Глава 11. Хеширование и хеш-таблицы

Для многих приложений достаточно динамических множеств, поддерживающих только стандартные словарные операции INSERT, SEARCH и DELETE. Например, компилятор, транслирующий язык программирования, поддерживает таблицу символов, в которой ключами элементов являются произвольные символьные строки, соответствующие идентификаторам в языке. Хеш-таблица представляет собой эффективную структуру данных для реализации словарей. Хотя на поиск элемента в хеш-таблице может в наихудшем случае потребоваться столько же времени, сколько на поиск в связанном списке, а именно — $\Theta(n)$, на практике хеширование исключительно эффективно. При вполне обоснованных допущениях среднее время поиска элемента в хеш-таблице составляет $O(1)$.

Хеш-таблица обобщает обычный массив. Возможность прямой индексации элементов обычного массива обеспечивает доступ к произвольной позиции в массиве за время $O(1)$. Более подробно прямая индексация рассматривается в разделе 11.1; она применима, если мы в состоянии выделить массив такого размера, какого достаточно для того, чтобы для каждого возможного значения ключа имелась своя ячейка.

Если количество реально хранящихся в массиве ключей мало́ по сравнению с количеством возможных значений ключей, эффективной альтернативой массива с прямой индексацией становится хеш-таблица, которая обычно использует массив, размер которого пропорционален количеству реально хранящихся в нем ключей. Вместо непосредственного использования ключа в качестве индекса массива индекс *вычисляется* по значению ключа. В разделе 11.2 представлены основные идеи хеширования, в первую очередь направленные на разрешение коллизий (когда несколько ключей отображается в один и тот же индекс массива) с помощью цепочек. В разделе 11.3 описывается, каким образом на основе значений ключей могут быть вычислены индексы массива. Здесь будет рассмотрено и проанализировано несколько вариантов хеш-функций. В разделе 11.4 вы познакомитесь с методом открытой адресации, представляющим собой еще один способ разрешения коллизий. Главный вывод, который следует из всего изложенного материала, — хеширование представляет собой исключительно эффективную и практичную технологию: в среднем все базовые словарные операции выполняются за время $O(1)$. В разделе 11.5 будет дано пояснение, каким образом "идеальное хеширование" может поддерживать *наихудшее* время поиска $O(1)$ в случае ис-

пользования статического множества хранящихся ключей (т.е. когда множество ключей, будучи сохраненным, более не изменяется).

11.1. Таблицы с прямой адресацией

Прямая адресация представляет собой простую технологию, которая хорошо работает для небольших совокупностей ключей. Предположим, что приложению требуется динамическое множество, каждый элемент которого имеет ключ из совокупности $U = \{0, 1, \ldots, m-1\}$, где m не слишком велико. Кроме того, предполагается, что никакие два элемента не имеют одинаковых ключей.

Для представления динамического множества мы используем массив, или ***таблицу с прямой адресацией***, который обозначим как $T[0..m-1]$, каждая позиция (position), или ячейка, слот (slot) которого соответствует ключу из совокупности ключей U. На рис. 11.1 проиллюстрирован данный подход. Ячейка k указывает на элемент множества с ключом k. Если множество не содержит элемента с ключом k, то $T[k] = \text{NIL}$.

Реализация словарных операций тривиальна.

DIRECT-ADDRESS-SEARCH(T, k)

1 **return** $T[k]$

DIRECT-ADDRESS-INSERT(T, x)

1 $T[x.key] = x$

DIRECT-ADDRESS-DELETE(T, x)

1 $T[x.key] = \text{NIL}$

Каждая из этих операций выполняется за время $O(1)$.

