Лекція 25

# Хеш-таблиці

В багатьох застосуваннях потрібні динамічні множини, що підтримують лише стандартні словникові операції вставки, пошуку і видалення. Наприклад, компілятор мови програмування підтримує таблицю символів, у якій ключами елементів є довільні символьні рядки, що відповідають ідентифікаторам у мові. Хеш-таблиця являє собою ефективну структуру даних для реалізації словників. Хоча на пошук елемента в хеш-таблиці може в найгіршому випадку знадобитися стільки ж часу, що й у зв'язаному списку, а саме $\Theta(n)$, на практиці хешування винятково ефективне. При цілком обґрунтованих припущеннях математичне очікування часу пошуку елемента в хеш-таблиці складає $O(1)$.

Хеш-таблиця являє собою узагальнення звичайного масиву. Можливість прямої індексації елементів звичайного масиву забезпечує доступ до довільної позиції в масиві за час $O(1)$. Пряма індексація можлива, якщо ми можемо виділити масив розміру, достатнього для того, щоб для кожного можливого значення ключа існувала окрема комірка.

Якщо реальна кількість ключів, що зберігаються в масиві, мала в порівнянні з кількістю можливих значень ключів, ефективною альтернативою масиву з прямою індексацією стає хеш-таблиця, що звичайно використовує масив з розміром, пропорційним кількості ключів, реально зберігаються в ній. Замість безпосереднього використання ключа як індекса масиву , індекс *обчислюється* за значенням ключа. Ми розглянемо основні ідеї хешування, у першу чергу спрямовані на дозвіл колізій (коли декілька ключів відображаються в один індекс масиву) за допомогою ланцюжків. Потім ми розглянемо, яким чином на основі значень ключів можуть бути обчислені індекси масиву. Тут буде розглянуте і проаналізовано декілька варіантів хеш-функцій. Далі ми опишемо метод відкритої адресації, що являє собою ще один спосіб розв'язання колізій. Головний висновок, який випливає з усього викладеного матеріалу: хешування являє собою винятково ефективну і практичну технологію — у середньому всі базові словникове операції вимагають лише $O(1)$ часу. В лекції буде дане пояснення, яким образом “ідеальне хешування” може підтримувати

*найгірший* час пошуку $O(1)$ у випадку використання статичної множини ключів, що зберігаються, (тобто коли множина ключів після запису більше не змінюється).

### 25.1. Таблиці з прямою адресацією

Пряма адресація являє собою найпростішу технологію, що добре працює для невеликих множин ключів. Припустимо, що додатку потрібно динамічна множина, кожен елемент якого має ключ з множини $U = \{0,1,\ldots,m-1\}$, де $m$ не занадто велико. Крім того, передбачається, що ніякі два елементи не мають однакових ключів.

Для представлення динамічної множини ми використовуємо масив, чи ***таблицю з прямою адресацією***, що позначимо як $T[0..m-1]$, кожна ***позиція***, чи ***осередок***, який відповідає ключу з простору ключів *U*. На рис. 25.1 представлено даний підхід. Осередок *k* указує на елемент множини з ключем *k*. Якщо множина не містить елемента з ключем *k*, то $T[k] = \text{NIL}$. На малюнку кожен ключ із простору $U = \{0,1,\ldots,9\}$ відповідає індексу таблиці. Множина реальних ключів $K = \{2,3,5,8\}$ визначає осередку таблиці, що містять вказівники на елементи. Інші осередки (зафарбовані темним кольором) містять значення NIL.

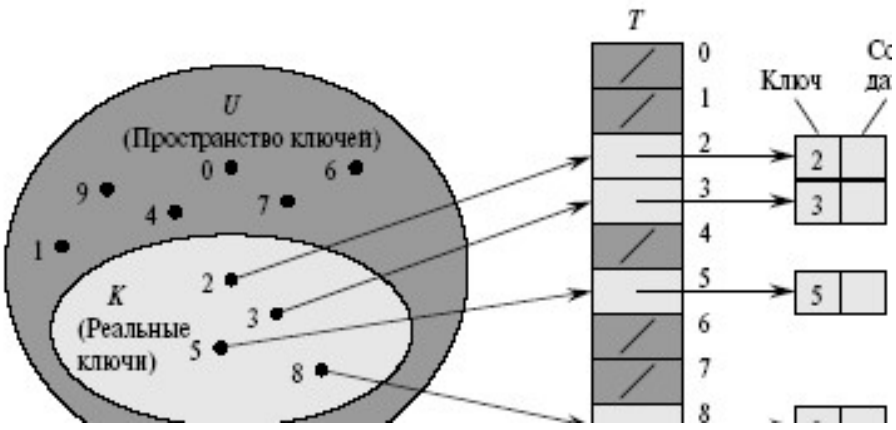


Рис. 25.1. Реалізація динамічної множини з використанням таблиці з прямою адресацією

Реалізація словникових операцій тривіальна:

```
DIRECT_ADDRESS_SEARCH(T,k)
    return T[k]

DIRECT_ADDRESS_INSERT(T,x)
    T[key[x]] ← x
```

```
DIRECT_ADDRESS_DELETE(T,x)
    T[key[x]] ← NIL
```

Кожна з приведених операцій дуже швидка: час їх роботи дорівнює $O(1)$. У деяких застосуваннях елементи динамічної множини можуть зберігатися безпосередньо в таблиці з прямою адресацією. Тобто замість збереження ключів і супутніх даних елементів в об'єктах, зовнішніх стосовно таблиці з прямою адресацією, а в таблиці — вказівників на ці об'єкти, ці об'єкти можна зберігати безпосередньо в осередках таблиці (що тим самим приводить до економії використовуваної пам'яті). Крім того, найчастіше збереження ключа не є необхідною умовою, оскільки якщо ми знаємо індекс об'єкта в таблиці, ми тим самим знаємо і його ключ. Однак якщо ключ не зберігається в осередку таблиці, то нам потрібний якийсь інший механізм для того, щоб позначати порожні осередки.

