Тема 4. Методы поиска

4.4. Хеширование

Методы *хеширования*, известные также как методы *рассеянной памяти*, методы *вычисления адреса*, основаны на том, что поиск ключа z в таблице $T = \{x_1, x_2, \ldots, x_n\}$ начинается с вычисления адреса по имени z. Суть хеширования заключается в следующем. Предполагается, что имеется память (*хеш-таблица*), состоящая из m ячеек M_j, $0 \le j \le m - 1$, при этом каждая ячейка может содержать одно имя. При заданном значении j обращение к ячейке M_j производится за постоянное время, не зависящее от размера m хеш-таблицы. Имеется функция $h : S \to A$, называемая *хеш-функцией*, которая равномерно отображает пространство имен S в пространство *хеш-адресов* $A = \{0, 1, \ldots, m - 1\}$, т. е. для любого имени $x \in S$ хеш-функция $h(x)$ принимает целочисленное значение из интервала $0, \ldots, m - 1$, которое является адресом ячейки хеш-таблицы. Число $h(x)$ называется *хеш-значением* (или *собственным адресом*) имени x. При анализе алгоритмов предполагается, что функция h может быть вычислена за время, не зависящее от мощности пространства имен, от размера m хеш-таблицы и от размера n исследуемой таблицы T.

Основным достоинством методов хеширования является то, что соответствующее им среднее время поиска не зависит от размера таблицы, если вся таблица умещается в оперативную память, т. е. асимптотически и часто также на практике они являются самыми быстрыми методами поиска.

В качестве недостатков методов хеширования можно отметить следующие:

а) табличный порядок имен обычно не связан с их естественным порядком;

б) худший случай может оказаться хуже, чем при последовательном поиске;

в) сложность динамического расширения таблиц, поскольку расширение может приводить к потере памяти, если таблица слишком велика, или к малой производительности, если таблица слишком мала.

4.4.1. Варианты хеширования

Идеи хеширования проще рассмотреть для различных вариантов таблиц и размеров пространства имен.

Вариант 1. Особенностью данного варианта является то, что пространство имен очень мало. Рассмотрим, например, пространство имен $S = \{A, B, \ldots, Z\}$ (односимвольные имена в латинском алфавите), хеш-таблицу $M = \{M_0, M_1, \ldots, M_{25}\}$ из 26 ячеек с пространством адресов $A = \{0, 1, \ldots, 25\}$ и хеш-функцию $h : S \to A$, которая отображает символы в адреса так, чтобы сохранялся естественный порядок имен в пространстве S, т. е. $h(A) = 0$, $h(B) = 1$, …, $h(Z) = 25$. Таблица $T = \{x_1, x_2, \ldots, x_n\} \subseteq S$ хранится в памяти следующим образом: в ячейках $M_{h(x_i)}$ размещаются имена x_i, а в ячейках, не содержащих имена (пустых ячейках), – специальный символ, не принадлежащий S (например, "–"):

$$S = \{A, B, \ldots, Z\} \qquad T = \{B, C, E, Z\}$$

A	0	1	2	3	4	⋯	24	25
M	–	B	C	–	E	⋯	–	Z

Четыре табличные операции выполняются простыми и быстрыми алгоритмами:

поиск z: **if** $M_{h(z)} = z$
then return $(h(z))$ // найдено: $h(z)$ указывает на z
else return (-1) // не найдено

включение z: $M_{h(z)} \leftarrow z$

исключение z: $M_{h(z)} \leftarrow$ "–"

распечатка: **for** $i \leftarrow 0$ **to** $m-1$ **do if** $M_i \neq$ "–" **then** write(M_i)

Необходимо обратить внимание на то, что в алгоритме поиска отсутствует явный цикл. Важно, чтобы любой цикл, присутствующий в вычислении хеш-функции h, не зависел от размера таблицы.

Рассмотренный вариант является идеальным случаем хеширования, когда размер хеш-таблицы оказался равным размеру пространства имен. На практике это выполняется редко, обычно память слишком мала, чтобы вместить все пространство имен.

