БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

СБОРНИК ЗАДАЧ

ПО ТЕОРИИ АЛГОРИТМОВ.

Структуры данных.
Часть. Организация поиска. Хеширование.

МИНСК
2020

УДК 510.51(075.8)
ББК 22.12я73-1
С23

А в т о р ы :
С. А. Соболь, К. Ю. Вильчевский, В. М. Котов,
Е. П. Соболевская

Р е ц е н з е н т ы :
кафедра информатики и методики преподавания информатики
физико-математического факультета Белорусского государственного
педагогического университета им. М. Танка
(заведующий кафедрой, кандидат педагогических наук,
доцент *С. В. Вабищевич*);
профессор кафедры информационных технологий в культуре
Белорусского государственного университета культуры и искусства,
кандидат физико-математических наук, доцент *П. В. Гляков*

ISBN 978-985-518-412-4

Часть 1

ХЕШИРОВАНИЕ

Потребность в структуре данных, реализующей интерфейс множества (см. раздел **??**), возникает при решении многих задач.

Естественным способом реализации такого интерфейса является дерево поиска. Если использовать сбалансированные бинарные поисковые деревья (например AVL-деревья или красно-чёрные деревья), то для множества из n элементов время выполнения каждой из указанных операций будет величиной $O(\log n)$.

Другим способом эффективной реализации интерфейса множества является хеш-таблица. На практике множества, построенные на хеш-таблицах, обычно работают быстрее, чем множества на основе деревьев, и все основные операции выполняются за $O(1)$. Но для конкретных реализаций на конкретных входных данных ситуация может быть прямо противоположной. Время выполнения основных операций с хеш-таблицей может варьироваться от $O(1)$ до $O(n)$.

В этой части книги рассмотрен принцип построения хеш-таблиц, а также затронуты теоретические аспекты, под ними лежащие. Умения строить хеш-таблицы на практике часто недостаточно для того, чтобы понимать, какие гарантии на время выполнения операций даёт такая структура данных. Оказывается, что хеш-таблицы могут быть быстрее деревьев в некотором доказуемом смысле при выполнении ряда условий.

1.1. УСТРОЙСТВО ХЕШ-ТАБЛИЦЫ

Будем строить структуру данных типа «множество», которая поддерживает базовые операции: добавление ключа, проверка наличия ключа, удаление ключа. Для простоты будем считать, что ключи являются целыми числами из диапазона $[0, N)$.

Обозначим через K множество возможных ключей:

$$K = \{0, 1, 2, \ldots, N-1\}.$$

На практике это множество обычно довольно большое. Часто в качестве ключей в промышленном программировании применяются 32-битные или 64-битные целые числа, т. е. $N = 2^{32} \approx 4{,}2 \cdot 10^9$ или $N = 2^{64} \approx 1{,}8 \cdot 10^{19}$.

1.1.1. Прямая адресация

Если у нас достаточно памяти для массива, число элементов которого равно числу всех возможных ключей, то для каждого возможного ключа можно отвести ячейку в этом массиве и тем самым иметь возможность добраться до любой записи за время $O(1)$.

Таким образом, для хранения множества используется булев массив T размера N, называемый *таблицей с прямой адресацией*. Элемент t_i массива содержит истинное значение, если ключ i входит в множество, и ложное значение, если ключ i в множестве отсутствует. Нетрудно видеть, что все три указанные операции легко выполняются за константное время.

Прямая адресация обладает очевидным недостатком: если множество K всевозможных ключей велико, то хранить в памяти массив T размера N непрактично, а то и невозможно. Кроме того, если число реально присутствующих в таблице записей мало по сравнению с N, то много памяти тратится зря. Размер таблицы с прямой адресацией не зависит от того, сколько элементов реально содержится в множестве.

Минимальным адресуемым набором данных в современных компьютерах является один байт, состоящий из восьми битов. Не представляет трудности реализовать таблицу с прямой адресацией так, чтобы каждый бит был использован для хранения одной ячейки. Если N — мощность множества возможных ключей, то для прямой адресации требуется выделить последовательный блок из как минимум N бит памяти. Так, для размеров множества K в 10^9 элементов таблица займёт около 120 МБ памяти. Во многих случаях такой расход памяти неприемлем, особенно когда есть необходимость создавать несколько таблиц. Тем не менее при сравнительно небольших N метод прямой адресации успешно используется на практике.

1.1.2. Хеш-функция

Хеш-таблицу можно рассматривать как обобщение обычного массива с прямой адресацией.

Мы задумываем некоторую функцию, называемую *хеш-функцией* (англ. *hash function*), которая отображает множество ключей в некоторое гораздо более узкое множество:

$$h : K \to \{0, 1, \ldots, M-1\}, \tag{1.1}$$

$$x \mapsto h(x).$$

Величина $h(x)$ называется *хеш-значением* (*hash value*) ключа x.

Далее вместо того, чтобы работать с ключами, мы работаем с хеш-значениями. При этом возникают так называемые *коллизии* (*collisions*). Это ситуации, когда разные ключи получают одинаковые хеш-значения:

$$x \neq y, \quad h(x) = h(y).$$

Хотелось бы выбрать хеш-функцию так, чтобы коллизии были невозможны. Но в общем случае при $M < N$ это неосуществимо: согласно принципу Дирихле, нельзя построить инъективное отображение из большего множества в меньшее.

Пример 1.1. Приведём примеры некоторых функций. Так, константа $h(x) \equiv 0$ — хеш-функция, для которой любая пара различных ключей будет давать коллизию. Безусловно, такая хеш-функция бесполезна несмотря на то, что она простая и быстро вычисляется. Функция $h(x) = \mathrm{rand}(M)$, всякий раз возвращающая случайное число от 0 до $M-1$ включительно, выбранное равновероятно независимо от x, не может быть использована как хеш-функция, потому что хеш-функция обязана для равных ключей возвращать одинаковые значения. Функция $h(x) = x \bmod M$, возвращающая остаток от деления ключа x на M, является вполне годной для практики хеш-функцией и часто применяется.

Коллизии практически неизбежны, когда требуется осуществлять хеширование случайных подмножеств большого множества K допустимых ключей. Попробуем оценить вероятность коллизии с точки зрения комбинаторики. Рассуждения аналогичны тем, что используются для объяснения парадокса дней рождения в теории вероятностей. Предположим, что хеш-значения ключей независимы и распределены идеально

равномерно от 0 до $M-1$. Пусть мы осуществляем хеширование для n различных ключей ($n \leqslant M$). Когда мы назначаем всем ключам их хеш-значения, мы по сути строим некоторый вектор длины n, каждый элемент которого принимает одно из M значений. Всего существует M^n таких векторов (размещения с повторениями из M по n). Число векторов, в которых все элементы различны, равно $\frac{M!}{(M-n)!}$ (размещения без повторений из M по n). Разделив вторую величину на первую, мы получим вероятность того, что все элементы вектора различны. Значит, вероятность того, что в векторе найдутся хотя бы два одинаковых элемента, т. е. случится коллизия, равна

$$p(M,n) = 1 - \frac{M!}{(M-n)! \times M^n}.$$

Такое выражение неудобно для вычислений, но при помощи формулы Стирлинга для факториала можно вывести приближённые формулы, дающие неплохую точность, когда M велико и n много меньше M, например

$$p(M,n) \approx 1 - e^{\frac{-n^2}{2M}}.$$

Пример 1.2. Допустим, что число M возможных значений хеш-функции равно одному миллиону. Если мы осуществляем хеширование для $n = 2450$ уникальных ключей, то с вероятностью 95 % найдутся такие два ключа, что их хеш-значения будут одинаковыми, т. е. будет иметь место коллизия. Интуитивно кажется, что коллизии очень маловероятны, но это не так.

Если бы коллизий не было, хеш-таблицу можно было бы организовать как массив, в котором в качестве индексов использовались бы хеш-значения. Но из-за того, что возникают коллизии, структура хеш-таблицы более сложная. Разработано несколько стратегий разрешения коллизий.

1.2. РАЗРЕШЕНИЕ КОЛЛИЗИЙ МЕТОДОМ ЦЕПОЧЕК

Образно говоря, хеш-функция раскладывает исходные ключи по *корзинам* (англ. *bins*, *buckets*) или *слотам* (англ. *slots*). Ключ попадает в корзину с номером, равным хеш-значению.