### 25.2. Хеш-таблиці

Недолік прямої адресації очевидний: якщо простір ключів $U$ великий, збереження таблиці $T$ розміром $|U|$ є непрактичним, а то й зовсім неможливим — у залежності від кількості доступної пам'яті і розміру простору ключів. Крім того, множина $K$ *реально збережених* ключів може бути малою в порівнянні з простором ключів $U$, а в цьому випадку пам'ять, виділена для таблиці $T$, в основному витрачається дарма.

Коли множина $K$ ключів, що зберігаються в словнику, набагато менше *простору* можливих ключів $U$, хеш-таблиця вимагає істотно менше місця, чим таблиця з прямою адресацією. Точніше кажучи, вимоги до пам'яті можуть бути знижені до $\Theta(|K|)$, при цьому час пошуку елемента в хеш-таблиці залишається рівним $O(1)$. Треба лише зауважити, що це границя *середнього часу* пошуку, у той час як у випадку таблиці з прямою адресацією ця границя має місце для *найгіршого випадку*.

У випадку прямої адресації елемент із ключем $k$ зберігається в осередку $k$. При хешуванні цей елемент зберігається в осередку $h(k)$, тобто ми використовуємо ***хеш-функцію*** $h$ для обчислення осередку для даного ключа $k$. Функція $h$ відображає простір ключів $U$ на осередки ***хеш-таблиці*** $T[0..m-1]$:

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

Ми говоримо, що елемент із ключем *k* хешується в осередок $h(k)$; величина $h(k)$ називається ***хеш-значенням*** ключа *k*. На рис. 25.2 представлено основну ідею хешування. Цель хеш-функції полягає в тому, щоб зменшити робочий діапазон індексів масиву, і замість $|U|$ значень ми можемо обійтися усього лише *m* значеннями. Відповідно знижуються і вимоги до кількості пам'яті.

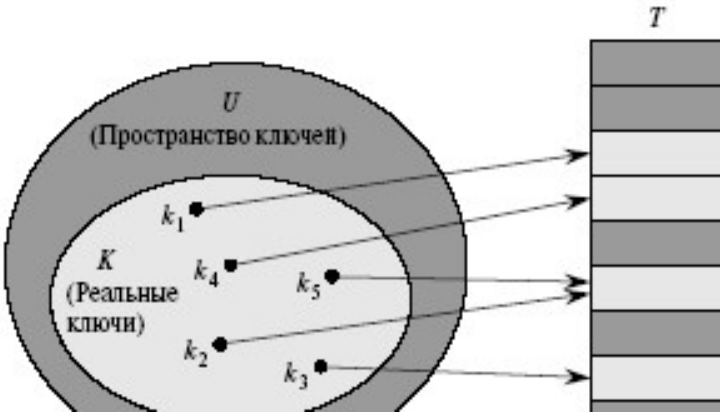


Рис. 25.2. Використання хеш-функції *h* для відображення ключів в осередки хеш-таблиці. Ключі $k_2$ і $k_5$ відображаються в один осередок, викликаючи колізію

Однак тут є одна проблема: два ключі можуть бути хешовані в ту саму осередок. Така ситуація називається ***колізією***. На щастя, існують ефективні технології для розв'зання конфліктів, що спричиняються колізіями.

Звісно, ідеальним рішенням було б повне усунення колізій. Ми можемо спробувати домогтися цього шляхом вибору придатної хеш-функції *h*. Одна з ідей полягає в тім, щоб зробити *h* "випадковою", що дозволило б уникнути колізій чи хоча б мінімізувати їхню кількість (цей характер функції хешування відображається в самім дієслові "to hash", що означає "дрібно порубати, перемішати"). Саме собою зрозуміло, функція *h* повинна бути детерміністичної і для того самого значення *k* завжди давати те саме хеш-значення $h(k)$. Однак оскільки $|U| > m$, повинне існувати як мінімум два ключі, що мають однакове хеш-значення. Таким чином, цілком уникнути колізій неможливо в принципі, і гарна хеш-функція може тільки мінімізувати кількість колізій. Таким чином, нам украй необхідний метод дозволу виникаючих колізій. Ми розглянемо найпростіший метод дозволу колізій — метод ланцюжків. Далі ми познайомимось з ще одним методом розв'язання колізій, що називається методом відкритої адресації.

### Розв'язування колізій за допомогою ланцюжків

При використанні даного методу ми поєднуємо всі елементи, хешовані в один осередок, у зв'язаний список, як показано на рис. 25.3. Осередок *j* містить вказівник на заголовок списку всіх елементів, хеш-значення ключа яких дорівнює *j*; якщо таких елементів немає, осередок містить значення NIL. На рис. 25.3 показано розв'язання колізій, що виникають через те, що $h(k_1) = h(k_4)$, $h(k_5) = h(k_2) = h(k_7)$ і $h(k_8) = h(k_6)$.