Вариант 2. Особенностью данного варианта является то, что пространство имен велико (не умещается в память); таблица статическая. Поскольку таблица статическая, хеш-функцию можно выбрать после того, как станет известно содержимое таблицы. В этом случае можно найти легко вычислимую функцию, которая взаимно однозначно отображает множество хранящихся в таблице имен в пространство адресов (даже с сохранением естественного порядка имен в таблице). Особенно легко найти такую функцию, если хеш-таблица содержит больше ячеек, чем имен.

Для примера рассмотрим размещение четырех слов ВЫ, МЫ, ОН и ТЫ в хеш-таблице из пяти ячеек $M_0, M_1, \ldots, M_4$. Пусть каждая буква представлена 5-разрядным кодом: А соответствует 00000, Б – 00001, В – 00010 и т. д. Тогда указанные слова имеют следующее представление:

	b_9	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
ВЫ	0	0	0	1	0	1	1	0	1	1
МЫ	0	1	1	0	0	1	1	0	1	1
ОН	0	1	1	1	0	0	1	1	0	1
ТЫ	1	0	0	1	0	1	1	0	1	1

Достаточно легко заметить, что двоичная цепочка $b_9b_8b_2$ однозначно определяет каждое из четырех слов и что значения этих двоичных цепочек, интерпретируемых как двоичные целые числа, лежат в пространстве адресов $A = \{0, 1, 2, 3, 4\}$. Таким образом, хеш-функция $h(x) = (b_9b_8b_2)_2$ однозначно отображает множество четырех имен в пространство адресов хеш-таблицы, кроме того, сохраняется естественный алфавитный порядок.

Поскольку таблица статическая, т. е. для нее не определены операции включения и исключения, алгоритмы поиска и распечатки полностью совпадают с соответствующими алгоритмами из первого варианта хеширования.

Таким образом, данный вариант сводится к первому варианту, алгоритмы поиска и распечатки полностью совпадают с соответствующими алгоритмами из первого варианта хеширования.

Вариант 3. Особенностью данного варианта является то, что пространство имен велико (не умещается в память); таблица динамическая. В этом случае содержимое таблицы заранее не известно. Поэтому хеш-функция должна быть построена независимо от этого на основании некоторой другой информации о таблице. Например, может быть известно, что число имен, которые необходимо хранить в таблице, не превышает некоторой границы; что некоторые имена встречаются с большей вероятностью, чем другие и т. п. На основании такой информации можно выбрать размер хеш-таблицы m и построить хеш-функцию h. Окончательным критерием выбора m и h является их эффективность на практике.

Для примера рассмотрим пространство имен S всех двоичных цепочек фиксированной длины 8, при этом предположим, что известен максимальный размер таблицы – шесть имен. Допустим, что выбрана хеш-таблица из восьми ячеек $M_0, M_1, \ldots, M_7$ и простая хеш-функция, которая отображает цепочку из трех самых правых разрядов, интерпретируемых как двоичное целое. Эта функция легко вычисляется и рассеивает пространство имен равномерно по пространству адресов.

Пусть таблица на некотором этапе (после выполнения ряда операций включения и исключения) состоит из пяти имен

$$\begin{aligned} x_1 &= 00001001 \qquad h(x_1) = (001)_2 = 1, \\ x_2 &= 01000011 \qquad h(x_2) = (011)_2 = 3, \\ x_3 &= 01100110 \qquad h(x_3) = (110)_2 = 6, \\ x_4 &= 10110000 \qquad h(x_4) = (000)_2 = 0, \\ x_5 &= 11101011 \qquad h(x_5) = (011)_2 = 3. \end{aligned}$$