Для хранения элементов с одинаковыми хеш-значениями внутри одной корзины можно использовать связные списки. На верхнем уровне

организуется массив размера M — по числу различных значений хеш-функции, каждый элемент которого — это односвязный список, состоящий из ключей, имеющих конкретное хеш-значение. Возникают цепочки ключей, из-за чего метод и получил название *метода цепочек* (англ. *separate chaining*).

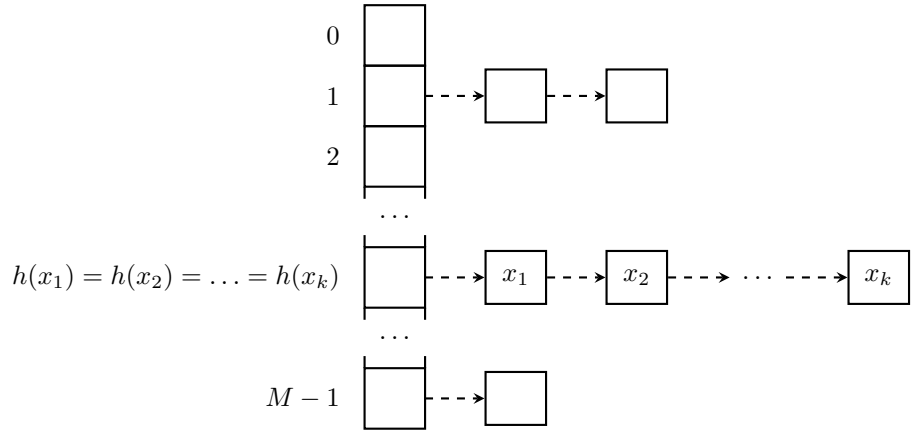

Рис. 1.1. Устройство хеш-таблицы с цепочками

Способ разрешения коллизий с помощью цепочек является одним из довольно часто применяемых на практике, хотя он и не единственный. Определим скорость работы такой хеш-таблицы.

Посмотрим на выполнение операций вставки. Сначала вычисляется хеш-значение $h(x)$ для ключа x, затем происходит обращение к соответствующему связному списку. Если не стоит задача проверять, присутствует элемент x в таблице или нет, то операция вставки может быть реализована за константное время: всегда можно добавить элемент в начало списка, и не нужно идти по всему связному списку. Однако обычно имеет смысл перед вставкой проверить, есть элемент x в таблице или нет, и добавлять только уникальные элементы. Это удобно в силу ряда причин: можно легко отвечать на запросы о числе элементов в множестве, меньше расход памяти (нередко на практике вставок выполняется много, но среди ключей мало различных), проще организовать удаление ключа. Поэтому операция вставки вначале выполняет проход по списку, и на это расходуется время, пропорциональное длине соответствующей цепочки.

Удаление ключа x также требует от нас выполнить прохождение списка в поиске элемента x. Отметим следующий факт: в общем случае из односвязного списка удалить элемент из середины сложно. Однако в рассматриваемом случае, несмотря на то, что список односвязный,

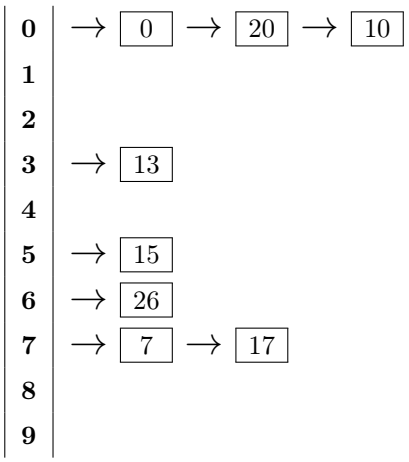

Рис. 1.2. Пример хеш-таблицы для заданного набора ключей

удалять из него нетрудно, потому что мы движемся слева направо и можем поддерживать указатель на текущий элемент и на предыдущий. При удалении указатель у предыдущего элемента перенаправляется на следующий элемент, а память из-под текущего элемента освобождается.

Таким образом, производительность всей конструкции связана с таким параметром, как длина цепочки. Для дальнейшего анализа введём обозначения для длин цепочек: $l_0, \ldots, l_{M-1}$. Для каждого хеш-значения длина своя.

Каждая из трёх рассмотренных операций с ключом x требует времени $\mathrm{O}(1 + l_i)$, где l_i — длина цепочки, в которую попадает ключ x. Отметим важность слагаемого 1 в асимптотике: даже если цепочка имеет нулевую длину, требуется время на то, чтобы вычислить хеш-значение (мы полагаем, что хеш-функция от ключа вычисляется за константу) и обратиться к соответствующей цепочке.

Пример 1.3. Используя хеш-функцию $h(x) = x \bmod 10$, сформируем хеш-таблицу размера 10 из последовательности ключей 0, 20, 15, 13, 26, 7, 17, 10. Результат показан на рис. 1.2.

1.3. РАЗРЕШЕНИЕ КОЛЛИЗИЙ МЕТОДОМ ОТКРЫТОЙ АДРЕСАЦИИ

Альтернативный подход называется *открытой адресацией* (англ. *open addressing*). В линейном массиве хранятся непосредственно ключи, а не заголовки связных списков. Когда выполняется операция вставки, ячейки массива проверяются, начиная с того места, в которое указы-

вает хеш-функция, в соответствии с некоторой *последовательностью проб* (англ. *probe sequence*), пока не будет найдено свободное место. Для осуществления поиска ключа массив проходится в той же последовательности, пока либо не будет найден искомый ключ, либо не будет обнаружена пустая ячейка (в таком случае утверждается, что искомого ключа нет). Операция удаления будет рассмотрена в разделе 1.3.2.

Название «открытая адресация» связано с тем фактом, что положение (адрес) элемента не определяется полностью его хеш-значением. Такой способ также называют «*закрытым хешированием*» (англ. *closed hashing*).

1.3.1. Последовательность проб

Обозначим через $h(x, i)$ номер ячейки в массиве, к которой следует обращаться на i-й попытке при выполнении операций с ключом x. Чтобы формулы были проще, удобно нумеровать попытки с нуля. Последовательность проб для ключа x получается такой:

$$h(x, 0),\ h(x, 1),\ h(x, 2), \ldots$$

Для успешной работы алгоритмов поиска последовательность проб должна быть такой, чтобы в результате M проб все ячейки хеш-таблицы оказались просмотренными ровно по одному разу в каком-либо порядке:

$$\forall x \in K \quad \{h(x, i) \mid i = 0, 1, \ldots, M-1\} = \{0, 1, \ldots, M-1\}.$$

Широко используются три вида последовательностей проб: линейная, квадратичная последовательность проб и двойное хеширование.

Линейное пробирование. В этом случае ячейки хеш-таблицы последовательно просматриваются с некоторым фиксированным интервалом c между ячейками:

$$h(x, i) = (h'(x) + c \cdot i) \bmod M, \tag{1.2}$$

где $h'(x)$ — некоторая хеш-функция. Можно заметить, что не всякое значение c является подходящим.

Теорема 1.1. *Для того чтобы в ходе M проб все ячейки таблицы оказались просмотренными по одному разу, необходимо и достаточно, чтобы число c в формуле* (1.2) *было взаимно простым с размером хеш-таблицы M.*

Доказательство. Воспользуемся сведениями из элементарной теории чисел.

Докажем необходимость. Пусть $h(x, i)$ пробегает все значения от 0 до $M - 1$. Значит, для любого t найдётся такой индекс i, что $c \cdot i \equiv t \pmod{M}$. В частности, это верно для $t = 1$. Следовательно, есть такое i, что $c \cdot i \equiv 1 \pmod{M}$, или, другими словами, $c \cdot i - 1$ делится на M. Пусть d — общий делитель чисел c и M. Тогда число 1 также вынуждено делиться на d. Значит, $d = \pm 1$, т. е. числа c и M взаимно просты и не могут иметь других общих делителей.