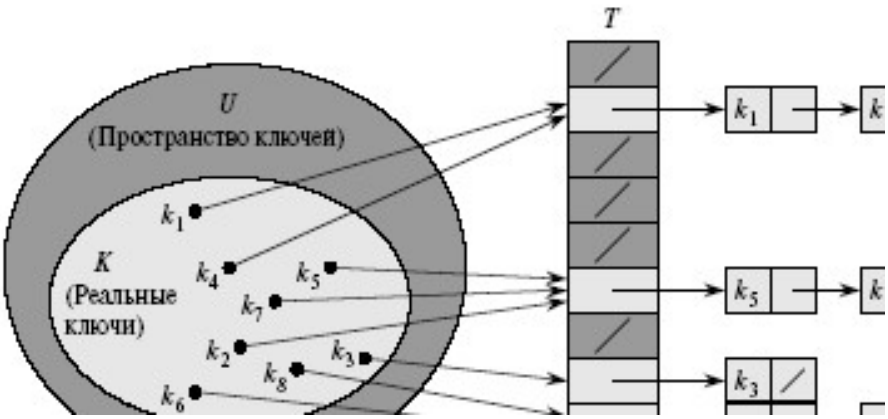


Рис. 25.3. Дозвіл колізій за допомогою ланцюжків

Словникові операції в хеш-таблиці з використанням ланцюжків для дозволу колізій реалізуються дуже просто:

```
CHAINED_HASH_INSERT(T,x)
    Уставити x у заголовок списку T[h(key[x])]

CHAINED_HASH_SEARCH(T,k)
    Пошук елемента з ключем k у списку T[h(k)]

CHAINED_HASH_DELETE(T,x)
    Видалення x зі списку T[h(key[x])]
```

Час, необхідний для вставки в найгіршому випадку, дорівнює $O(1)$. Процедура вставки виконується дуже швидко, оскільки передбачається, що елемент, що вставляється, відсутній у таблиці. При необхідності це припущення може бути перевірене шляхом виконання пошуку перед вставкою. Час роботи пошуку в найгіршому випадку пропорціональний довжині списку; ми проаналізуємо цю операцію трохи пізніше. Видалення елемента може бути виконане за час $O(1)$ при використанні двозв'язних списків. (Зверніть увагу на те, що

процедура CHAINED_HASH_DELETE приймає як аргумент елемент *x*, а не його ключ, тому немає необхідності в попередньому пошуку *x*. Якщо список однозв'язний, то передача як аргумент *x* не дає нам особливого виграшу, оскільки для коректного відновлення поля *next* попередника *x* нам все одно треба виконати пошук *x* у списку $T\left[h\left(key[x]\right)\right]$. У такому випадку, як неважко зрозуміти, видалення і пошук мають по суті однаковий час роботи.)

### Аналіз хешування з ланцюжками

Наскільки висока продуктивність хешування з ланцюжками? Зокрема, скільки часу потрібно для пошуку елемента з даним ключем?

Нехай у нас є хеш-таблиця *T* з *m* осередками, у яких зберігаються *n* елементів. Визначимо ***коефіцієнт заповнення*** таблиці *T* як $\alpha = n/m$, тобто середню кількість елементів, що зберігаються в одному ланцюжку. Наш аналіз буде спиратися на значення величини $\alpha$, який може бути менше, дорівнює чи більше одиниці.

У найгіршому випадку хешування з ланцюжками поводиться вкрай неприємно: усі *n* ключів хешовані в ту саму осередок, створюючи список довжиною *n*. Таким чином, час пошуку в найгіршому випадку дорівнює $\Theta\left(n\right)$ плюс час обчислення хеш-функції, що нітрохи не краще, ніж у випадку використання зв'язаного списку для збереження всіх *n* елементів. Зрозуміло, що використання хеш-таблиць у найгіршому випадку зовсім безглуздо. (Ідеальне хешування (застосовне у випадку статичної множини ключів), що буде розглянуто в розділі 25.5, забезпечує високу продуктивність навіть у найгіршому випадку.)

Середня ефективність хешування залежить від того, наскільки ефективно хеш-функція *h* розподіляє множину ключів, що зберігаються, *по m* осередках у середньому. Ми розглянемо це питання докладніше пізніше, а пока будемо вважати, що всі елементи хешуються по осередках рівномірно і незалежно, і назвемо дане припущення "***простим рівномірним хешуванням***".

Позначимо довжини списків $T\left[j\right]$ для $j = 0,1,\ldots,m-1$ як $n_j$, так що

$$n = n_0 + n_1 + \ldots + n_{m-1}, \tag{1}$$

а середнє значення $n_j$ дорівнює $\mathrm{E}\left[n_j\right] = \alpha = n/m$ .

Ми вважаємо, що хеш-значення $h\left(k\right)$ може бути обчислене за час $O\left(1\right)$, так що час, необхідний для пошуку елемента з ключем *k*, лінійно залежить від довжини $n_{h(k)}$ списку

$T\left[h(k)\right]$. З огляду на час $O(1)$, що вимагається для обчислення хеш-функції і доступу до осередку $h(k)$, розглянемо математичне очікування кількості елементів, що повинне бути перевірено алгоритмом пошуку (тобто кількість елементів у списку $T\left[h(k)\right]$, що перевіряються на рівність ключів величині *k*). Ми повинні розглянути два випадки: по-перше, коли пошук невдалий і в таблиці немає елементів із ключем *k*, і, по-друге, коли пошук закінчується успішно й у таблиці визначається елемент із ключем *k*.

**Теорема 25.1.**

У хеш-таблиці з розв'язуванням колізій методом ланцюжків математичне очікування часу невдалого пошуку в припущенні простого рівномірного хешування дорівнює $\Theta(1+\alpha)$.

***Доведення***. У припущенні простого рівномірного хешування будь-який ключ *k*, що ще не знаходиться в таблиці, може бути поміщений з рівною імовірністю в кожний з *m* осередків. Математичне очікування часу невдалого пошуку ключа *k* дорівнює часу пошуку до кінця списку $T\left[h(k)\right]$, математичне очікування довжини якого $\mathrm{E}\left[n_{h(k)}\right]=\alpha$. Таким чином, при невдалому пошуку математичне очікування кількості елементів, що перевіряються, $\alpha$, , а загальний час, необхідний для пошуку, включаючи час обчислення $h(k)$ хеш-функції, $\Theta(1+\alpha)$. ■

Успішний пошук дещо відрізняється від невдалого, оскільки імовірність пошуку в списку різна для різних списків і пропорційна кількості елементів, що містяться в ньому. Тем, і в цьому випадку математичне очікування часу пошуку залишається рівним $\Theta(1+\alpha)$.