Нетрудно заметить, что h отображает x_2 и x_5 на один и тот же адрес. Такая ситуация называется *коллизией*, или *конфликтом*. Какое-то из этих имен должно храниться в другой ячейке, адрес которой не совпадает с собственным адресом имени, определяемым функцией h. Выбор другой ячейки, если ячейка с собственным адресом занята, должен осуществляться на основании определенных правил, которые устанавливаются выбранным *методом разрешения коллизий*. В рассматриваемом примере для разрешения коллизии определим следующее правило: если ячейка с собственным адресом имени занята, то имя помещается в первую свободную ячейку, следующую за его собственным адресом. Таким образом, имя x_5 с собственным адресом $h(x_5) = 3$ в соответствии с выбранным методом разрешения коллизий помещается в ячейку M_4 (рис. 4.16)

j	0	1	2	3	4	5	6	7
M_j	x_4	x_1	–	x_2	x_5	–	x_3	–

Рис. 4.1. Размещение имени x_5 при коллизии с именем x_2

Поскольку возникает задача распознавания состояния ячейки (занята или свободна), свободные (пустые) ячейки должны отличаться от занятых. Это можно решить либо добавлением дополнительного разряда к каждой ячейке, либо, если это возможно, использованием специального значения, не принадлежащего пространству имен, для обозначения пустой ячейки.

При возникновении коллизий поиск становится более сложным и обычно включает цикл, который может выполняться до m раз в худшем случае. При выбранном выше способе разрешения коллизий поиск требует последовательного просмотра памяти, при этом просмотр должен начинаться с собственного адреса искомого имени z и переходить от ячейки M_{m-1} к ячейке M_0 циклически. Поиск завершается безуспешно, если встречается пустая ячейка или когда просмотрена вся память. Чтобы избежать явной проверки последнего случая, необходимо в памяти всегда иметь по крайней мере одну пустую ячейку. Тогда операцию поиска ключа z в хеш-таблице можно представить алгоритмом 4.9.

$i \leftarrow h(z)$
while M_i не пуста **do**
  if $z = M_i$
    then **return**(i) // найдено: i указывает на z
    else $i \leftarrow (i+1)$ **mod** m
return(-1) // не найдено

Алгоритм 4.9. Поиск z в хеш-таблице

Включение имени z в хеш-таблицу представлено алгоритмом 4.10. Во время включения должна выполняться явная проверка того, что в хеш-таблице существует более одной пустой ячейки. В алгоритме это осуществляется сравнением числа n уже входящих в таблицу имен с размером хеш-таблицы m. Если выбранный размер памяти m (размер хеш-таблицы) по каким-либо причинам не удовлетворяет, то расширить таблицу для методов хеширования не так просто, как в других методах поиска, поскольку хеш-функция явно зависит от размера хеш-таблицы. Расширение памяти требует нового хеширования, т. е. отыскания новой хеш-функции и перемещения всех ранее размещенных имен в соответствии с новой хеш-функцией.

$i \leftarrow h(z)$
while M_i не пуста **do**
  if $z = M_i$
    then // z уже в таблице
    else $i \leftarrow (i+1)$ **mod** m
// z нет в таблице; включить его, если это возможно
if $n = m-1$
  then // таблица заполнена; z включать нельзя
  else
- // таблица не заполнена; включить z
- $M_i \leftarrow z$
- пометить M_i "занято"
- $n \leftarrow n+1$

Алгоритм 4.10. Включение z в хеш-таблицу

Очевидный способ исключения имени из ячейки M_i, когда M_i помечается как пустая, является неправильным для большинства схем хеширования. Например, пусть исключается имя x_2 и ячейка M_3 помечается как пустая (см. рис. 4.16). Тогда, если необходимо найти x_5, то последовательный поиск, начинающийся с собственного адреса имени $h(x_5) = 3$, находит, что M_3 пуста, и ошибочно заключает, что имени x_5 в таблице нет, хотя оно хранилось в ячейке M_4, из-за возникновения коллизии с x_2. Пока ячейка M_3 была занята, имя x_5 было достижимо, но когда M_3 стала пустой, x_5 стало недостижимым для алгоритма поиска. Если переместить x_5 из M_4 в его собственный адрес M_3, то это не дает гарантии, что пустая ячейка M_4 не сделает недоступным еще какое-либо имя. Просмотр всей последовательности имен (до первой пустой ячейки), анализ их собственных адресов и, если необходимо, перемещение требуют широкого поиска и проверки с существенным снижением эффективности работы. Поэтому исключение должно осуществляться по-другому.