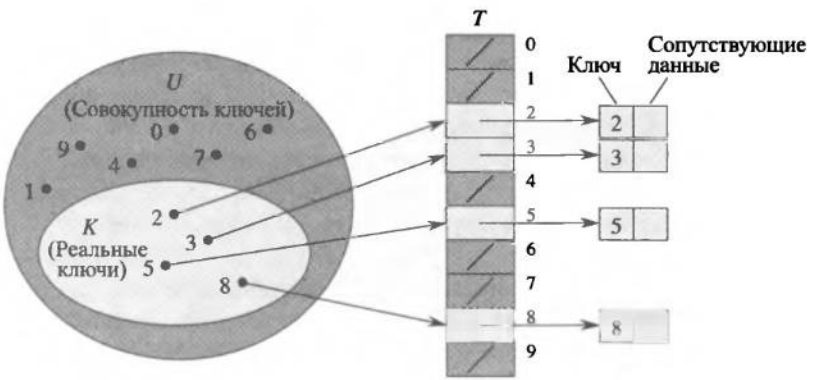

Рис. 11.1. Реализация динамического множества с использованием таблицы с прямой адресацией T. Каждый ключ в совокупности $U = \{0, 1, \ldots, 9\}$ соответствует индексу в таблице. Множество $K = \{2, 3, 5, 8\}$ реальных ключей определяет ячейки в таблице, которые содержат указатели на элементы. Прочие (закрашенные темным цветом) ячейки содержат значение NIL.

В некоторых приложениях элементы динамического множества могут храниться непосредственно в таблице с прямой адресацией. Иначе говоря, вместо хранения ключей и сопутствующих данных элементов в объектах, внешних по отношению к таблице с прямой адресацией, а в таблице — указателей на эти объекты, эти объекты можно хранить непосредственно в ячейках таблицы (что тем самым приводит к экономии используемой памяти). При этом для указания пустой ячейки можно воспользоваться специальным значением ключа. Кроме того, зачастую хранение ключа не является необходимым условием, поскольку если мы знаем индекс объекта в таблице, значит, мы знаем и его ключ. Однако если ключ не хранится в ячейке таблицы, то нам нужен какой-то иной механизм для того, чтобы помечать пустые ячейки.

Упражнения

11.1.1

Предположим, что динамическое множество S представлено таблицей с прямой адресацией T длиной m. Опишите процедуру, которая находит максимальный элемент S. Чему равно время работы этой процедуры в наихудшем случае?

11.1.2

Битовый вектор представляет собой массив битов (нулей и единиц). Битовый вектор длиной m занимает существенно меньше места, чем массив из m указателей. Каким образом можно использовать битовый вектор для представления динамического множества различных элементов без сопутствующих данных? Словарные операции должны выполняться за время $O(1)$.

11.1.3

Предложите способ реализации таблицы с прямой адресацией, в которой ключи хранящихся элементов могут совпадать, а сами элементы — иметь сопутствующие данные. Все словарные операции (INSERT, DELETE и SEARCH) должны выполняться за время $O(1)$. (Не забудьте, что аргументом процедуры DELETE является указатель на удаляемый объект, а не ключ.)

11.1.4 ⋆

Предположим, что мы хотим реализовать словарь с использованием прямой адресации *очень большого* массива. Первоначально в массиве может содержаться "мусор", но инициализация всего массива нерациональна в силу его размера. Разработайте схему реализации словаря с прямой адресацией при описанных условиях. Каждый хранимый объект должен использовать $O(1)$ памяти; операции SEARCH, INSERT и DELETE должны выполняться за время $O(1)$; инициализация структуры данных также должна выполняться за время $O(1)$. (*Указание:* для определения, является ли данная запись в большом массиве корректной, воспользуйтесь дополнительным массивом, работающим в качестве стека, размер которого равен количеству ключей, сохраненных в словаре.)

11.2. Хеш-таблицы

Недостаток прямой адресации очевиден: если совокупность ключей U велика, хранение таблицы T размером $|U|$ непрактично, а то и вовсе невозможно — в зависимости от количества доступной памяти и размера совокупности ключей. Кроме того, множество K *реально сохраненных* ключей может быть мало́ по сравнению с совокупностью ключей U, а в этом случае память, выделенная для таблицы T, в основном расходуется напрасно.

Когда множество K хранящихся в словаре ключей гораздо меньше совокупности возможных ключей U, для хеш-таблицы требуется существенно меньше места, чем для таблицы с прямой адресацией. Точнее говоря, требования к памяти могут быть снижены до $\Theta(|K|)$, при этом время поиска элемента в хеш-таблице остается равным $O(1)$. Нужно только заметить, что это граница времени поиска *в среднем случае*, в то время как в случае таблицы с прямой адресацией эта граница справедлива для *наихудшего случая*.