**Теорема 25.2**

У хеш-таблиці з розв'язуванням колізій методом ланцюжків математичне очікування часу успішного пошуку в припущенні простого рівномірного хешування в середньому дорівнює $\Theta(1+\alpha)$.

***Доведення***. Ми вважаємо, що шуканий елемент із рівною імовірністю може бути будь-яким елементом, що зберігається в таблиці. Кількість елементів, що перевіряються в процесі успішного пошуку елемента *x*, на 1 більше, ніж кількість елементів, що знаходяться в списку перед *x*. Елементи, що знаходяться в списку до *x*, були вставлені в список після того, як елемент *x* був збережений у таблиці, тому що нові елементи містяться в початок списку. Для

того щоб знайти математичне очікування кількості елементів, що перевіряються, ми візьмемо середнє значення по всіх елементах таблиці, що дорівнює 1 плюс математичне очікування кількості елементів, доданих у список після шуканого. Нехай $x_i$ позначає *i-й* елемент, вставлений у таблицю ($i = 1,2,\ldots,n$ ), а $k_i = key\left[x_i\right]$. Визначимо для ключів $k_i$ і $k_j$ індикаторну випадкову величину $X_{ij} = \mathrm{I}\left\{h\left(k_i\right) = h\left(k_j\right)\right\}$. У припущенні простого рівномірного хешування, $\Pr\left\{h\left(k_i\right) = h\left(k_j\right)\right\} = 1/m$ і $\mathrm{E}\left[X_{ij}\right] = 1/m$. Таким чином, математичне очікування числа елементів, що перевіряються, у випадку успішного пошуку

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

$$= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\mathrm{E}\left[X_{ij}\right]\right) =$$ (у силу лінійності математичного очікування)

$$= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right) =$$

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

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

$$= 1+\frac{1}{nm}\left(n^2-\frac{n\left(n+1\right)}{2}\right) =$$ (відповідно до (A.1))

$$= 1+\frac{n-1}{m} =$$

$$= 1+\frac{\alpha}{2}-\frac{\alpha}{2n}.$$

Отже, повний час, необхідний для проведення успішного пошуку (включаючи час на обчислення хеш-функції), складає $\Theta\left(2+\alpha/2-\alpha/2n\right) = \Theta\left(1+\alpha\right)$. ■

Що ж означає проведений аналіз? Якщо кількість осередків у хеш-таблиці, як мінімум, пропорционально кількості елементів, що зберігаються в ній, то $n = O\left(m\right)$ і, отже, $\alpha = n/m = O\left(m\right)/m = O\left(1\right)$, а виходить, пошук елемента в хеш-таблиці в середньому

вимагає постійного часу. Оскільки в гіршому випадку вставка елемента в хеш-таблицю займає $O(1)$ часу (як і видалення елемента при використанні двохв'язних списків), можна зробити висновок, що всі словникові операції в хеш-таблиці в середньому виконуються за час $O(1)$.

### 25.3. Хеш-функції

У цьому розділі ми розглянемо деякі питання, зв'язані з розробкою якісних хеш-функцій, і познайомимося з трьома схемами їх побудови. Дві з них, хешування розподілом і хешування множенням, є евристичними за своєю природою, у той час як третя схема — універсальне хешування — використовує рандомізацію.

#### У чому полягає якість хеш-функції

Якісна хеш-функція задовольняє (наближено) припущення простого рівномірного хешування: для кожного ключа рівноймовірно розміщення в кожному з $m$ осередків, незалежно від хешування інших ключів. На жаль, цю умову зазвичай неможливо перевірити, оскільки, як правило, розподіл імовірностей, відповідно до якого надходять внесені в таблицю ключі, невідомо; крім того, що вставляються ключі можуть не бути незалежними.

Іноді розподіл імовірностей виявляється відомим. Наприклад, якщо відомо, що ключі являють собою випадкові дійсні числа, рівномірно розподілені в діапазоні $0 \le k < 1$, то хеш-функція $h(k) = \lfloor km \rfloor$ задовольняє умові простого рівномірного хешування.

На практиці при побудові якісних хеш-функцій найчастіше використовуються різні евристичні методики. У процесі побудови дуже корисною є інформація про розподіл ключів. Розглянемо, наприклад, таблицю символів компілятора, у якій ключами служать символьні рядки, що представляють ідентифікатори в програмі. Найчастіше в одній програмі зустрічаються схожі ідентифікатори, наприклад, `pt` і `pts`. Гарна хеш-функція повинна мінімізувати шанси влучення цих ідентифікаторів в один осередок хеш-таблиці.

При побудові хеш-функції гарним підходом є підбор функції таким чином, щоб вона ніяк не корелювала з закономірностями, яким можуть підкорятися існуючі дані. Наприклад, метод розподілу, що обчислює хеш-значення як залишок від розподілу ключа на деяке просте число. Якщо це просте число ніяк не зв'язане з розподілом вихідних даних, метод часто дає гарні результати.

На закінчення помітимо, що деякі додатки хеш-функцій можуть накладати додаткові вимоги, крім вимог простого рівномірного хешування. Наприклад, ми можемо зажадати, щоб “близькі” у деякому змісті ключі давали далекі хеш-значения (ця властивість особлива бажано при використанні лінійного дослідження). Універсальне хешування найчастіше приводить до бажаних результатів.