Для реализации исключения необходимо иметь три состояния ячейки памяти: «занято», «пусто» и «исключено». Тогда исключение имени выполняется пометкой его ячейки как «исключено». Алгоритм поиска (алгоритм 4.9) для правильной его работы необходимо модифицировать так, чтобы ячейки с пометкой «исключено» игнорировались во время поиска. Алгоритм включения (алгоритм 4.10) нужно модифицировать так, чтобы новое имя включалось в пустую или «исключенную» ячейку в зависимости от того, какая из них встретится первой; в любом случае использованная ячейка помечается как занятая. Другими словами, «исключенная» ячейка ведет себя как пустая относительно включения и как заполненная относительно поиска. Данный подход при всей его привлекательности и гибкости имеет существенный недостаток. Поскольку «исключенная» ячейка ведет себя как заполненная ячейка относительно поиска, то это может привести к тому, что безуспешный поиск, который завершается, когда встречается первая пустая ячейка, становится очень неэффективным, если последовательность включений и исключений оставляет все меньше и меньше ячеек с пометкой «пусто», даже если общее число имен в таблице остается постоянным. В худшем случае при безуспешном поиске будет просматриваться вся хеш-таблица.

4.4.2. Хеш-функции

В идеальном варианте хеш-функция $h : S \to A$ должна быть легко вычислимой и отображать пространство имен в пространство адресов так, чтобы равномерно рассеять имена по памяти. Если второе требование предполагает, что каждый адрес $i \in A$ является образом примерно одинакового числа имен $x \in S$, то приемлемы простые хеш-функции, например функции, в которых подцепочки имени интерпретируются как двоичные целые. Однако на практике, поскольку вероятность появления различных имен из S в таблице различна, лучше позаботиться о том, чтобы равномерно рассеивалось по памяти содержимое таблицы, а не все пространство имен. Поэтому хеш-функция должна быть построена так, чтобы равномерно рассеять по памяти те подмножества множества S, которые могут встретиться в качестве содержимого таблицы. К сожалению, эти вероятные подмножества редко можно охарактеризовать точно. Следовательно, построение хеш-функций больше опирается на здравый смысл и интуицию, чем на какие-то формализованные методы.

Следует избегать хеш-функций, которые плохо работают на некоторых распространенных множествах имен. В частности, необходимо избегать хеш-функций, которые склонны отображать скученные имена (например, $x1$, $x2$, $x3$ или $a1$, $b1$, $c1$ и т. п.) в скученные адреса, поскольку они могут вызвать чрезмерное число коллизий.

Для обеспечения быстрого вычисления хеш-адресов большинство хеш-функций близко к примитивным операциям, допустимым для ЭВМ. Поэтому они лучше описываются на уровне операций над двоичными наборами в представлениях имен и адресов. В связи с этим будем полагать, что имена и адреса представлены двоичными наборами длины l_{name} и l_{addr} соответственно. Часто l_{name} меньше или равно длине машинного слова, и во всех представляющих интерес случаях $l_{\text{name}} > l_{\text{addr}}$, поскольку в случае $l_{\text{name}} \le l_{\text{addr}}$ каждому имени можно сопоставить свой адрес.

Простейшие хеш-функции выбирают некоторое специальное подмножество из l_{addr} разрядов из строки с l_{name} разрядами и используют их для формирования адреса. Существенным недостатком таких хеш-функций является то, что они имеют тенденцию порождать чрезмерное количество коллизий, так как все имена, представления которых совпадают по подмножеству разрядов, выбранному хеш-функцией, отображаются в один адрес.

