допустить значительно большую степень заполнения таблицы, чем при использовании линейного опробования, прежде чем производительность заметно снизится. В следующей таблице приведено ожидаемое количество проб для успешного и неудачного поиска при использовании двойного хеширования:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **коэффициент загрузки ($\alpha$)** | **1/2** | **2/3** | **3/4** | **9/10** |
| успешный поиск | 1,4 | 1,6 | 1,8 | 2,6 |
| неудачный поиск | 1,5 | 2,0 | 3,0 | 5,5 |

Неудачный поиск всегда требует больших затрат, чем успешный, но и тому, и другому в среднем требуется лишь несколько проб, даже в таблице, заполненной на девять десятых.

Можно взглянуть на эти результаты и под другим углом: для получения такого же среднего времени поиска двойное хеширование позволяет использовать меньшие таблицы, чем нужно при использовании линейного опробования.



**Рис. .** Затраты на выполнение поиска с открытой адресацией

На этих графиках показаны затраты на построение хеш-таблицы размером 1000 вставками ключей в первоначально пустую таблицу с помощью линейного опробования (вверху) и двойного хеширования (внизу). Каждый вертикальный столбец представляет затраты на вставку 20 ключей.

***Лемма 5.*** Сохраняя коэффициент загрузки меньшим, чем $1 — 1/\sqrt{t}$ для линейного опробования и меньшим, чем $1 — 1/ t$ для двойного хеширования, можно обеспечить, что в среднем для выполнения всех поисков потребуется меньше t проб.

Например, чтобы среднее количество проб при поиске было меньшим 10, при использовании линейного опробования необходимо сохранять таблицу пустой не менее чем на 32%, а при использовании двойного хеширования - лишь на 10%. Чтобы можно было выполнить неудачный поиск менее чем за 10 проб при обработке 105 элементов, нужно свободное место лишь для 104 дополнительных элементов. В то же время при использовании цепочек переполнения потребовалась бы дополнительная память для более чем 105 ссылок, а при использовании деревьев бинарного поиска - еще вдвое больший объем памяти.

Метод реализации операции удалить, приведенный в программе 14.5 (повторное хеширование ключей, для которых путь поиска может содержать удаляемый элемент), для двойного хеширования не годится, т.к. удаленный элемент может присутствовать во многих различных последовательностях проб для ключей, разбросанных по всей таблице. Поэтому приходится применять другой метод: удаленный элемент заменяется сигнальным элементом, помечающим позицию таблицы как занятую, но не соответствующим ни одному из ключей.

Подобно линейному опробованию, двойное хеширование не годится в качестве основы для реализации полнофункционального АТД таблицы символов, в котором необходимо поддерживать операции сортировать или выбрать.

#### **ДИНАМИЧЕСКИЕ ХЕШ-ТАБЛИЦЫ**

С увеличением количества ключей в хеш-таблице скорость поиска снижается. При использовании цепочек переполнения время поиска увеличивается постепенно: если количество ключей в таблице удваивается, то и время поиска удваивается. Это же справедливо и для разреженных таблиц по отношению к таким методам с открытой адресацией, как линейное опробование и двойное хеширование, но по мере заполнения таблицы затраты существенно возрастают и, что гораздо хуже, наступает момент, когда невозможно вставить ни один ключ. Эта ситуация отличается от деревьев поиска, в которых рост происходит естественным образом. Например, в RB-дереве при каждом удвоении количества узлов затраты на поиск возрастают совсем ненамного (на одно сравнение).

Один из способов расширения хеш-таблиц - удвоение размера таблицы, когда она начинает заполняться. Удвоение размера таблицы - дорогостоящая операция, поскольку все элементы в таблице должны быть вставлены повторно, однако она выполняется нечасто. Пример 7 является реализацией увеличения таблицы с линейным опробованием путем ее удвоения. Пример таких удвоений показан на рис. 12.

*Пример 7. Динамическая вставка в хеш-таблицу (для линейного опробования)*

Данная реализация операции insert для линейного опробования обрабатывает произвольное количество ключей, удваивая размер таблицы при каждом заполнении таблицы наполовину (этот же подход может быть использован для двойного хеширования или цепочек переполнения). Удвоение требует распределения памяти для новой таблицы и повторного хеширования в нее всех ключей, а затем освобождения памяти, занимаемой старой таблицей. Функция-член init используется для построения или повторного построения таблицы, заполненной пустыми элементами указанных размеров.

private:

void expand()

{ Item \*t = st;

init(M+M);

for (int i = 0; i < M/2; i++)

if (!t[i].null()) insert(t[i]);

delete t;