#### Інтерпретація ключів як цілих невід'ємних чисел

Для більшості хеш-функцій простір ключів представляється множиною цілих невід’ємних чисел $\mathbf{N} = \{0,1,2,\ldots\}$. Якщо ж ключі не є цілими невід’ємними числами, то можна знайти спосіб їх інтерпретації як таких. Наприклад, рядок символів може розглядатися як ціле число, записане у відповідній системі числення. Так, ідентифікатор `pt` можна розглядати як пари десяткових чисел (112, 116), оскільки в ASCII-наборі символів `p` = 112 і `t` = 116. Розглядаючи `pt` як число в системі числення з базою 128, ми знаходимо, що воно відповідає значенню $112 \cdot 128 + 116 = 14452$. У конкретних застосуваннях нескладно розробити метод для представлення ключів у вигляді (можливо, великих) цілих чисел. Далі при викладі матеріалу ми будемо вважати, що всі ключі представляють цілі невід’ємні числа.

### 25.3.1. Метод ділення

Побудова хеш-функції ***методом ділення*** складається у відображенні ключа *k* в один з осередків шляхом одержання залишку від ділення *k* на *m*, тобто хеш-функція має вид $h(k) = k \bmod m$.

Наприклад, якщо хеш-таблиця має розмір $m = 12$, а значення ключа $k = 100$, то $h(k) = 4$. Оскільки для обчислення хеш-функції потрібно тільки одна операція ділення, хешування методом розподілу вважається досить швидким.

При використанні даного методу ми звичайно намагаємося уникати деяких значень *m*. Наприклад, *m* не повинно бути ступенем 2, оскільки якщо $m = 2^p$, то $h(k)$ являє собою просто *p* молодших битів числа *k*. Якщо заздалегідь не відомо, що всі набори молодших *p* бітів ключів рівноймовірні, краще будувати хеш-функцію таким чином, щоб її результат залежав від усіх бітів ключа.

Найчастіше гарні результати можна одержати, вибираючи як значення *m* просте число, достатньо далеке від ступеня двійки. Припустимо, наприклад, що ми хочемо створити хеш-

таблицю з розв'язуванням колізій методом ланцюжків для збереження $n = 2000$ символьних рядків, розмір символів у який дорівнює 8 бітам. Нас влаштовує перевірка в середньому трьох елементів при невдалому пошуку, так що ми вибираємо розмір таблиці рівним $m = 701$. Число 701 обране як просте число, близьке до величини $2000/3$ і не є ступенем 2. Розглядаючи кожен ключ *k* як ціле число, ми одержуємо шукану хеш-функцію:

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

### 25.3.2. Метод множення

Побудова хеш-функції ***методом множення*** виконується в два етапи. Спочатку ми множимо ключ *k* на константу $0 < A < 1$ й одержуємо дробову частину отриманого добутку. Потім ми множимо отримане значення на *m* і застосовуємо до нього функцію заокруглення вниз тобто

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

де вираз "$kA \bmod 1$ " означає одержання дробової частини добутку *k*, тобто величину $kA - \lfloor kA \rfloor$.

Перевага методу множення полягає в тому, що значення *m* перестає бути критичним. Як правило, величина *m* з розумінь зручності реалізації функції вибирається рівною ступеню 2. Нехай у нас є комп'ютер з розміром слова *w* бітів і *k* міститься в одному слові. Обмежимо можливі значення константи *A* видом $s/2^w$, де *s* — ціле число з діапазону $0 < s < 2^w$. Тоді ми спочатку множимо *k* на *w-бітове* ціле число $s = A \cdot 2^w$. Результат являє собою 2*w-бітове* число $r_1 2^w + r_0$, де $r_1$ — старше слово добутку, а $r_0$ — молодше. Старші *p* бітів числа $r_0$ являють собою шукане *p-бітове* хеш-значення (рис. 25.4)

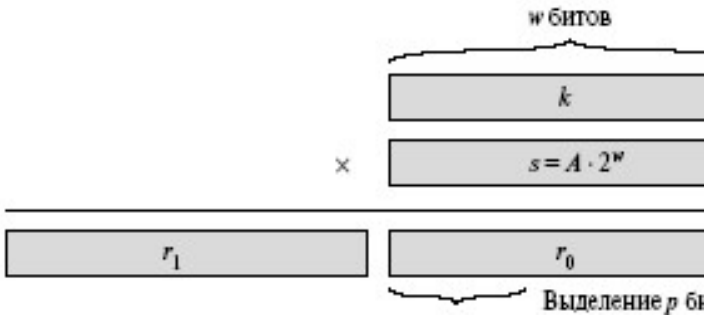


Рис. 25.4. Хешування методом множення

Хоча описаний метод працює з будь-якими значеннями константи *A*, деякі значення дають кращі результати в порівнянні з іншими. Оптимальний вибір залежить від характеристик хешируемых даних. Наприклад. Дональд Кнут запропонував використовувати значення, що дає непогані результати

$$A \approx \left(\sqrt{5}-1\right)/2 \approx 0.6180339887\ldots \tag{2}$$

Візьмемо як приклад $k = 123456$, $p = 14$, $m = 2^{14} = 16384$ і $w = 32$. Приймаючи пропозицію Кнута, вибираємо значення $A$ у вигляді $s/2^w$, найближче до величини $\left(\sqrt{5}-1\right)/2$, так що $A = 2654435769/2^{32}$. Тоді

$$k \cdot s = 327706022297664 = \left(76300 \cdot 2^{32}\right) + 17612864 ,$$

і, відповідно, $r_1 = 76300$ і $r_0 = 17612864$. Старші 14 бітів числа $r_0$ дають нам хеш-значення $h(k) = 67$.