Большинство используемых на практике хеш-функций основано на другом подходе, связанном с требованием, чтобы каждый из l_{addr} разрядов адреса зависел от всех l_{name} разрядов имени. Наиболее эффективный путь достижения этой зависимости состоит в том, чтобы производить арифметические операции, результаты которых зависят от всех разрядов имени, и затем выделить l_{addr} разрядов из этого результата. В качестве примера можно привести некоторые методы построения хеш-функций.

Метод середины квадрата. Метод состоит в том, что имя представляется целым числом, затем находится его квадрат, после чего из середины представления квадрата выделяются l_{addr} разрядов. Хотя этот метод часто является удовлетворительным, он обладает существенным недостатком, связанным с тем, что возведение в квадрат не сохраняет равномерности распределения. Особенно неприятно проявляется такое нарушение распределений, когда представление имени оканчивается нулями. Его квадрат будет иметь в конце вдвое больше нулей, и эти нули могут распространяться на среднюю область представления квадрата, искажая получающийся в результате адрес. Поскольку задачей хеширования является равномерное распределение имен в хеш-таблице, в общем случае от метода середины квадрата отказываются.

Метод мультипликативного хеширования. Метод свободен от недостатков, присущих методу квадрата, поскольку его основой является умножение на константу, что сохраняет равномерность распределения имен в памяти. При выборе константы следует помнить, что арифметические действия выполняются в некоторой дискретной конечной системе счисления, свойства которой (например, основание системы счисления и длину слова) необходимо учитывать, чтобы избежать вырожденных случаев, которые приводят к большому числу коллизий. Основная опасность мультипликативного хеширования относится к таблицам, которые содержат подмножества имен с представлениями, образующими арифметические прогрессии. Мультипликативные хеш-функции отображают такие прогрессии имен в арифметические прогрессии адресов, например, с разностью i. Если константа или размер памяти m выбраны произвольно, то i может оказаться делителем m и эти подмножества будут использовать только $1/i$ часть объема памяти. Следствием этого является порождение чрезмерного количества коллизий.

Метод деления с остатком. Этот метод использует хеш-функцию типа

$$h(x) = (\text{представление } x, \text{ рассматриваемое как целое}) \ \mathbf{mod}\ m,$$

где m – размер хеш-таблицы, **mod** – операция получения остатка от деления целых чисел. Здесь решающим является выбор величины m. Например, если m четно, то $h(x)$ четно тогда и только тогда, когда представление x будет четным целым. Эта неравномерность может оказать существенное влияние на эффективность хеширования. Если m – степень основания системы счисления, применяемой в ЭВМ, то $h(x)$ будет зависеть только от самых правых символов имени x, что является еще одной опасностью неравномерности. Вообще к проблемам может приводить любой собственный делитель m. Поэтому рекомендуется, чтобы m было простым, далеко отстоящим от степеней числа 2, что гарантирует еще и то, что каждая цифра адреса зависит от всех символов имени.

Рассмотренных методов построения хеш-функций достаточно, чтобы обратить внимание на то, что любая разумная хеш-функция будет удовлетворительно работать на большинстве множеств имен, но в то же время для любой хеш-функции существуют множества имен, на которых она работает плохо. Это связано с тем, что хеш-функция делит пространство имен на группы имен с одинаковыми хеш-значениями. Если k – мощность пространства имен (в общем случае пространство имен может быть бесконечным), m – размер хеш-таблицы, то среднее число имен в таких группах будет k/m. Поэтому вполне возможна ситуация, когда большинство имен в таблице будет иметь один и тот же хеш-адрес (такая ситуация называется *первичным скучиванием*) и, как следствие, большое число коллизий.

4.4.3. Разрешение коллизий

Пока хеш-таблица не слишком заполнена, коллизии появляются редко, и производительность схемы хеширования определяется прежде всего временем, требующимся для вычисления хеш-функции. По мере заполнения хеш-таблицы доступ к именам требует все большего времени из-за коллизий. Поэтому когда хеш-таблица используется в большой степени, эффективность схемы хеширования определяется схемой разрешения коллизий, и выбор схемы разрешения коллизий обычно важнее выбора хеш-функции.