}

public:

ST(int maxN)

{ init(4); }

void insert(Item item)

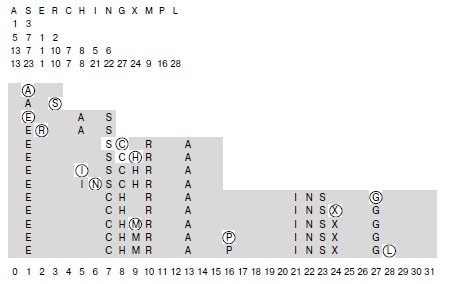
{ int i = hash(item.key(), M);

while (!st[i].null()) i = (i+1) % M;

st[i] = item;

if (N++ >= M/2) expand();

}



**Рис. 12.** Динамическое расширение хеш-таблицы

На этой диаграмме показан процесс вставки ключей A S E R C H I N G X M P L в динамическую хеш-таблицу, которая расширяется удвоением размера, с использованием хеш-значений, приведенных вверху, и разрешением коллизий с помощью линейного опробования. В четырех строках под ключами приводятся хеш-значения для размеров таблицы, равных 4, 8, 16 и 32. Начальный размер таблицы равен 4, затем, перед вставкой E, он удваивается до 8, перед вставкой C - до 16 и перед вставкой G - до 32. При каждом удвоении размера таблицы для всех ключей выполняются повторные хеширование и вставка. Все вставки выполняются в разреженные таблицы (заполненные менее чем на одну четверть для повторной вставки и от четверти до половины в остальных случаях), поэтому коллизий возникает мало.

Это же решение работает и для двойного хеширования, а основная идея применима и для цепочек переполнения . Каждый раз, когда таблица заполняется более чем наполовину, она расширяется путем удвоения ее размера. После первого расширения степень заполнения таблицы всегда составляет от одной четвертой до одной второй, поэтому средние затраты на поиск составляют менее трех проб. И хотя операция повторного построения таблицы является дорогостоящей, она выполняется столь редко, что ее стоимость составляет лишь постоянную долю общих затрат на построение таблицы.

Эту концепцию можно выразить и по-другому, сказав, что средние затраты на одну вставку составляют меньше четырех проб. Это утверждение не равносильно утверждению, что для каждой вставки требуется менее четырех проб; и действительно, мы знаем, что вставки, которые приводят к удвоению размера таблицы, требуют большого количества проб. Это рассуждение - простой пример амортизационного анализа: нельзя гарантировать быстрое выполнение каждой операции этого алгоритма, но можно гарантировать, что будут низкими средние затраты на одну операцию.

Хотя общие затраты низки, профиль производительности вставок неравномерен: большинство операций выполняется исключительно быстро, но иногда для некоторых операций требуется почти столько же времени, сколько ранее было затрачено на построение всей таблицы. При увеличении размера таблицы от 1 тысячи до 1 миллиона ключей это замедление может случиться около 10 раз. Такое поведение приемлемо во многих приложениях, но может оказаться недопустимым, если желательны или обязательны абсолютные гарантии производительности. Например, если банк или авиакомпания могут допустить, что 10 из миллиона клиентов будут ожидать длительное время, то в таких приложениях, как онлайновая система обработки финансовых транзакций или система управления авиаполетами, долгое ожидание может повлечь катастрофические последствия.

При поддержке операции АТД удалить может иметь смысл сжимать таблицу, уменьшая вдвое ее размер при уменьшении количества ее ключей. Но здесь требуется выполнение одного условия: границы уменьшения должны отличаться от границ увеличения, поскольку иначе небольшое количество операций вставить и удалить может привести даже для очень больших таблиц к серии операций увеличения и уменьшения размера вдвое.

***Лемма 6.*** Последовательность t операций найти, вставить и удалить в таблицах символов может быть выполнена за время, пропорциональное t, при использовании объема памяти, не превышающего числа ключей в таблице, умноженного на некоторый постоянный коэффициент.