В случае прямой адресации элемент с ключом k хранится в ячейке k. При хешировании этот элемент хранится в ячейке $h(k)$, т.е. мы используем ***хеш-функцию*** h для вычисления ячейки для данного ключа k. Функция h отображает совокупность ключей U на ячейки ***хеш-таблицы*** $T[0\,..\,m-1]$:

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

где размер m хеш-таблицы обычно гораздо меньше значения $|U|$. Мы говорим, что элемент с ключом k ***хешируется*** в ячейку $h(k)$; величина $h(k)$ называется ***хеш-значением*** ключа k. На рис. 11.2 представлена основная идея хеширования. Цель хеш-функции состоит в том, чтобы уменьшить рабочий диапазон индексов массива, и вместо размера $|U|$ значений мы можем обойтись массивом всего лишь размером m.

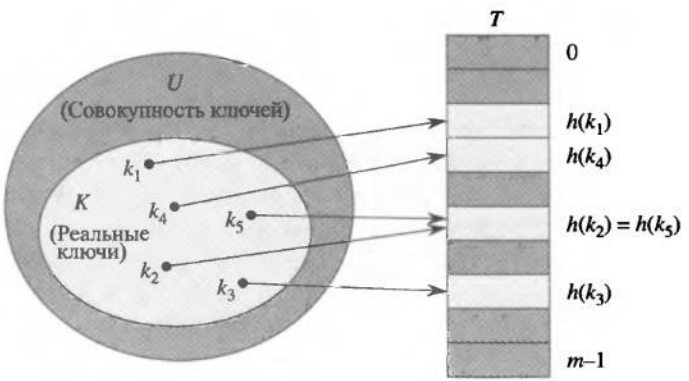

Рис. 11.2. Применение хеш-функции h для отображения ключей в ячейки хеш-таблицы. Ключи k_2 и k_5 отображаются в одну ячейку, вызывая коллизию.

Однако здесь есть одна проблема: два ключа могут быть хешированы в одну и ту же ячейку. Такая ситуация называется ***коллизией***. К счастью, имеются эффективные технологии разрешения конфликтов, вызываемых коллизиями.

Конечно, идеальным решением было бы полное устранение коллизий. Мы можем попытаться добиться этого путем выбора подходящей хеш-функции h. Одна из идей заключается в том, чтобы сделать функцию h "случайной", что позволило бы избежать коллизий или хотя бы минимизировать их количество (этот характер функции хеширования отображается в самом глаголе "to hash", который означает "мелко порубить, перемешать"). Само собой разумеется, функция h должна быть детерминистической и для одного и того же значения k всегда давать одно и то же хеш-значение $h(k)$. Однако поскольку $|U| > m$, должно существовать как минимум два ключа, которые имеют одинаковое хеш-значение. Таким образом, полностью избежать коллизий невозможно в принципе, и хорошая хеш-функция в состоянии только минимизировать их количество. Таким образом, нам все равно нужен метод разрешения возникающих коллизий.

В оставшейся части данного раздела мы рассмотрим простейший метод разрешения коллизий — метод цепочек. В разделе 11.4 вы познакомитесь с еще одним методом разрешения коллизий, который называется методом открытой адресации.

Разрешение коллизий с помощью цепочек

При разрешении коллизий ***с помощью цепочек*** мы помещаем все элементы, хешированные в одну и ту же ячейку, в связанный список, как показано на рис. 11.3. Ячейка j содержит указатель на заголовок списка всех элементов, хеш-значение ключа которых равно j; если таких элементов нет, ячейка содержит значение NIL.

Словарные операции в хеш-таблице с использованием цепочек для разрешения коллизий реализуются очень просто.