Схема разрешения коллизий каждому имени x сопоставляет последовательность адресов $\alpha_0, \alpha_1, \alpha_2, \ldots$, где $\alpha_0 = h(x)$ является собственным адресом имени x. Алгоритм включения проверяет ячейки до тех пор, пока не найдет пустую. Для гарантии того, что пустая ячейка встречается, если она существует, каждый адрес i, $0 \le i \le m - 1$, должен появляться в последовательности точно один раз. По существу, эту последовательность можно представить двумя способами, соответствующими последовательному и связному распределению списков.

Открытая адресация. В рассмотренном третьем варианте хеширования использована простейшая форма открытой адресации, называемая *линейным опробованием*, которое порождает последовательность адресов $\alpha_i = (h(x) + i)$ **mod** m. Простота полученной таким путем последовательности не компенсирует существенный недостаток, который делает ее обычно неудовлетворительной. Этим недостатком является *вторичное скучивание*. Оно возникает, когда имена с различными хеш-значениями имеют одинаковые (или почти одинаковые) последовательности адресов $\alpha_1, \alpha_2, \ldots$. Когда возникает первичное скучивание, линейное опробование порождает последовательность занятых следующих одна за другой ячеек и все имена, попадающие в любые из этих ячеек в результате хеширования, порождают последовательность адресов, которая пробегает эти занятые ячейки.

Вторичного скучивания можно избежать, используя линейное опробование с приращением $\Delta(x)$, которое является функцией от x. Это приращение позволяет получить последовательность адресов $\alpha_i = (h(x) + i\,\Delta(x))$ **mod** m, которая будет опробовать каждую ячейку, если $\Delta(x)$ и m взаимно просты. Поскольку $\Delta(x)$, по существу, является другой хеш-функцией, этот метод называется *двойным хешированием*. Двойное хеширование является наилучшим методом разрешения коллизий для открытой адресации.

Метод цепочек. Метод цепочек – это способ построения последовательности указателей из собственного адреса $h(x)$ в ячейку, где x размещается окончательно. С помощью дополнительного расхода памяти для хранения указателей такой способ позволяет избежать проблемы вторичного скучивания во время поиска, но не во время включения. Для иллюстрации рассмотрим пример, представленный на рис. 4.17.

Пусть имена x и y образуют коллизию: x размещается по своему собственному адресу $\alpha_0 = h(x)$ и y размещается по следующему адресу α_1 в последовательности разрешения коллизий для α_0 (рис. 4.17, *а*). Предположим, что имя z нельзя записать по его собственному адресу $h(z) = \beta_0$ и что, следуя по его последовательности разрешения коллизий $\beta_1, \beta_2, \ldots$, путь проходит через α_0 и α_1: например, $\beta_1 = \alpha_0$ и $\beta_2 = \alpha_1$. Для отыскания пустой ячейки во время включения просматриваются α_0 и α_1 (на рис. 4.17, *б* последовательность просмотра ячеек показана штриховыми стрелками), но как только z разместится в β_3, ячейка β_3 привязывается непосредственно к $\beta_0 = h(z)$ (рис. 4.17, *б*). Поэтому при поиске z производится меньше проб, чем при его включении.

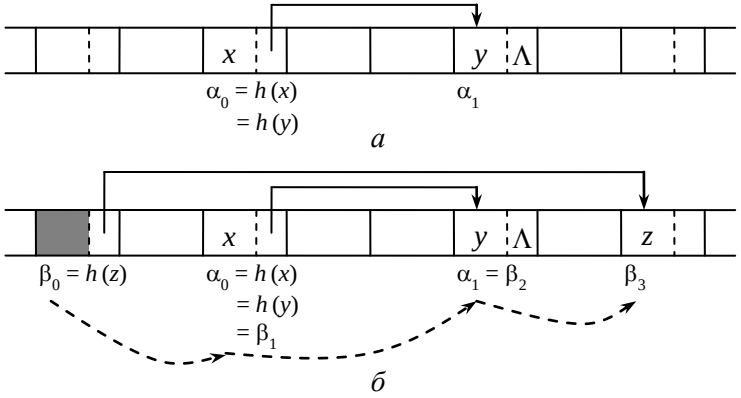

Рис. 4.17. Разрешение коллизий методом цепочек:
а – размещение имен x и y; *б* – добавление имени z