### 25.3.3. Універсальне хешування

Якщо зловмисник буде навмисне вибирати ключі для хешування за допомогою конкретної хеш-функції, то він зможе підібрати *n* значень, що будуть хешуватися в той самий осередок таблиці, приводячи до середнього часу вибірки $\Theta(n)$. Таким чином, будь-яка фіксована хеш-функція стає уразливої, і єдиний ефективний вихід із ситуації — *випадковий* вибір хеш-функції, що *не залежить* від того, з якими саме ключами їй має бути працювати. Такий підхід, що називається ***універсальним хешуванням***, гарантує гарну продуктивність у середньому, незалежно від того, які дані будуть обрані зловмисником.

Головна ідея універсального хешування складається у випадковому виборі хеш-функції з деякого ретельно відібраного класу функцій на початку роботи програми. Як і у випадку швидкого сортування, рандомізація гарантує, що ті самі вхідні дані не можуть постійно давати найгірше поводження алгоритму. У силу рандомізації алгоритм буде працювати всякий раз по-різному, навіть для тих самих вхідних даних, що гарантує високу середню ефективність для будь-яких вхідних даних. Повертаючи до приклада з таблицею символів компілятора, ми знайдемо, що ніякий вибір програмістом імен ідентифікаторів не може привести до постійного зниження ефективності хешування. Таке зниження можливе лише тоді, коли компілятором обрана випадкова хеш-функція, що приводить до поганого

хешування конкретних вхідних даних; однак імовірність такої ситуації дуже мала й однакова для будь-якої множини ідентифікаторів одного і те ж розміру.

Нехай $\mathcal{H}$ — кінцева множина хеш-функцій, що відображають простір ключів $U$ у діапазон $\{0,1,2,\ldots,m-1\}$. Така множина називається ***універсальною***, якщо для кожної пари різних ключів $k,l \in U$ кількість хеш-функцій $h \in \mathcal{H}$, для яких $h(k) = h(l)$, не перевищує $|\mathcal{H}|/m$. Іншими словами, при випадковому виборі хеш-функції з $\mathcal{H}$ імовірність колізії між різними ключами $k$ і $l$ не перевищує імовірності збігу двох випадковим образом обраних хеш-значень з множини $\{0,1,2,\ldots,m-1\}$, що дорівнює $1/m$. Наступна теорема показує, що універсальні хеш-функції забезпечують гарну середню ефективність. У приведеній теоремі $n_i$, як уже згадувалося, позначає довжину списку $T[i]$.

**Теорема 25.3.**

Нехай хеш-функція $h$, обрана з універсальної множини хеш-функцій, використовується для хешування $n$ ключів у таблицю $T$ розміру $m$, з використанням для розв'язання колізій методу ланцюжків. Якщо ключ $k$ відсутній у таблиці, то математичне очікування $\mathrm{E}\left[n_{h(k)}\right]$ довжини списку, у який хешується ключ $k$, не перевищує $\alpha$. Якщо ключ $k$ знаходиться в таблиці, то математичне очікування $\mathrm{E}\left[n_{h(k)}\right]$ довжини списку, у якому знаходиться ключ $k$, не перевищує $1+\alpha$.

***Доведення***. Помітимо, що математичне очікування обчислюється на множини функцій вибору і не залежить від яких би то ни було припущень про розподіл ключів. Визначимо для кожної пари різних ключів $k$ і $l$ індикаторну випадкову величину $X_{kl} = \mathrm{I}\{h(k) = h(l)\}$. Оскільки за визначенням пари ключів викликає колізію з імовірністю не вище $1/m$, одержуємо, що $\Pr\{h(k) = h(l)\} \le 1/m$, так що, відповідно до леми 5.1, $\mathrm{E}[X_{kl}] \le 1/m$.

Далі для кожного ключа k визначимо випадкову величину $Y_k$, що дорівнює кількості ключів, що відрізняються від $k$ і хешируемых у той же осередок, що і ключ $k$:

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

Відповідно, одержуємо:

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

$$= \sum_{\substack{l\in T\\ l\neq k}} E\left[X_{kl}\right] \leq \qquad \text{(у силу лінійності математичного очікування)}$$

$$\leq \sum_{\substack{l\in T\\ l\neq k}} \frac{1}{m}.$$

Частина доказу, що залишилася, залежить від того, ли знаходиться ключ $k$ у таблиці $T$.

- Якщо $k \notin T$, то $n_{h(k)} = Y_k$ і $\left|\left\{l : l \in T \text{ и } l \neq k\right\}\right| = n$. Відповідно, $\mathrm{E}\left[n_{h(k)}\right] = \mathrm{E}\left[Y_k\right] \leq n/m = \alpha$.
- Якщо $k \in T$, то оскільки $k$ знаходиться в списку $T\left[h\left(k\right)\right]$ і значення $Y_k$ не включає ключ $k$, ми маємо $n_{h(k)} = Y_k + 1$ і $\left|\left\{l : l \in T \text{ и } l \neq k\right\}\right| = n - 1$. Таким чином, $\mathrm{E}\left[n_{h(k)}\right] = \mathrm{E}\left[Y_k\right] + 1 \leq \left(n-1\right)/m + 1 = 1 + \alpha - 1/m < 1 + \alpha$. ■

Наслідок з даної теореми говорить, що універсальне хешування забезпечує бажаний виграш: тепер неможливо вибрати послідовність операцій, що приведуть до найгіршого часу роботи. Шляхом рандомізації вибору хеш-функції в процесі роботи програми гарантується гарний середній час роботи алгоритму для будь-яких вхідних даних.