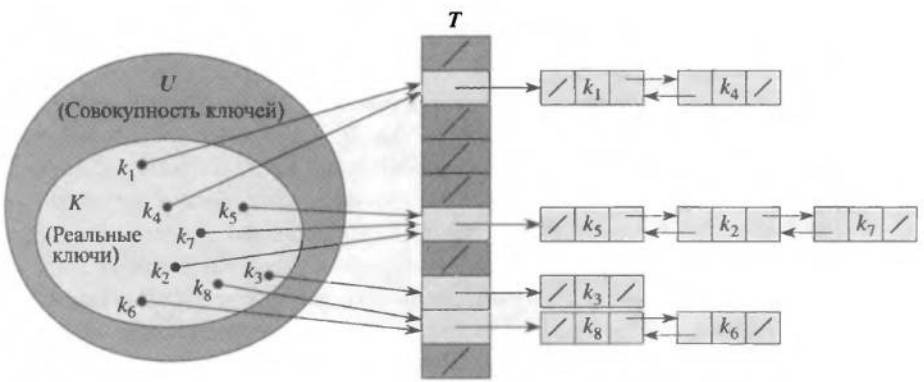

Рис. 11.3. Разрешение коллизий с помощью цепочек. Каждая ячейка хеш-таблицы $T[j]$ содержит связанный список всех ключей с хеш-значением j. Например, $h(k_1) = h(k_4)$ и $h(k_5) = h(k_7) = h(k_2)$. Связанный список может быть одинарно или дважды связанным; мы показываем его как дважды связанный, поскольку удаление в этом случае выполняется гораздо быстрее.

CHAINED-HASH-INSERT(T, x)

1 Вставка x в заголовок списка $T[h(x.key)]$

CHAINED-HASH-SEARCH(T, k)

1 Поиск элемента с ключом k в списке $T[h(k)]$

CHAINED-HASH-DELETE(T, x)

1 Удаление x из списка $T[h(x.key)]$

Время, необходимое для вставки в наихудшем случае, равно $O(1)$. Процедура вставки выполняется очень быстро, в частности, потому, что предполагается, что вставляемый элемент отсутствует в таблице. При необходимости это предположение может быть проверено дополнительной ценой выполнения поиска элемента с ключом $x.key$ перед вставкой. Время работы поиска в наихудшем случае пропорционально длине списка; мы проанализируем эту операцию немного позже. Удаление элемента может быть выполнено за время $O(1)$ при использовании дважды связанных списков, как на рис. 11.3. (Обратите внимание на то, что процедура CHAINED-HASH-DELETE принимает в качестве аргумента элемент x, а не его ключ, поэтому нет необходимости в предварительном поиске x. Если хеш-таблица поддерживает удаление, ее списки должны быть двусвязными для ускорения процесса удаления. Если список односвязный, то передача в качестве аргумента x не дает нам особого выигрыша, поскольку для корректного обновления атрибута $next$ предшественника x нам все равно нужно выполнить поиск x в списке $T[h(x.key)]$. В таком случае, как нетрудно понять, удаление и поиск имеют, по сути, одно и то же асимптотическое время работы.)

Анализ хеширования с цепочками

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

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

В наихудшем случае хеширование с цепочками ведет себя крайне неприятно: все n ключей хешированы в одну и ту же ячейку, создав список длиной n. Таким образом, время поиска в наихудшем случае равно $\Theta(n)$ плюс время вычисления хеш-функции, что ничуть не лучше, чем в случае использования связанного списка для хранения всех n элементов. Понятно, что использование хеш-таблиц в наихудшем случае совершенно бессмысленно. (Идеальное хеширование, применимое в случае статического множества ключей и рассмотренное в разделе 11.5. обеспечивает высокую производительность даже в наихудшем случае.)

Производительность хеширования в среднем случае зависит от того, насколько хорошо хеш-функция h распределяет множество сохраняемых ключей по m ячейкам в среднем. Мы рассмотрим этот вопрос подробнее в разделе 11.3, а пока

11.3. Хеш-функции

В этом разделе мы рассмотрим некоторые вопросы, связанные с разработкой качественных хеш-функций, и познакомимся с тремя схемами их построения. Две из них, хеширование делением и хеширование умножением, эвристичны по своей природе, в то время как третья схема — универсальное хеширование — использует рандомизацию для обеспечения доказуемо высокой производительности.

Чем определяется качество хеш-функции

Качественная хеш-функция удовлетворяет (приближенно) предположению простого равномерного хеширования: для каждого ключа равновероятно помещение в любую из m ячеек независимо от хеширования остальных ключей. К сожалению, это условие обычно невозможно проверить, поскольку, как правило, распределение вероятностей, в соответствии с которым поступают вносимые в таблицу ключи, неизвестно. Более того, вставляемые ключи могут не быть независимыми.

Иногда распределение вероятностей оказывается известным. Например, если известно, что ключи представляют собой случайные действительные числа, равномерно распределенные в диапазоне $0 \leq k < 1$, то хеш-функция

$$h(k) = \lfloor km \rfloor$$

удовлетворяет условию простого равномерного хеширования.