Докажем достаточность. Пусть числа c и M взаимно просты. Предположим от противного, что при i-й и j-й пробах получаются одинаковые индексы: $h(x, i) = h(x, j)$. Но это значит, что $ci \equiv cj \pmod{M}$, или $c(i - j) \equiv 0 \pmod{M}$. Отсюда следует, что разность $i - j$ делится на M без остатка. Но раз номера попыток i и j оба лежат на отрезке от 0 до $M - 1$, то это возможно лишь в случае, когда $i = j$. Противоречие. Значит, при всех попытках пробы получаются разными. Поскольку попыток всего M, каждая ячейка будет учтена по одному разу. □

Пример 1.4. Пусть $M = 10$. Формула $h(x, i) = (x + 2i) \bmod 10$ не подходит в качестве последовательности проб. Например, при $x = 5$, подставляя i от 0 до 9, будем получать индексы

$$5, 7, 9, 1, 3, 5, 7, 9, 1, 3,$$

в результате чётные позиции оказываются не посещены. В то же время функция $h(x, i) = (x + 3i) \bmod 10$ работает правильно и возвращает каждый индекс один раз:

$$5, 8, 1, 4, 7, 0, 3, 6, 9, 2.$$

В простейшем случае можно взять единицу в качестве константы c в (1.2). Тогда ячейки просматриваются подряд слева направо, за последней ячейкой просматривается первая. Недостаток такой последовательности проб проявляется в том, что на реальных данных часто образуются кластеры из занятых ячеек (длинные последовательности ячеек, идущих подряд). При непрерывном расположении заполненных ячеек увеличивается время добавления нового элемента и других операций. Таким образом, линейная последовательность проб довольно далека от равномерного хеширования.

Пример 1.5. Пусть $M = 10$, тогда хеш-таблица представляет собой массив из десяти элементов. При осуществлении последовательности проб используется самая простая формула $h(x, i) = (x + i) \bmod 10$. Пусть в хеш-таблицу добавляются ключи

$$7, 29, 67, 38, 25, 27, 18$$

в указанном порядке. Тогда массив будет заполнен следующим образом:

0	1	2	3	4	5	6	7	8	9
38	27	18			25		7	67	29

Квадратичное пробирование. Интервал между ячейками с каждым шагом увеличивается на константу:

$$h(x, i) = (h'(x) + c_1 \cdot i + c_2 \cdot i^2) \bmod M,$$

где числа c_1 и c_2 фиксированы. Значения c_1 и c_2 должны быть тщательно подобраны, чтобы в результате M попыток все ячейки были посещены.

На практике такой метод часто работает лучше линейного, разбрасывая ключи более равномерно по массиву. Тенденции к образованию кластеров нет, но аналогичный эффект проявляется в форме образования вторичных кластеров.

Двойное хеширование. Интервал между проверяемыми ячейками фиксирован, как при линейном пробировании, но, в отличие от него, размер интервала вычисляется второй, вспомогательной хеш-функцией, а значит, может быть различным для разных ключей:

$$h(x, i) = (h'(x) + h''(x) \cdot i) \bmod M.$$

Последовательность проб в этой ситуации при работе с ключом x представляет собой арифметическую прогрессию (по модулю M) с первым членом $h'(x)$ и шагом $h''(x)$.

Чтобы последовательность проб покрыла всю таблицу, значение $h''(x)$ должно быть ненулевым и взаимно простым с M.

Для этого можно поступить следующим образом:

• выбрать в качестве M степень двойки, а функцию $h''(x)$ взять такую, чтобы она принимала только нечётные значения;

• выбрать в качестве M простое число и потребовать, чтобы вспомогательная хеш-функция $h''(x)$ принимала значения от 1 до $M - 1$.

1.3.2. Поддержка операции удаления

Удаление элементов в схеме с открытой адресацией несколько затруднено, и все операции немного усложняются. Рассмотрим один из способов. Для ячейки вводятся три состояния:

- EMPTY — ячейка пуста;
- KEY(x) — ячейка содержит ключ x;
- DELETED — ячейка ранее содержала ключ, но он был удалён.

В любой момент каждая из M ячеек массива находится в одном из трёх состояний. Для удобства обычно поддерживают отдельную целочисленную величину — счётчик ячеек, заполненных актуальными ключами (ячеек в состоянии KEY). Например, это позволяет быстро отвечать на вопрос об общем числе хранящихся элементов. Иначе для этого всякий раз нужно было бы сканировать весь массив за время $O(M)$.

Итак, изначально все ячейки пусты (состояние EMPTY), счётчик выставлен в нуль.

При поиске ключа x необходимо просмотреть все возможные местоположения этого ключа: проверяются ячейки $h(x, 0)$, $h(x, 1)$, ..., пока не найдётся либо ячейка KEY(x) (говорим, что ключ найден), либо ячейка EMPTY (говорим, что искомого ключа в таблице нет). Если встречается ячейка DELETED, процесс пробирования её игнорирует и идёт дальше.

Перед вставкой ключа x сначала выполняется поиск этого ключа. Если ключ уже есть, вставка не требуется. Иначе проверяется принципиальное наличие ячеек, не занятых ключами (используем счётчик занятых ячеек). Если все M ячеек имеют состояние KEY, вставка невозможна: таблица переполнена. Иначе вновь пробуются позиции $h(x, 0)$, $h(x, 1)$, ..., пока не будет найдена либо свободная ячейка EMPTY, либо ячейка с удалённым ключом DELETED (удалённые ячейки при вставке приравниваются к свободным, а при поиске нет). Ячейка переводится в состояние KEY(x), счётчик занятых ячеек увеличиваем на единицу.

При удалении ключа x вначале выполняем поиск ключа x. Если такая ячейка найдена, переводим её в состояние DELETED и счётчик занятых ячеек уменьшаем на единицу.

Нетрудно видеть, что наличие большого числа DELETED-ячеек отрицательно сказывается на времени выполнения операции поиска, а значит и других операций. Чтобы исправить ситуацию, после ряда удалений можно перестраивать хеш-таблицу заново, уничтожая удалённые ячейки.

1.3.3. Плюсы и минусы открытой адресации

Недостаток систем с открытой адресацией состоит в том, что число хранимых ключей не может превышать размер хеш-массива. По факту, даже при использовании хороших хеш-функций, производительность резко падает, когда хеш-таблица оказывается заполненной на 70 % и более. Во многих приложениях это приводит к обязательному использованию динамического расширения таблицы, о котором будет идти речь в разделе 1.4.

Схема с открытой адресацией предъявляет более строгие требования к хеш-функции, чтобы механизм работал хорошо. Кроме того, чтобы распределять значения максимально равномерно по корзинам, функция должна минимизировать кластеризацию хеш-значений, которые стоят рядом в последовательности проб. Так, в методе цепочек одна забота — чтобы много ключей не получило одно и то же хеш-значение, и если хеш-значения получились разные, совершенно всё равно, рядом ли они стоят, отличаются ли на единицу и т. д. Если при использовании метода открытой адресации образовались большие кластеры, время выполнения всех операций может стать неприемлемым даже при том, что заполненность таблицы в среднем невысокая и коллизии редки.

Открытая адресация позволяет существенно экономить память, если размер ключа невелик по сравнению с размером указателя. В методе цепочек приходится хранить в массиве указатели на начала списков, а каждый элемент списка хранит, кроме ключа, указатель на следующий элемент, поэтому на все эти указатели расходуется память.

Открытая адресация не требует затрат времени на выделение памяти на каждую новую запись и может быть реализована даже на миниатюрных встраиваемых системах, где полноценный аллокатор недоступен. Также в открытой адресации нет лишней операции обращения по указателю (*indirection*) при доступе к элементу. Открытая адресация обеспечивает лучшую локальность хранения, особенно с линейной функцией проб. Когда размеры ключей небольшие, это даёт лучшую производительность за счёт хорошей работы кеша процессора, который ускоряет обращения к оперативной памяти. Однако когда ключи «тяжёлые» (не целые числа, а составные объекты), они забивают все кеш-линии процессора, к тому же много места в кеше тратится на хранение незанятых ячеек. Как вариант, можно в массиве с открытой адресацией хранить не сами ключи, а указатели на них. Очевидно, часть преимуществ при этом будет утрачена.