Рассмотренные ранее методы разрешения коллизий используют ячейки только из хеш-таблицы. Метод хеширования, в котором для хранения имен используется ограниченное пространство хеш-таблицы, называется *закрытым*, или *прямым, хешированием*. Если такие требования не установлены, образующие коллизии имена можно помещать в отдельной дополнительной области памяти. Такое хеширование называется *открытым*, или *внешним*, и эквивалентно комбинации разных методов поиска: хеширование используется на первом шаге, а другие методы используются для поиска среди имен с одинаковыми хеш-значениями. На рис. 4.18 показана простейшая организация данных для открытого хеширования, когда в хеш-таблице хранятся не сами имена, а указатели на связные списки имен, образующих коллизии. Для повышения эффективности поиска множество имен с одинаковыми хеш-значениями можно упорядочить и использовать для его представления более гибкие и эффективные структуры данных.

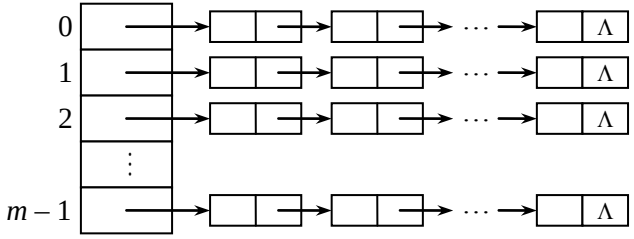

Рис. 4.18. Простейшая организация открытого хеширования

Что касается ожидаемой эффективности методов хеширования, то можно отметить следующее. Если предположить, что время доступа к любой ячейке памяти постоянно и что хеш-функция равномерно отображает пространство имен в пространство адресов, а также что содержимое таблицы есть беспристрастная выборка из пространства имен, ожидаемая эффективность метода вычисления адреса не зависит от числа имен n в таблице. Она зависит в основном от *коэффициента заполнения* хеш-таблицы $\lambda = n/m$. В экстремальном случае, когда память используется полностью ($\lambda = 1$), очевидно, что быстрого времени доступа достигнуть невозможно. Поэтому важно рассмотреть эффективность метода, когда хеш-таблица почти заполнена.

Для иллюстрации можно привести результаты приближенного анализа двойного хеширования. Математическое ожидание $U(\lambda)$ числа проб, необходимых для безуспешного поиска ключа z в хеш-таблице с коэффициентом заполнения λ, можно определить как $U(\lambda) = 1/(1 - \lambda)$, а математическое ожидание $S(\lambda)$ числа проб, необходимых для успешного поиска, как

$$S(\lambda) = \frac{1}{\lambda} \ln \frac{1}{1-\lambda}.$$

Функции $U(\lambda)$ и $S(\lambda)$ стремятся к 1 при $\lambda \to 0$, указывая, что при поиске в почти пустой таблице достаточно одной пробы. Функции $U(\lambda)$ и $S(\lambda)$ неограниченно растут при $\lambda \to 1$, что является следствием интерпретации λ как непрерывной величины, что равнозначно предположению о бесконечной памяти. Следующая таблица показывает значения функций $U(\lambda)$ и $S(\lambda)$ для различных значений коэффициента заполнения λ.

λ	0,5	0,75	0,9	0,95	0,99
$U(\lambda)$	2,0	4,0	10,0	20,0	100,0
$S(\lambda)$	1,39	1,85	2,56	3,15	4,65

Таким образом, если адреса, порожденные методом разрешения коллизий, независимы и равномерно распределены на пространстве адресов, как обычно бывает при двойном хешировании, то среднее число требуемых проб мало даже для такого большого коэффициента заполнения, как 90 %.