На практике при построении качественных хеш-функций зачастую используются различные эвристические методики. В процессе построения большую помощь оказывает информация о распределении ключей. Рассмотрим, например, таблицу символов компилятора, в которой ключами служат символьные строки, представляющие идентификаторы в программе. Зачастую в одной программе встречаются похожие идентификаторы, например `pt` и `pts`. Хорошая хеш-функция должна минимизировать шансы попадания этих идентификаторов в одну ячейку хеш-таблицы.

При построении хеш-функции хорошим подходом является подбор функции таким образом, чтобы она никак не коррелировала с закономерностями, которым могут подчиняться существующие данные. Например, метод деления, который рассматривается в разделе 11.3.1, вычисляет хеш-значение как остаток от деления ключа на некоторое простое число. Если это простое число никак не связано с распределением исходных данных, метод часто дает хорошие результаты.

В заключение заметим, что некоторые приложения хеш-функций могут накладывать более строгие требования по сравнению с требованиями простого равномерного хеширования. Например, мы можем потребовать, чтобы "близкие" в некотором смысле ключи давали далекие хеш-значения (это свойство особенно желательно при использовании линейного исследования, описанного в разделе 11.4). Универсальное хеширование, описанное в разделе 11.3.3, часто приводит к желаемым результатам.

Интерпретация ключей как целых неотрицательных чисел

Для большинства хеш-функций совокупность ключей представляется множеством целых неотрицательных чисел $\mathbb{N} = \{0, 1, 2, \ldots\}$. Если же ключи не являются целыми неотрицательными числами, то можно найти способ их интерпретации как таковых. Например, строка символов может рассматриваться как целое число, записанное в соответствующей системе счисления. Так, идентификатор `pt` можно рассматривать как пару десятичных чисел $(112, 116)$, поскольку в ASCII-наборе символов `p` $= 112$ и `t` $= 116$. Рассматривая `pt` как число в системе счисления с основанием 128, мы находим, что оно соответствует значению $(112 \cdot 128) + 116 = 14452$. В конкретных приложениях обычно не представляет особого труда разработать метод для представления ключей в виде (возможно, больших) целых чисел. Далее при изложении материала мы будем считать, что все ключи представляют собой целые неотрицательные числа.

11.3.1. Метод деления

Построение хеш-функции ***методом деления*** состоит в отображении ключа k в одну из m ячеек путем получения остатка от деления k на m, т.е. хеш-функция имеет вид

$$h(k) = k \bmod m \ .$$

Например, если хеш-таблица имеет размер $m = 12$, а значение ключа $k = 100$, то $h(k) = 4$. Поскольку для вычисления хеш-функции требуется только одна операция деления, хеширование методом деления достаточно быстрое.

При использовании данного метода мы обычно стараемся избегать некоторых значений m. Например, m не должно быть степенью 2, поскольку если $m = 2^p$, то $h(k)$ представляет собой просто p младших битов числа k. Если только заранее не известно, что все наборы младших p битов ключей равновероятны, лучше строить хеш-функцию таким образом, чтобы ее результат зависел от всех битов ключа. В упр. 11.3.3 требуется показать неудачность выбора $m = 2^p - 1$, когда ключи представляют собой строки символов, интерпретируемые как числа в системе счисления с основанием 2^p, поскольку перестановка символов ключа не приводит к изменению его хеш-значения.

Зачастую хорошие результаты можно получить, выбирая в качестве значения m простое число, достаточно далекое от степени двойки. Предположим, например, что мы хотим создать хеш-таблицу с разрешением коллизий методом цепочек для хранения порядка $n = 2000$ символьных строк, размер символов в которых равен 8 бит. Нас устраивает проверка в среднем трех элементов при неудачном поиске, так что мы выбираем размер таблицы равным $m = 701$. Число 701 выбрано как простое число, близкое к величине $2000/3$ и не являющееся степенью 2. Рассматривая каждый ключ k как целое число, мы получаем искомую хеш-функцию

$$h(k) = k \bmod 701 \ .$$

11.3.2. Метод умножения

Построение хеш-функции ***методом умножения*** выполняется в два этапа. Сначала мы умножаем ключ k на константу $0 < A < 1$ и выделяем дробную часть полученного произведения. Затем мы умножаем полученное значение на m и применяем к нему функцию "пол". Короче говоря, хеш-функция имеет вид

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

где выражение "$kA \bmod 1$" означает получение дробной части произведения kA, т.е. величину $kA - \lfloor kA \rfloor$.