Так или иначе, любой подход к реализации хеш-таблицы может работать достаточно быстро на реальных нагрузках. Время, которое занимают операции с хеш-таблицами, обычно составляет малую долю от общего времени работы программы. Расход памяти редко играет решающую роль. Часто выбор между той или иной реализацией хеш-таблицы делается на основании других факторов в зависимости от ситуации.

1.4. КОЭФФИЦИЕНТ ЗАПОЛНЕНИЯ

Критически важным показателем для хеш-таблицы является *коэффициент заполнения* (англ. *load factor*) — отношение числа ключей, которые хранятся в хеш-таблице, к размеру хеш-таблицы:

$$\alpha = \frac{n}{M}.$$

Коэффициент заполнения может быть как меньше, так и больше единицы (не для всех способов разрешения коллизий такое возможно: например, это недопустимо для разрешения коллизий методом открытой адресации).

Пусть для разрешения коллизий используется метод цепочек. На первый взгляд очевидно, что чем больше коэффициент заполнения, тем медленнее работает хеш-таблица. Однако коэффициент заполнения не показывает различия между заполненностью отдельных корзин. Например, пусть есть две хеш-таблицы, в каждой используется 1000 корзин и хранится всего 1000 ключей, поэтому коэффициент заполнения в обоих случаях равен единице. Однако в первой таблице в каждой цепочке по одному ключу, а во второй все ключи лежат в одной длинной цепочке. Очевидно, что вторая хеш-таблица будет работать очень медленно.

Низкий коэффициент заполнения не является абсолютным благом. Если коэффициент близок к нулю, это говорит о том, что большая часть таблицы не используется и память тратится впустую.

Для оптимального использования хеш-таблицы желательно, чтобы её размер был примерно пропорционален числу ключей, которые нужно хранить. На практике редко случается, что число ключей фиксировано и можно заранее выставить хорошее значение параметра M. Если ставить его заведомо больше, то много памяти будет потрачено зря (особенно если нужно организовать много хеш-таблиц с небольшим числом ключей в каждой).

Реализация хеш-таблицы общего назначения обязана поддерживать операцию изменения размера.

Часто используемым приёмом является автоматическое изменение размера, когда коэффициент заполнения превышает некоторый порог $\alpha_{\max}$. Выделяется память под новую, бóльшую таблицу, все элементы из старой таблицы перемещаются в новую, затем память из-под старой хеш-таблицы освобождается. Аналогично, если коэффициент заполненности опускается ниже другого порога $\alpha_{\min}$, элементы перемещаются в хеш-таблицу меньшего размера.

Чисто технически в языках программирования под хеш-функцией понимается функция, отображающая ключ в произвольное целое число без ограничения на величину (не требуется попадания в промежуток $[0, M)$, как в определении (1.1)). В этом есть здравый смысл. Хеш-значение — это свойство ключей, и оно не зависит от размера хеш-таблицы. Тот факт, что мы далее берём его по модулю M, — это исключительно особенность его использования внутри конкретной хеш-таблицы конкретного размера. Не нужно предварительно брать остаток. Часто на практике хеш-кодами выступают всевозможные беззнаковые целые числа, и остаток берётся по нужному модулю непосредственно в месте использования.

1.5. ОЦЕНКИ ВРЕМЕНИ ВЫПОЛНЕНИЯ ОПЕРАЦИЙ

Оценим время выполнения операций над хеш-таблицей в случае разрешения коллизий методом цепочек. Метод открытой адресации кажется существенно более сложным для теоретического анализа.

Нетрудно понять, что от выбора хеш-функции многое зависит. Если она будет «плохая» (например константная), то будут иметь место сплошные коллизии (хеш-таблица превратится в связный список). Как определить меру качества хеш-функции и научиться получать «хорошие» хеши?

За годы развития вычислительной техники сформировалась наука о том, как практически строить хеш-функции. Разработаны всевозможные правила на тему того, какие биты ключа x стоит взять, на сколько позиций выполнить поразрядный сдвиг, как применить операцию исключающего «или», на что умножить, чтобы получить хеш, который бы выглядел как случайный. Чтобы найти примеры практических хеш-функций, можно взять какую-нибудь стандартную библиотеку

языка программирования (см. раздел 1.8). Эти функции часто ничего не гарантируют: коллизии, конечно, есть. Отметим, что коллизии неизбежны. Любую фиксированную хеш-функцию можно всегда «сломать» — придумать ситуацию, когда там будут одни сплошные коллизии.

Как же тогда оценивать трудоёмкость? Когда длины цепочек «хорошие», всё работает быстро. Однако на практике длины цепочек бывают «плохими». Хотелось бы мыслить в терминах оценок сверху, однако в худшем случае время работы оказывается неприемлемым. Теоретически есть два направления возможной деятельности.

Первое направление заключается в том, чтобы добавить рандомизацию: внести элемент случайности, чтобы не было фиксированного «плохого» случая. Есть идея не брать какую-либо одну конкретную хеш-функцию, а, например, организовать программу так, чтобы при каждом запуске хеш-функция выбиралась заново. Тогда одним и тем же входом «сломать» программу не получится, и можно будет рассуждать о длине цепочки с точки зрения теории вероятностей: говорить о том, какая длина цепочки в среднем, какая у неё дисперсия... Такой подход называют *универсальным хешированием* (англ. *universal hashing*).

Второе направление состоит в следующем. Несмотря на то, что потенциальное множество ключей велико, в каждый момент времени мы храним ключей меньше, чем число корзин в хеш-таблице. Пусть S — фактическое множество ключей в хеш-таблице, и у этого множества размер n, где $n \leqslant M$. Тогда понятно, что существует хеш-функция, которая не даёт коллизий: если элементов меньше, чем слотов, то их всегда можно отобразить без коллизий. Это хорошая хеш-функция, её называют *совершенной* (англ. *perfect hash*). В чём проблема? Она всегда есть и зависит от набора ключей. Даже если предположить, что набор ключей константный и известен заранее, есть трудность в том, как эффективно задать эту хеш-функцию.

Использовать для задания функции таблицу значений непрактично, потому что множество K образов для этой функции слишком большое. Если применять какую-либо ассоциативную структуру, это ровно та же задача, которую мы пытаемся решать (если для вычисления хеш-функции делать запрос в хеш-таблицу, то какую хеш-функцию использовать в этой новой хеш-таблице?). Другое решение: например, можно все значения S отсортировать по возрастанию и в качестве хеш-функции элемента использовать его порядковый номер. Это правильно с точки зрения избавления от коллизий, но как тогда определять номер

элемента во время работы программы? Мы не можем использовать бинарный поиск (см. часть **??**), потому что он требует логарифмического времени и сведёт на нет смысл построения хеш-таблицы. Удобно, когда хеш-функция — небольшое выражение, которое может быть вычислено за константное время.

Итак, возникает задача построения хеш-функции, которая является

- простой, т. е. достаточно константного объёма памяти, чтобы её хранить;
- быстрой, т. е. требуется константное время на вычисление;
- совершенной, т. е. коллизии отсутствуют.

Оказывается, сделать это можно, но рассмотрение этого вопроса выходит за рамки данной книги.

1.6. ИДЕАЛЬНОЕ ХЕШИРОВАНИЕ

Первая идея — взять в качестве h случайную функцию. Это так называемое *идеальное хеширование* в том смысле, что оно нереализуемо на практике и существует лишь в теории.

Что значит «h — случайная функция»? Функцию можно представить как таблицу, в которой для каждого значения аргумента написано, чему функция равна. Под случайной функцией понимают функцию, у которой каждое значение выбирается случайно равновероятно на множестве $\{0, 1, \ldots, M-1\}$ и все значения выбираются независимо. Мы пытаемся заполнить таблицу числами от 0 до $M-1$, причём числа могут повторяться, количество элементов в этой таблице равно N. Каждая клетка выбирается независимо в соответствии с равномерным распределением.

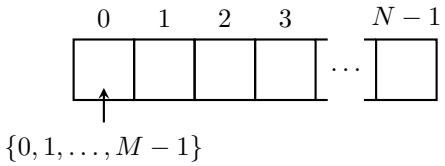

Рис. 1.3. Задание идеальной хеш-функции с помощью таблицы

1.6.1. Оценка длины цепочки

Идеальная хеш-функция позволяет получить хорошую оценку для длины цепочки.