#### **ПЕРСПЕКТИВЫ**

Как было показано при рассмотрении методов хеширования, выбор метода, который наиболее подходит для конкретного приложения, зависит от множества различных факторов. Все методы могут уменьшить время выполнения операций таблицы символов найти и вставить, сделав его постоянным, и все методы годятся для применения в широком множестве приложений. Можно грубо охарактеризовать три основных метода (линейное опробование, двойное хеширование и цепочки переполнения) следующим образом: линейное опробование является самым быстрым из этих трех методов (при наличии достаточного объема памяти, чтобы таблица была разреженной), двойное хеширование наиболее эффективно использует память (но тратит дополнительное время для вычисления второй хеш-функции), а цепочки переполнения проще всего реализовать и применять (при наличии хорошего механизма распределения памяти). Экспериментальные данные и комментарии, характеризующие производительность алгоритмов, приведены в таблица 1.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Таблица 14.1. Экспериментальное сравнение реализаций хеш-таблиц | | | | | | | | | | |
| **N** | **Создание** | | | | | **Неудачный поиск** | | | | |
| **R** | **H** | **P** | **h** | **P\*** | **R** | **H** | **P** | **h** | **P\*** |
| 1250 | 1 | 0 | 5 | 3 | 0 | 1 | 1 | 0 | 1 | 0 |
| 2500 | 3 | 1 | 3 | 4 | 2 | 1 | 1 | 0 | 0 | 0 |
| 5000 | 6 | 1 | 4 | 4 | 3 | 2 | 1 | 0 | 1 | 0 |
| 12500 | 14 | 6 | 5 | 5 | 5 | 6 | 1 | 2 | 2 | 1 |
| 25000 | 34 | 9 | 7 | 8 | 11 | 16 | 5 | 3 | 4 | 3 |
| 50000 | 74 | 18 | 11 | 12 | 22 | 36 | 15 | 8 | 8 | 8 |
| 100000 | 182 | 35 | 21 | 23 | 47 | 84 | 45 | 23 | 21 | 15 |
| 150000 |  | 54 | 40 | 36 | 138 |  | 99 | 89 | 52 | 21 |
| 160000 |  | 58 | 43 | 44 | 147 |  | 115 | 133 | 66 | 23 |
| 170000 |  | 68 | 55 | 45 | 136 |  | 121 | 226 | 85 | 25 |
| 180000 |  | 65 | 61 | 50 | 152 |  | 133 | 449 | 125 | 27 |
| 190000 |  | 79 | 106 | 59 | 155 |  | 144 | 2194 | 261 | 30 |
| 200000 | 407 | 84 |  |  | 159 | 186 | 156 |  |  | 33 |
| Обозначения: |  | | | | |
| R | RB-дерево бинарного поиска) | | | | |
| H | Цепочки переполнения при размере таблицы 20000 | | | | |
| P | Линейное опробование при размере таблицы 200000 | | | | |
| D | Двойное хеширование при размере таблицы 200000 | | | | |
| P | Линейное опробование с расширением путем удвоения | | | | |

Эти относительные значения времени построения таблиц символов из случайных последовательностей 32-битовых целых чисел и поиска в них подтверждают, что для легко хешируемых ключей хеширование работает значительно быстрее, чем поиск по дереву. Из всех методов хеширования двойное хеширование в разреженных таблицах работает медленнее, чем метод цепочек переполнения и линейное опробование (из-за затрат на вычисление второй хеш-функции), но значительно быстрее линейного опробования в почти заполненной таблице; кроме того, этот метод - единственный, который может обеспечить быстрый поиск с использованием лишь небольшого объема дополнительной памяти. Динамические хеш-таблицы, построенные с использованием линейного опробования и расширения удвоением, требуют больших затрат времени на создание, чем другие хеш-таблицы, из-за распределения памяти и повторного хеширования, но несомненно обеспечивают наиболее быстрый поиск. Этот метод удобен тогда, когда чаще всего выполняется поиск и заранее нельзя точно предвидеть количество ключей.

Выбор между линейным опробованием и двойным хешированием зависит прежде всего от затрат на вычисление хеш-функции и от коэффициента загрузки таблицы. Для разреженных таблиц (малых значений коэффициента $\alpha$) оба метода используют лишь несколько проб, но двойное хеширование может потребовать больше времени, если понадобится вычислять две хеш-функции для длинных ключей. По мере приближения а к 1 двойное хеширование начинает существенно превосходить по производительности линейное опробование .

Сравнение линейного опробования и двойного хеширования с методом цепочек переполнения выполнить сложнее, поскольку необходимо точно учитывать использование памяти. Цепочки переполнения используют дополнительную память под ссылки; методы с открытой адресацией неявно используют дополнительную память внутри таблицы для завершения последовательностей проб. Следующий конкретный пример иллюстрирует эту ситуацию. Предположим, что имеется таблица М списков, построенная хешированием с цепочками переполнения, что средняя длина списков равна 4, и что каждый элемент и каждая ссылка занимают по одному машинному слову. Предположение, что элементы и ссылки занимают одинаковый объем памяти, оправданно во многих ситуациях, поскольку очень большие элементы обычно заменяются ссылками на них. В этом случае таблица занимает 9М слов памяти (4М для элементов и 5М для ссылок), и требует для выполнения поиска в среднем 2 пробы. Но при линейном опробовании для 4М элементов в таблице размером 9М требуется всего $(1 + 1/(1 — 4/9))/2 = 1,4$ пробы для успешного поиска, что на 30% меньше, чем с цепочками переполнения при том же объеме используемой памяти; а при линейном опробовании для 4М элементов в таблице размером 6М для успешного поиска требуется (в среднем) 2 пробы и, следовательно, используется на 33% меньше памяти, чем с цепочками переполнения при том же времени выполнения. Кроме того, можно использовать динамический метод, наподобие программы 14.7, для сохранения небольшого коэффициента загрузки таблицы с помощью увеличения ее размера.