Преимущество метода умножения заключается в том, что значение m перестает быть критичным. Обычно величина m из соображений удобства реализации функции выбирается равной степени 2 ($m = 2^p$ для некоторого натурального p), поскольку такая функция легко реализуема на большинстве компьютеров. Пусть у нас имеется компьютер с размером слова w бит и k помещается в одно слово. Ограничим возможные значения константы A дробями вида $s/2^w$, где s — целое число из диапазона $0 < s < 2^w$. Тогда мы сначала умножаем k на w-битовое целое число $s = A \cdot 2^w$. Результат представляет собой $2w$-битовое число $r_1 2^w + r_0$, где r_1 — старшее слово произведения, а r_0 — младшее. Старшие p бит числа r_0 представляют собой искомое p-битовое хеш-значение (рис. 11.4).

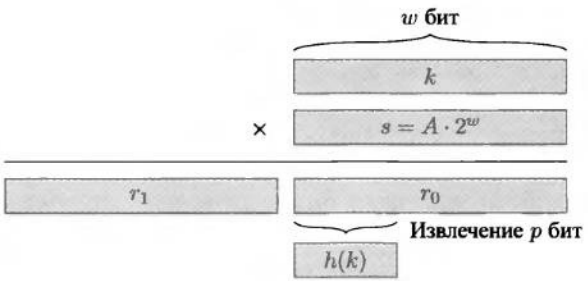

Рис. 11.4. Хеширование методом умножения. w-битовое представление ключа k умножается на w-битовое значение $s = A \cdot 2^w$. p старших битов младшей w-битовой половины произведения образует искомое хеш-значение $h(k)$

Хотя описанный метод работает с любыми значениями константы A, одни значения дают лучшие результаты по сравнению с другими. Оптимальный выбор зависит от характеристик хешируемых данных. В [210][1] Кнут (Knuth) предложил использовать дающее неплохие результаты значение

$$A \approx (\sqrt{5} - 1)/2 = 0.6180339887\ldots \qquad (11.2)$$

В качестве примера предположим, что $k = 123456$, $p = 14$, $m = 2^{14} = 16384$ и $w = 32$. Принимая предложения Кнута, выберем A в виде дроби $s/2^{32}$, бли-

[1] Имеется русский перевод: Д. Кнут. *Искусство программирования, т. 3. Сортировка и поиск, 2-е изд.* — М.: И.Д. "Вильямс", 2000.

жайшей к значению $(\sqrt{5}-1)/2$, так что $A = 2654435769/2^{32}$. Тогда $k \cdot s = 327706022297664 = (76300 \cdot 2^{32}) + 17612864$, и, таким образом, $r_1 = 76300$ и $r_0 = 17612864$. Старшие 14 бит числа r_0 дают хеш-значение $h(k) = 67$.

⋆ 11.3.3 Универсальное хеширование

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

Главная идея универсального хеширования состоит в случайном выборе хеш-функции из некоторого тщательно отобранного класса функций в начале работы программы. Как и в случае быстрой сортировки, рандомизация гарантирует, что одни и те же входные данные не могут постоянно давать наихудшее поведение алгоритма. В силу рандомизации алгоритм будет работать всякий раз по-разному, даже для одних и тех же входных данных, что гарантирует высокую среднюю производительность для любых входных данных. Возвращаясь к примеру с таблицей символов компилятора, мы обнаружим, что никакой выбор программистом имен идентификаторов не может привести к постоянно низкой производительности хеширования. Такое снижение возможно только тогда, когда компилятором выбрана случайная хеш-функция, которая приводит к плохому хешированию конкретных входных данных; однако вероятность такой ситуации очень мала и одинакова для любого множества идентификаторов одного и то же размера.

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

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

Теорема 11.3

Пусть хеш-функция h, случайным образом выбранная из универсального множества хеш-функций, применяется для хеширования n ключей в таблицу T размером m с использованием для разрешения коллизий метода цепочек. Если ключ k отсутствует в таблице, то математическое ожидание $\mathrm{E}\left[n_{h(k)}\right]$ длины списка, в ко-