Пусть $S = \{x_1, \ldots, x_n\}$. Для фиксированного хеш-значения t длина цепочки l_t будет случайной величиной. Определим её математическое ожидание:

$$\mathbf{E}\{l_t\} = \mathbf{E}\left\{\sum_{i=1}^{n} \mathbb{1}_{\{h(x_i)=t\}}\right\},$$

где $\mathbb{1}_{\{h(x_i)=t\}}$ — индикатор того события, что i-й ключ попал в список, соответствующий хеш-значению t, т. е.

$$\mathbb{1}_{\{h(x_i)=t\}} = \begin{cases} 1, & \text{если } h(x_i) = t, \\ 0, & \text{иначе.} \end{cases}$$

Нетрудно видеть, что

$$\mathbf{E}\left\{\mathbb{1}_{\{h(x_i)=t\}}\right\} = 1 \cdot \mathbf{P}\{h(x_i) = t\} + 0 \cdot \mathbf{P}\{h(x_i) \neq t\} = \frac{1}{M}$$

в силу того, что $h(x_i)$ равновероятно принимает некоторое значение из M возможных, одно из которых t, и не зависит от i.

Из линейности математического ожидания следует, что

$$\mathbf{E}\{l_t\} = \sum_{i=1}^{n} \mathbf{E}\left\{\mathbb{1}_{\{h(x_i)=t\}}\right\} = \frac{n}{M} = \alpha,$$

где α — введённый ранее в разделе 1.4 коэффициент заполнения хеш-таблицы, равный отношению количества хранящихся в хеш-таблице элементов к количеству цепочек.

Аналогично можно вычислить дисперсию длины цепочки. Действительно, каждый индикатор является дискретной случайной величиной, имеющей распределение Бернулли с вероятностью успеха $1/M$, поэтому

$$\mathbf{V}\left\{\mathbb{1}_{\{h(x_i)=t\}}\right\} = \frac{1}{M}\left(1 - \frac{1}{M}\right).$$

Поскольку случайные величины независимы, дисперсия суммы равна сумме дисперсий:

$$\mathbf{V}\{l_t\} = \sum_{i=1}^{n} \cdot \mathbf{V}\left\{\mathbb{1}_{\{h(x_i)=t\}}\right\} = \frac{n}{M}\left(1 - \frac{1}{M}\right) < \frac{n}{M} = \alpha.$$

Получается, что длина цепочки имеет константное матожидание и константную дисперсию. В среднем элементы располагаются по корзинам равномерно и длины цепочек небольшие.

1.6.2. Сложность идеальной хеш-функции

С точки зрения теории информации хеш-функция сложна, потому что количество информации, содержащейся в этой функции, велико. Где-то надо хранить эту информацию и быстро получать к ней доступ.

Представлять, что хеш-функция — это таблица из N ячеек, непрактично, потому что N обычно очень велико. Если бы можно было создать такую таблицу, не было бы смысла использовать хеширование вместо прямой адресации, когда по ключу прямо вынимается значение.

Таблицу можно строить не сразу, а постепенно. А именно, когда спрашивают значение $h(x)$, можно сгенерировать случайное значение и его выдать. Величина $h(x)$ случайна, но для одного и того же x хеш-функция должна возвращать одни и те же значения. Поэтому необходимо хранить для уже встретившихся x ранее выданные значения, а эта задача аналогична той, которую мы решаем.

На практике идеальных хеш-функций не бывает, но они обладают рядом интересных свойств. Во-первых, этих свойств достаточно для того, чтобы получать некоторые оценки. Во-вторых, существуют способы построения функций h, обладающих теми же свойствами, но построение этих функций менее затратно.

Нужно отказаться от сильного свойства полной покоординатной случайности и независимости и заменить на что-то более простое, но так, чтобы сохранить оценку матожидания (дисперсию сохранить, увы, не удастся).

Заметим, что для получения оценки матожидания мы мало чем пользовались. Так, полученная оценка верна, даже если хеш-функция на всех аргументах принимает одно и то же случайно выбранное значение. Вообще, в анализе алгоритмов хорошее матожидание мало что значит, важнее оценивать матожидание в совокупности с дисперсией, а ещё лучше оценивать худший случай с высокой вероятностью, т. е. оценку, которая верна «почти всегда».

1.7. УНИВЕРСАЛЬНОЕ ХЕШИРОВАНИЕ

Универсальное хеширование (англ. *universal hashing*) — это хеширование, при котором хеш-функция выбирается случайным образом из определённого семейства хеш-функций

$$H = \{h : K \to \{0, 1, \ldots, M-1\}\}.$$

Семейство хеш-функций H называется *универсальным* (англ. *universal*), если для любых двух различных значений ключа x и y вероятность получить коллизию ведёт себя ожидаемым образом, т. е.

$$\mathbf{P}_{h \in H} \{h(x) = h(y)\} = \mathrm{O}\left(\frac{1}{M}\right). \tag{1.3}$$

Когда константа, скрытая в асимптотике, равна единице, семейство называют *сильно универсальным*. Часто для теоретических рассуждений величина константы особой роли не играет.

Можно пытаться усилить универсальность и ввести понятие k-независимости. Семейство функций H называют *k-независимым*, если для любых различных ключей $x_1, x_2, \ldots, x_k$ и для любых возможных значений хеш-функций $t_1, t_2, \ldots, t_k$ имеет место условие

$$\mathbf{P}_{h \in H} \{h(x_1) = t_1,\ h(x_2) = t_2,\ \ldots,\ h(x_k) = t_k\} = \mathrm{O}\left(\frac{1}{M^k}\right). \tag{1.4}$$

Формально у нас есть некоторое семейство хеш-функций H с некоторым распределением вероятностей на нём, и $h \in H$ выбирается в соответствии с этим распределением. Важно понимать, что в формулах (1.3) и (1.4) вероятности берутся не по значениям ключей (все x_i фиксируются), а по h. Поэтому на множестве всех h есть некоторое распределение (обычно равномерное, но не обязательно). Итак, есть семейство H, и из этого семейства мы выбираем h.

В эту схему, когда хеш-функция выбирается из семейства, отлично вкладывается концепция идеальной хеш-функции. В этом случае H — множество всех функций с равномерным распределением. Свойство универсальности, конечно, всегда справедливо для идеальной хеш-функции. Более того, оно справедливо с жёсткой константой, равной единице.

1.7.1. Построение универсального семейства хеш-функций для целочисленных ключей

Рассмотрим один способ построения универсального семейства хеш-функций для целочисленных ключей. Метод был предложен Дж. Л. Картером (J. L. Carter) и М. Н. Вегманом (M. N. Wegman) в 1977 г. Фактически семейство будет даже сильно универсальным. Способ линейно-алгебраический по содержанию и состоит из двух шагов.

На первом шаге возьмём простое число p, которое больше или равно N. Например, можно взять ближайшее простое не меньше N. По

разным свойствам распределений простых чисел мы знаем, что p будет, вообще говоря, недалеко от N. Так, постулат Бертрана гласит, что существует достаточно близкое к N простое число: $N \leqslant p < 2N$.

Пример 1.6. Для $N = 1000$ можно взять $p = 1009$.

Через $\mathbb{Z}_p$ будем обозначать множество наименьших неотрицательных вычетов по модулю p:

$$\mathbb{Z}_p = \{0, 1, 2, \ldots, p-1\}.$$

На втором шаге мы независимо формируем два случайных вычета по модулю p, один из которых ненулевой:

$$a \in \mathbb{Z}_p \setminus \{0\}, \quad b \in \mathbb{Z}_p.$$

После того как эти два значения выбраны, возникает функция, задаваемая выражением

$$(ax + b) \bmod p. \tag{1.5}$$

Заметим, что раз $N \leqslant p$, то ключ x, как a и b, является вычетом по модулю p, но старшая часть вычетов запрещена (от N до $p-1$ включительно).