**Теорема 25.4.**

Використання універсального хешування і розв'язання колізій методом ланцюжків у хеш-таблиці з $m$ осередками дає математичне очікування часу виконання будь-якої послідовності з $n$ вставок, пошуків і видалень, у якій міститься $O\left(m\right)$ вставок, рівне $\Theta\left(n\right)$.

***Доведення***. Оскільки кількість уставок дорівнює $O\left(m\right)$, $n = O\left(m\right)$ і, відповідно, $\alpha = O\left(1\right)$. Час роботи операцій вставки і видалення — величина постійна, а відповідно до теореми 3 математичне очікування часу виконання кожної операції пошуку дорівнює $O\left(1\right)$. Таким чином, використовуючи властивість лінійності математичного очікування, одержуємо,

що очікуваний час, необхідне для виконання всієї послідовності операцій, дорівнює $O(n)$. Оскільки кожна операція забирає час $\Omega(1)$, звідси випливає границя $\Theta(n)$. ■

### Побудова універсальної множини хеш-функцій

Побудувати таку множина задоволена просто, що випливає з теорії чисел. Почнемо з вибору простого числа *p*, досить великого, щоб усі можливі ключі знаходилися в діапазоні від 0 до $p-1$ включно. Нехай $\mathbf{Z}_p$ позначає множина $\{0,1,\ldots,p-1\}$, а $\mathbf{Z}_p^*$ — множина $\{1,2,\ldots,p-1\}$. Оскільки *p* — простої число, ми можемо вирішувати рівняння по модулю *p* за допомогою методів, описаних у главі 31. З припущення про те, що простір ключів більше, ніж кількість осередків у хеш-таблиці, випливає, що $p > m$.

Тепер визначимо хеш-функцію $h_{a,b}$ для будь-яких $a \in \mathbf{Z}_p^*$ і $b \in \mathbf{Z}_p$ в такий спосіб:

$$h_{a,b}(k) = \big((ak+b) \bmod p\big) \bmod m. \tag{3}$$

Наприклад, при $p = 17$ і $m = 6$ $h_{3,4}(8) = 5$. Сімейство всіх таких функцій утворить множину

$$\mathcal{H}_{p,m} = \{h_{a,b} : a \in \mathbf{Z}_p^* \text{ и } b \in \mathbf{Z}_p\}. \tag{4}$$

Кожна хеш-функція $h_{a,b}$ відображає $\mathbf{Z}_p$ на $\mathbf{Z}_m$. Цей клас хеш-функцій має таку властивість, що розмір *m* вихідного діапазону довільний і не обов'язково являє собою просте число. Цю властивість буде використано далі. Оскільки число *a* можна вибрати $p-1$ способом, і *p* способами — число *b*, усього в множини $\mathcal{H}_{p,m}$ міститься $p(p-1)$ хеш-функцій.

### Теорема 25.5.

Множина хеш-функцій $\mathcal{H}_{p,m}$, обумовлене рівняннями (3) і (4), є універсальним.

***Доведення***. Розглянемо два різних ключі *k* і *l* з $\mathbf{Z}_p$, тобто $k \neq l$. Нехай для даної хеш-функції $h_{a,b}$

$$r = (ak+b) \bmod p,$$
$$s = (al+b) \bmod p.$$

Помітимо, що $r \neq s$. Чому? Розглянемо різницю

$$r - s \equiv a(k - l) \pmod{p} .$$

Оскільки $p$ — просте число, причому як $a$, так і $(k - l)$ не дорівнюють нулю по модулю $p$, то звідси випливає що $r \neq s$, так що їх добуток також повинний бути відмінним від нуля по модулю $p$. Отже, обчислення будь-якої хеш-функції $h_{a,b} \in \mathcal{H}_{p,m}$ для різних ключів $k$ і $l$ приводить до різних хеш-значень $r$ і $s$ по модулю $p$. Таким чином, колізії "по модулю $p$" відсутні. Більш того, кожна з $p(p-1)$ можливих пар $(a,b)$, у яких $a \neq 0$, приводять до *різних* пар $(r,s)$, у яких $r \neq s$. Щоб переконатися в цьому, досить розглянути можливість однозначного визначення $a$ і $b$ за данним $r$ і $s$:

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

де $\left((k - l)^{-1} \bmod p\right)$ позначає єдине мультиплікативне обернене по модулю $p$ значення $k - l$. Оскільки існує тільки $p(p-1)$ можливих пар $(r,s)$, таких що $r \neq s$, то існує взаємно-однозначна відповідність між парами $(a,b)$, де $a \neq 0$, і парами $(r,s)$, у яких $r \neq s$. Таким чином, для будь-якої даної пари вхідних значень $k$ і $l$ при рівномірному випадковому виборі пари $(a,b)$ з $\mathbf{Z}_p^* \times \mathbf{Z}_p$, одержувана в результаті пари $(r,s)$ може бути з рівною імовірністю кожної з пар зі значеннями, що відрізняються, по модулю $p$.