Приведенные рассуждения показывают, что обычно выбор цепочек переполнения вместо открытой адресации по соображениям производительности не оправдан. Однако на практике цепочки переполнения с фиксированным значением М часто выбирают по ряду других причин: их легко реализовать (особенно операцию удалить); они требуют небольшого дополнительного объема памяти для элементов с уже выделенными полями ссылок, пригодными для использования таблицей символов и другими АТД, которые могут в них нуждаться. Хотя производительность этого метода снижается с увеличением количества элементов в таблице, это снижение вполне терпимо и происходит так, что вряд ли повредит приложению, поскольку производительность все равно в М раз выше, чем при последовательном поиске.

Существует много других методов хеширования, которые находят применение в особых ситуациях. Мы не можем останавливаться на этом подробно, но все же кратко рассмотрим три примера, иллюстрирующие сущность специальных методов хеширования.

Один класс методов перемещает элементы во время вставки при двойном хешировании, делая успешный поиск более эффективным. Так, Брент (Brent) разработал метод, при использовании которого даже в заполненной таблице среднее время успешного поиска ограничено константой (см. раздел ссылок). Такой метод может быть удобным в приложениях, в которых основной операцией является успешный поиск.

Другой метод, называемый упорядоченным хешированием (ordered hashing), использует упорядочение для уменьшения затрат на неудачный поиск при использовании линейного опробования, приближая их к затратам на успешный поиск. В стандартном линейном опробовании поиск прекращается при обнаружении пустой позиции таблицы или элемента, ключ которого равен искомому; в упорядоченном хешировании поиск прекращается при обнаружении ключа, который больше или равен искомому ключу (чтобы эта процедура работала, таблица должна быть построена специальным образом) (см. раздел ссылок). Это усовершенствование путем ввода упорядочения в таблицу аналогично эффекту от упорядочения цепочек переполнения. Данный метод предназначен для приложений, в которых преобладают неудачные поиски.

Таблица символов с быстрым неудачным поиском и несколько более медленным успешным поиском может использоваться для реализации словаря исключений (exception dictionary). Например, система текстовой обработки может содержать алгоритм для разбиения слов на слоги, который успешно работает для большинства слов, но не работает в отдельных случаях (вроде слова " безыскусный " ). Скорее всего, в словаре исключений будет найдено лишь небольшое количество слов очень большого документа, поэтому почти все поиски будут неудачными.

Это лишь некоторые примеры из множества алгоритмических усовершенствований, предложенных для хеширования. Многие из них представляют интерес и находят важные применения. Однако, как обычно, следует избегать неоправданного применения сложных методов, если только это не обусловлено серьезными причинами, а компромисс между производительностью и сложностью не проанализирован самым тщательным образом, поскольку цепочки переполнения, линейное опробование и двойное хеширование просты, эффективны и приемлемы для большинства приложений.

Во многих приложениях хеширование предпочтительнее структур бинарных деревьев, для реализации таблиц символов, поскольку оно несколько проще и может обеспечить оптимальное (постоянное) время поиска, если ключи относятся к стандартному типу или достаточно просты, чтобы можно было уверенно разработать для них хорошую хеш-функцию. Преимущества структур бинарных деревьев по сравнению с хешированием заключается в том, что деревья основываются на более простом абстрактном интерфейсе (не требуется разработка хеш-функции); деревья являются динамическими структурами (не требуется никакая предварительная информация о количестве вставок); деревья могут обеспечить гарантированную производительность в худшем случае (даже наилучшая хеш-функция может отобразить все элементы в одну и ту же позицию); и, наконец, деревья поддерживают более широкий диапазон операций (самое главное - операции сортировать и выбрать). Если эти факторы не важны, безусловно, следует выбирать хеширование, но с одной важной оговоркой: если ключи являются длинными строками, их можно встроить в структуры данных, которые обеспечивают методы поиска, работающие еще быстрее хеширования.