Формула (1.5) для любого ключа x даёт на выходе вычет по модулю p, который в качестве хеш-значения не годится, потому что p может быть велико. Есть отличный способ сделать его меньше — взять по модулю M:

$$h_{a,b}(x) = [(ax + b) \bmod p] \bmod M. \tag{1.6}$$

Полученная функция $h_{a,b}(x)$ может служить хеш-функцией для построения хеш-таблиц размера M.

Пример 1.7. Пусть $M = 10$ и $N = 1000$. Зафиксируем простое число $p = 1009$. Тогда коэффициент a можно выбирать от 1 до 1008 (всего 1008 способов), коэффициент b можно выбирать от 0 до 1008 (всего 1009 способов). Итого можно получить $1008 \times 1009 = 1\,017\,072$ разных функции, например

$$\begin{aligned} h_{3,4}(x) &= ((3x + 4) \bmod 1009) \bmod 10, \\ h_{670,905}(x) &= ((670x + 905) \bmod 1009) \bmod 10, \\ h_{101,0}(x) &= ((101x \bmod 1009) \bmod 10, \\ &\ldots \end{aligned} \tag{1.7}$$

Заметим, что в общем случае нельзя «для простоты счёта» опустить первое взятие остатка, например

$$((1010 \bmod 1009) \bmod 10) = 1 \neq 0 = (1010 \bmod 10).$$

Конструкция (1.6) выглядит странно: часто один вычет по модулю другого вычета не имеет особого смысла, только если один модуль не является делителем другого. Но в данном случае h как раз оказывается хеш-функцией, выбранной равномерно из универсального семейства. Справделива следующая теорема.

Теорема 1.2. *Семейство функций*

$$H = \{h_{a,b} \mid a, b \in \mathbb{Z}_p, a \neq 0\},$$

где $h_{a,b}(x)$ определяется по формуле (1.6)*, число p простое, является сильно универсальным.*

Пример 1.8. Пусть $N = 1000$, $M = 10$. Выберем p равным 1009, как в примере 1.7. Ранее отмечалось, что $|H| = 1\,017\,072$.

Проверим теорему 1.2 на практике. Зафиксируем два различных ключа x и y, например $x = 1$ и $y = 2$. Затем промоделируем процесс выбора хеш-функции на компьютере: будем перебирать в двух вложенных циклах a от 1 до 1008 и b от 0 до 1008 включительно, каждый раз будем вычислять хеши $h(x)$ и $h(y)$, сравнивать их и подсчитывать число случаев, когда случается коллизия. Оказывается, что это число равно 100 800 и не зависит от x и y. Значит, вероятность того, что на данной паре ключей при случайном выборе хеш-функции из универсального семейства будет коллизия, равна $\frac{100\,800}{1\,017\,072} \approx 0{,}0991 < 0{,}1 = \frac{1}{M}$.

Можно рассмотреть хеш-функции более простого вида:

$$h_a(x) = (ax \bmod p) \bmod M. \tag{1.8}$$

Теорема 1.3. *Семейство функций*

$$H = \{h_a \mid a \in \mathbb{Z}_p, a \neq 0\},$$

где $h_a(x)$ определяется по формуле (1.8)*, число p простое, обладает свойством универсальности, но не является сильно универсальным.*

1.7.2. Универсальное хеширование векторов

Если в качестве ключей выступают не просто целые числа, а объекты других типов, то нужно использовать соответствующие универсальные семейства хеш-функций.

Например, пусть ключ представляет собой вектор фиксированной длины

$$x = (x_1, x_2, \ldots, x_r),$$

составленный из целых чисел, каждое из которых лежит в отрезке от 0 до $N - 1$, при этом N небольшое. Размер хеш-таблицы M выберем простым и таким, чтобы выполнялось условие $M \geqslant N$.

Хеш-функцию будем строить по формуле

$$h_a(x) = \sum_{i=1}^{r} a_i x_i \bmod M, \tag{1.9}$$

где $a = (a_1, a_2, \ldots, a_r)$ — случайно выбираемый вектор из вычетов по модулю M. Нетрудно видеть, что всего существует M^r таких векторов.

Теорема 1.4. *Семейство H, составленное из всех функций вида* (1.9)*, является универсальным семейством хеш-функций.*

Пример 1.9. IP-адрес представляет собой 32-битное целое число, которое часто записывают как четвёрку 8-битных целых чисел, например 192.168.1.2. Пусть нужно организовать хранение примерно 250 IP-адресов в хеш-таблице.

Каждый адрес можно рассматривать как четырёхкомпонентный вектор. Тогда получается, что $N = 256$. Размер таблицы M предлагается взять равным 257 (это простое число).

Для выбора хеш-функции из универсального семейства нужно сгенерировать четыре случайных целых числа из отрезка от 0 до 256 включительно, а потом подставить их в формулу (1.9). Например, если выбрано $a = (87, 23, 125, 4)$, то получается такая хеш-функция:

$$h(x) = (87x_1 + 23x_2 + 125x_3 + 4x_4) \bmod 257.$$

1.8. ХЕШ-ТАБЛИЦЫ НА ПРАКТИКЕ

Структуры данных на основе хеш-таблиц реализованы в стандартных библиотеках всех широко используемых языков программирования. В большинстве случаев библиотеки предоставляют как множество, так и ассоциативный массив.

1.8.1. Требования к ключам

Зачастую при программировании ключами в множествах и ассоциативных массивах выступают не просто целые числа, а какие-либо более сложные объекты, например строки, пары чисел, структуры из нескольких полей и прочие объекты произвольной природы.

Чтобы объект некоторого типа мог выступать ключом в хеш-таблице, необходимо выполнение двух условий:

- должна быть задана хеш-функция, которая ставит в соответствие объекту его хеш-значение (целое число, причём диапазон допустимых значений оговаривается в спецификациях);
- должна быть определена функция, которая для пары объектов отвечает, равны они или нет.

Формально от хеш-функции требуется лишь, чтобы для равных объектов хеш-значения были равными. Функция проверки на равенство должна задавать на множестве объектов отношение эквивалентности, то есть бинарное отношение, для которого выполнены следующие условия:

1) $a = a$ для любого a (свойство рефлексивности);
2) если $a = b$, то $b = a$ (свойство симметричности);
3) если $a = b$ и $b = c$, то $a = c$ (свойство транзитивности).

Для того чтобы объект мог служить ключом в бинарном дереве поиска, требование выдвигается совсем иное: нужно уметь сравнивать любые два элемента. Строго говоря, множество объектов должно быть линейно упорядочено. Напомним, что линейным порядком на множестве называют бинарное отношение $\leqslant$, которое удовлетворяет следующим условиям:

1) любые два объекта сравнимы между собой, т. е. верно хотя бы одно из неравенств $a \leqslant b$ или $b \leqslant a$ (свойство полноты);
2) если $a \leqslant b$ и $b \leqslant a$, то $a = b$ (свойство антисимметричности);
3) если $a \leqslant b$ и $b \leqslant c$, то $a \leqslant c$ (свойство транзитивности).

Такой порядок также называют *полным порядком* (англ. *total order*). Линейно упорядоченные объекты могут быть размещены на прямой линии один за другим. Первое свойство влечёт за собой свойство рефлексивности, т. е. $a \leqslant a$. Следовательно, линейный порядок является также частичным порядком. Понятие частично упорядоченного множества слабее (в определении частичного порядка в первом пункте требуется лишь рефлексивность).

От языка программирования зависит способ, при помощи которого можно определять те или иные операции для объектов произвольного

типа (подсчёт хеш-функции, проверка на равенство или неравенство).

На программиста возлагается обязанность соблюдать условия. Например, если функция проверки двух объектов на равенство будет возвращать каждый раз случайное значение (истинное или ложное), естественно ожидать, что хеш-таблица будет работать неправильно. Если функция сравнения для множества на основе дерева не обеспечивает линейного упорядочивания, программа будет вести себя непредсказуемо и, возможно, аварийно завершится с ошибкой.

1.8.2. Объединение хеш-значений

Нередко на практике требуется реализовать хеш-функцию от структуры, содержащей два поля, для каждого из которых по отдельности хеш-функции определены. Например, координаты точек на плоскости хранятся в виде пар целых чисел (x, y), и нужно создать множество точек с использованием хеш-таблицы, а для этого необходимо вычислять хеш-значения от точек. Пусть получены хеш-значения двух координат $h(x)$ и $h(y)$. Как их объединить, чтобы получить хеш от пары? Пусть для простоты верхняя граница возможных значений хеш-функции не фиксирована (где-то дальше в реализации хеш будет взят по нужному модулю M).

Часто на практике программисты для соединения хешей пишут тривиальные функции, например через операцию побитового исключающего или (xor):

```
def combine(hx, hy):
    return hx ^ hy
```

Такой вариант часто работает на практике приемлемо, но не лишён очевидных недостатков. Например, для всех точек с равными координатами x и y хеш-функция будет принимать нулевое значение, и если точек на прямой $y = x$ во входных данных окажется много, производительность будет низкой из-за коллизий. Также очевидно, что разные точки (x, y) и (y, x), симметричные относительно той же прямой, получат одинаковые хеш-значения.

Чтобы подобрать пары, дающие коллизию, было труднее, для объединения хешей используют более сложные функции с обилием «магических» констант и странных операций. Например, в C++-библиотеке boost используется примерно такая формула:

```
def combine(hx, hy):
    return hx ^ (hy + 0x9e3779b9 + (hx << 6) + (hx >> 2))
```

Часто берут линейную комбинацию двух хеш-значений с, например, большими взаимно простыми коэффициентами. Как вариант:

```
def combine(hx, hy):
    return hx + 1000000007 * hy
```

Основной смысл таких манипуляций — сделать так, чтобы на реально встречающихся в жизни данных коллизии были более редкими. Но контрпример при желании можно подобрать. Лучшего универсального решения в этом деле нет.

Отметим, что если научиться объединять хеши двух элементов, то можно последовательно объединить хеши любого числа элементов.

1.8.3. Проход по содержимому хеш-таблицы

В процессе программирования может возникнуть необходимость выполнить обход всех элементов структуры данных и, например, распечатать их. Предположим, требуется вывести все ключи, которые содержатся в заданном множестве.

Функция для итерации по содержимому структуры данных не вводилась в разделе **??**, но является полезной, поэтому обычно поддерживается в реализациях хеш-контейнеров, с которыми ведётся работа на практике.

В большинстве реализаций проход по хеш-множествам выполняется в произвольном порядке, не гарантируется какой-либо отсортированности ключей. В случае, если внутренняя реализация хеш-таблицы использует метод цепочек, обычно функция обхода выдаёт сначала все элементы первой корзины (с хеш-значением 0) в порядке их следования в цепочке, затем все элементы второй корзины (с хеш-значением 1), и т. д. Для внешнего наблюдателя может казаться, что ключи выходят в случайном порядке.

Более того, если распечатать элементы хеш-множества, добавить новый ключ, сразу удалить его, вновь распечатать элементы, то порядок может получиться другим. Такое может случиться, если добавление нового ключа привело к перестроению хеш-таблицы с изменением числа M корзин и элементы были перераспределены по корзинам вновь.

Не стоит нигде в коде закладываться на порядок итерации по хеш-контейнерам: большинство реализаций в разных языках программирования могут гарантировать только то, что посещены будут все элементы, не важно в каком порядке.

Наоборот, средства итерации по ключам множества, которое построено на базе бинарного поискового дерева, обычно возвращают ключи в порядке возрастания (выполняется внутренний обход дерева). Порядок фиксирован и каждый раз одинаковый. Часто предсказуемость результата удобна, например, для написания модульных тестов к частям программы.

Таким образом, если порядок итерации важен, возможно, стоит использовать «древесные» структуры данных.

1.8.4. Хеш-таблицы в C++

Долгое время в языке C++ не было стандартных реализаций структур данных на основе хеш-таблиц. Контейнеры `std::set` и `std::map` из STL строятся на основе сбалансированных бинарных поисковых деревьев (во всех популярных реализациях применяются красно-чёрные деревья). Хеш-таблицы существовали в виде нестандартных расширений (например `stdext::hash_set` в Visual Studio) или внешних библиотек (например boost).

Наконец, в стандарте C++11 в STL официально были добавлены хеш-таблицы. Стандарт предусматривает четыре контейнера на основе хеш-таблиц, которые отличаются от своих аналогов на основе деревьев наличием префикса `unordered_` в названии. Так, `std::unordered_set` представляет собой динамическое множество, `std::unordered_map` — ассоциативный массив. Существует также два `multi`-контейнера, которые допускают хранение одинаковых ключей.

Стандарт требует, чтобы в построении этих структур данных авторы компиляторов использовали разрешение коллизий методом цепочек. Метод открытой адресации не был стандартизирован из-за внутренних трудностей при удалении элементов. Однако детали реализации хеш-таблиц стандартом не регламентируются.

В качестве хеш-значения в C++ используется число типа `size_t`. Все хеш-контейнеры предоставляют метод `rehash()`, который позволяет установить размер хеш-таблицы (число корзин M). Метод под названием `load_factor()` возвращает текущий коэффициент заполнения.

Рассмотрим более подробно реализацию в компиляторе GCC. Пусть ключи добавляются в `std::unordered_set` по одному. Когда коэффициент заполнения достигает значения 1, происходит перестроение хеш-таблицы: в качестве нового числа корзин берётся первое простое

число из заранее составленного списка, не меньшее удвоенного старого числа корзин (таким образом, размер таблицы как минимум удваивается и является простым числом). Длины отдельных цепочек никак не анализируются (появление одной длинной цепочки не повлечёт за собой операцию перестроения).

1.8.5. Хеш-таблицы в Java

Предполагаем, что речь идёт о языке Java актуальных на момент написания этих строк версий.

Коллекции `HashSet` и `HashMap` реализуются как хеш-таблицы, для разрешения коллизий используется метод цепочек.

Для хеширования целых чисел применяется функция следующего вида:

```
int hash(int h) {
    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
}
```

(здесь операция `>>>` — беззнаковый сдвиг вправо: биты смещаются вправо, число слева дополняется нулями, операция `^` — поразрядное сложение по модулю 2, исключающее «или»). Затем в классе коллекции результат функции `hash` берётся по модулю числа корзин, по которым раскладываются элементы. Число корзин, оно же число различных значений хеш-функции M, в Java всегда выбирается как некоторая степень числа 2, чтобы деление на M можно было заменить операцией битового сдвига вправо (современные процессоры выполняют инструкцию деления целых чисел существенно медленнее, чем битовые операции).

В версии Java 8 разработчики озаботились вопросом устойчивости коллекций, использующих хеширование, к коллизиям. В исходном коде библиотеки Java можно найти следующую константу:

```
static final int TREEIFY_THRESHOLD = 8;
```

В случае, если новый ключ попадает в корзину, в которой уже лежат как минимум восемь других ключей, библиотека преобразует связный список для данной корзины в бинарное сбалансированное поисковое дерево. Получается гибридная структура: корзины для тех хеш-значений, где ключей мало, хранятся списками, а корзины, где ключей накопилось много, хранятся в виде деревьев.

1.8.6. Хеш-таблицы в Python

Рассмотрим реализацию языка программирования CPython версии 2.7.

Встроенный тип `dict` — ассоциативный массив, словарь — очень широко используется в языке. Он реализован в виде хеш-таблицы, где коллизии разрешаются методом открытой адресации. Разработчики предпочли метод открытой адресации методу цепочек ввиду того, что он позволяет значительно сэкономить память на хранении указателей, которые используются в хеш-таблицах с цепочками.

Подробное описание принципов устройства словаря в Python на русском языке можно найти в сети Интернет.

Интерпретатором CPython поддерживается опция командной строки `-R`, которая активирует на старте случайный выбор начального значения (англ. *seed*), которое затем используется для вычисления хеш-значений от строк и массивов байт.

1.8.7. Атаки против стандартных хеш-функций

Для целочисленной хеш-функции, которая используется в Java, оказалось нетрудно придумать обратную, поскольку функция выполняет линейное преобразование битов исходного ключа. С использованием этого результата можно строить наборы различных ключей, которые при добавлении в `HashSet` попадают в одну корзину.

Более того, на одних и тех же входных данных может наблюдаться деградация производительности сразу нескольких реализаций хеш-таблиц. Так, в рамках соревнования IPSC 2014 была предложена следующая задача. Дано две программы. Первая программа на языке C++ просто читает числа на входе и добавляет их по одному в множество на основе `std::unordered_set`. Вторая программа на языке Java полностью аналогична, выполняет чтение чисел и занесение их в `HashSet`. Требовалось построить такую последовательность из не более чем 50 тысяч 64-битных целых чисел, чтобы время работы обоих программ превысило 10 секунд на сервере организаторов (указывались особенности архитектуры и версии компиляторов обоих языков программирования). Задача была решена многими участниками, т. е. им удалось подобрать такой набор входных данных, что построение обеих хеш-таблиц заняло квадратичное время из-за коллизий.

Существует целый класс атак на серверы, называемый *hash-flooding DoS* — отказ в обслуживании, вызванный заполнением хеша. Злоумыш-

ленники формируют специальные запросы, которые вызывают на сервере большое число хеш-коллизий и оттого медленно обрабатываются. В результате вычислительные ресурсы тратятся впустую, легальные пользователи системы не могут получить доступ к ресурсам либо этот доступ затруднён.

1.8.8. Криптографические хеш-функции

Для приложений в области криптографии разработаны специальные хеш-функции. Можно считать, что в криптографии множество K возможных ключей бесконечно, и любой блок данных является ключом (в принципе, произвольный массив байт можно рассматривать как двоичную запись некоторого числа). Хеш-функция $h(x)$ называется криптографической, если она удовлетворяет следующим требованиям:

- *необратимость*: для заданного значения хеш-функции c должно быть сложно определить такой ключ x, для которого $h(x) = c$;
- *стойкость к коллизиям первого рода*: для заданного ключа x должно быть вычислительно невозможно подобрать другой ключ y, для которого $h(x) = h(y)$;
- *стойкость к коллизиям второго рода*: должно быть вычислительно невозможно подобрать пару ключей x и y, имеющих одинаковый хеш.

Криптографические хеш-функции обычно не используются в хеш-таблицах, потому что они сравнительно медленно вычисляются и имеют большое множество значений. Зато такие хеш-функции широко применяются в системах контроля версий, системах электронной подписи, во многих системах передачи данных для контроля целостности.

Примерами криптографических хеш-функций являются алгоритмы MD5, SHA-1, SHA-256. Так, метод SHA-1 ставит в соответствие произвольному входному сообщению некоторую 20-байтную величину, т. е. результат вычисления SHA-1 принимает одно из 2^{160} различных значений. Вот пример вычисления SHA-1 от ASCII-строки, где результат записан в шестнадцатеричной системе счисления:

```
SHA-1("The quick brown fox jumps over the lazy dog")
  = 0x2fd4e1c67a2d28fced849ee1bb76e7391b93eb12
```

В настоящий момент коллизии для MD5 и SHA-1 обнаружены, поэтому методы постепенно выходят из широкого использования. Более

новые алгоритмы семейства SHA-2 считаются существенно более стойкими к коллизиям. Тем не менее следует понимать, что коллизии есть обязательно, потому что нельзя биективно отобразить бесконечное множество в конечное. Вопрос только в том, насколько трудно эти коллизии отыскать.

1.9. ВЫВОДЫ

Таким образом, рассмотрены подходы к реализации динамических множеств и ассоциативных массивов на основе хеш-таблиц в теории и на практике.

Как уже говорилось, альтернативой хеш-таблицам являются структуры на основе сбалансированных бинарных поисковых деревьев. Возникает вопрос, как теоретический, так и практический, о том, насколько эти способы отличаются, какой из них лучше или хуже. Ответ здесь такой: хеш-таблицы в некотором доказуемом смысле могут быть быстрее.

Интуитивно более-менее понятно, почему так: в случае хеш-таблицы мы работаем с ключами, которые являются просто числами, причём на этих числах не предполагается никакого порядка, при этом интерфейс не может данный порядок воспроизвести. Например, нельзя найти все ключи от a до b, просто потому что на ключах порядка как такового нет. Конечно, если ключи — это числа, то порядок возникает, но никакой семантики не несёт. Мы отказываемся от требования упорядоченности, потому можем сделать структуру данных более эффективной.

Так, если к исходному интерфейсу множества (см. раздел **??**) добавить операцию типа LowerBound(x) — найти первый ключ, больший либо равный x, то получится уже не просто множество (англ. *set*), а упорядоченное множество (англ. *ordered set*). Новая операция осуществляет поиск в окрестности значения x. На первый взгляд единственный доступный нам приём все эти операции поддерживать — это использовать деревья поиска. Оказывается, нет. Существуют интересные гибриды, находящиеся посередине между деревьями поиска и хеш-таблицами, они в частности реализуют концепцию упорядоченного множества. Это всевозможные деревья Ван Эмде Боасса (Van Emde Boas tree), X-fast-, Y-fast- и Fusion-деревья, у которых в оценках временной сложности появляется двойной логарифм.

БИБЛИОГРАФИЧЕСКИЕ ССЫЛКИ

1. Алгоритмы: построение и анализ / Т. Кормен [и др.]. — М. : Вильямс, 2005. — 1296 с.

2. *Котов В. М.*, *Мельников О. И.* Информатика. Методы алгоритмизации : учеб. пособие для 10 – 11 кл. общеобразоват. шк. с углубл. изучением информатики. — Минск : Нар. асвета, 2000. — 221 с.

3. *Котов В. М.*, *Соболевская Е. П.*, *Толстиков А. А.* Алгоритмы и структуры данных : учеб. пособие. — Минск : БГУ, 2011. — 267 с. — (Классическое университетское издание).

4. Сборник задач по теории алгоритмов : учеб.-метод. пособие/ В. М. Котов [и др.]. — Минск : БГУ, 2017. — 183 с.

5. Теория алгоритмов : учеб. пособие / П. А. Иржавский [и др.]. — Минск : БГУ, 2013. — 159 с.

6. *Соболь С. А.*, *Котов В. М.*, *Соболевская Е. П.* Опыт использования образовательной платформы Insight Runner на факультете прикладной математики и информатики Белорусского государственного университета // Роль университетского образования и науки в современном обществе : материалы междунар. науч. конф., Минск, 26–27 февр. 2019 г. / Белорус. гос. ун-т ; редкол.: А. Д. Король (пред.) [и др.]. — Минск : БГУ, 2019. — С. 263-267.

7. *Соболь С. А.*, *Котов В. М.*, *Соболевская Е. П.* Методика преподавания дисциплин по теории алгоритмов с использованием образовательной платформы iRunner // Судьбы классического университета: национальный контекст и мировые тренды [Электронный ресурс] : материалы XIII Респ. междисциплинар. науч.-теорет. семинара «Инновационные стратегии в современной социальной философии» и междисциплинар. летней школы молодых ученых «Экология культуры», Минск, 9 апр. 2019 г. / Белорус. гос. ун-т ; сост.: В. В. Анохина, В. С. Сайганова ; редкол.: А. И. Зеленков (отв. ред.) [и др.] — С. 346–355.

СОДЕРЖАНИЕ

Учебное издание

Соболь Сергей Александрович
Вильчевский Константин Юрьевич
Котов Владимир Михайлович и др.

СБОРНИК ЗАДАЧ ПО ТЕОРИИ АЛГОРИТМОВ. СТРУКТУРЫ ДАННЫХ

Учебно-методическое пособие

Редактор *Х. Х. XXXXXXX*
Художник обложки *С. А. Соболь*
Технический редактор *Х. Х. XXXXXXX*
Компьютерная вёрстка *С. А. Соболя*
Корректор *Х. Х. XXXXXX*

Подписано в печать 29.02.2020. Формат 60×84/16. Бумага офсетная.
Печать офсетная. Усл. печ. л. 10,69. Уч.-изд. л. 9,6.
Тираж 150 экз. Заказ

Белорусский государственный университет.
Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий №1/270 от 03.04.2014.
Пр. Независимости, 4, 220030, Минск.

Республиканское унитарное предприятие
«Издательский центр Белорусского государственного университета».
Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий №2/63 от 19.03.2014.
Ул. Красноармейская, 6, 220030, Минск.