Звідси можна укласти, що імовірність того, що різні ключі $k$ і $l$ приводять до колізії, дорівнює імовірності того, що $r \equiv s \pmod{m}$ при довільному виборі отличающихся по модулю $p$ значень $r$ і $s$. Для даного значення $r$ мається $p - 1$ можливе значення $s$. При цьому число значень $s$, таких що $s \neq r$ і $s \equiv r \pmod{p}$, не перевищує

$$\lceil p/m \rceil - 1 \leq \left((p + m - 1)/m\right) - 1 = (p - 1)/m .$$

Імовірність того, що $s$ приводить до колізії з $r$ при приведенні по модулю $m$, не перевищує

$$\left((p - 1)/m\right)/(p - 1) = 1/m .$$

Отже, для будь-якої пари різних значень $k, l \in \mathbf{Z}_p$

$$\Pr\{h_{a,b}(k) = h_{a,b}(l)\} \leq 1/m ,$$

так що множина хеш-функцій $\mathcal{H}_{p,m}$ є універсальною. ■

### 25.4. Відкрита адресація

При використанні методу ***відкритої адресації*** всі елементи зберігаються безпосередньо в хеш-таблиці, тобто кожен запис таблиці містить або елемент динамічної множини, або значення NIL. При пошуку елемента ми систематично перевіряємо осередки таблиці доти, поки не знайдемо шуканий чи елемент поки не переконаємося в його відсутності в таблиці. Тут, на відміну від методу ланцюжків, немає ані списків, ані елементів, що зберігаються поза таблицею. Таким чином, у методі відкритої адресації хеш-таблиця може виявитися заповненої, роблячи неможливої вставку нових елементів; коефіцієнт заповнення $\alpha$ не може перевищувати 1.

Звісно, при хешуванні з розв'язанням колізій методом ланцюжків можна використовувати вільні місця в хеш-таблиці для збереження зв'язаних списків, але перевага відкритої адресації полягає в тім, що вона дозволяє цілком відмовитися від вказівників. Замість того щоб випливати по вказівниках, ми *обчислюємо* послідовність осередків, що перевіряються. Додаткова пам'ять, що звільняється в результаті відмовлення від вказівників, дозволяє використовувати хеш-таблиці більшого розміру при том же загальній кількості пам'яті, потенційно приводячи до меншої кількості колізій і більш швидкого вибору.

Для виконання вставки при відкритій адресації ми послідовно перевіряємо, чи ***досліджуємо***, осередку хеш-таблиці доти, поки не знаходимо порожній осередок, у яку розташовуємо ключ, що вставляється. Замість фіксованого порядку дослідження осередків 0, 1, …, $m-1$ (для чого потрібно $\Theta(n)$ часу), послідовність досліджуваних осередків *залежить від ключа, що* вставляється в таблицю. Для визначення досліджуваних осередків ми розширимо хеш-функцію, включивши в неї як другий аргумент номер дослідження (що починається з 0). У результаті хеш-функція стає наступною:

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

У методі відкритої адресації потрібно, щоб для кожного ключа $k$ ***послідовність досліджень*** $\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$ являла собою перестановку множини $\langle 0,1,\ldots,m-1 \rangle$, щоб у кінцевому рахунку могли бути переглянуті всі осередки хеш-таблиці. У приведеному далі псевдокоді передбачається, що елементи в таблиці $T$ являють собою ключі без супутньої інформації; ключ $k$ тотожній елементу, що містить ключ $k$. Кожен осередок містить або ключ, або значення NIL (якщо вона не заповнена):

```
HASH_INSERT(T, k)
 1  i ← 0
```

```
 2  repeat j ← h(k,i)
 3      if T[j] = NIL
 4          then T[j] ← k
 5               return j
 6          else i ← i + 1
 7    until i = m
 8  error "Хеш-таблиця переповнена"
```

Алгоритм пошуку ключа *k* досліджує ту ж послідовність осередків, що й алгоритм вставки ключа *k*. Таким чином, якщо при пошуку зустрічається порожній осередок, пошук завершується невдало, оскільки ключ *k* повинний був би бути вставлений у цей осередок у послідовності досліджень, і ніяк не пізніше її. (Ми припускаємо, що видалень з хеш-таблиці не було.) Процедура HASH_SEARCH одержує як вхідні параметри хеш-таблицю *T* і ключ *k* і повертає номер осередку, що містить ключ *k* (чи значення NIL, якщо ключ у хеш-таблиці не виявлений):

```
HASH_SEARCH(T,k)
 1  i ← 0
 2  repeat j ← h(k,i)
 3      if T[j] = k
 4          then return j
 5      i ← i + 1
 6    until T[j] = NIL or i = m
 7  return NIL
```

Процедура видалення з хеш-таблиці з відкритою адресацією досить складна. При видаленні ключа з осередку *i* ми не можемо просто позначити її значенням NIL. Надійшовши так, ми можемо унеможливити вибірку ключа *k*, у процесі вставки якого досліджувалася і виявився зайнятої осередок *i*. Одне з рішень полягає в тому, щоб позначати такі осередки спеціальним значенням DELETED замість NIL. При цьому ми повинні злегка змінити процедуру HASH_INSERT, для того щоб вона розглядала такий осередок як порожню і могла вставити в неї новий ключ. У процедурі HASH_SEARCH ніякі зміни не вимагаються, оскільки ми просто пропускаємо такі осередки при пошуку і досліджуємо наступні осередки в послідовності. Однак при використанні спеціального значення DELETED час пошуку перестає залежати від коефіцієнта заповнення $\alpha$, і з цієї причини, як правило, при необхідності видалень з хеш-таблиці як метод дозволу колізій вибирається метод ланцюжків.

У нашому подальшому аналізі ми будемо виходити з припущення ***рівномірного хешування***, тобто ми припускаємо, що для кожного ключа як послідовність досліджень рівноймовірні усі $m!$ перестановок множини $\{0,1,\ldots,m-1\}$. Рівномірне хешування являє собою узагальнення визначеного раніше простого рівномірного хешування, що полягає в тім, що тепер хеш-функція дає не одне значення, а ціла послідовність досліджень. Реалізація істинно рівномірного хешування досить важка, однак на практиці використовуються придатні апроксимації (такі, наприклад, як визначене нижче подвійне хешування).

Для обчислення послідовності досліджень для відкритої адресації звичайно використовуються три методи: лінійне дослідження, квадратичне дослідження і подвійне хешування. Ці методи гарантують, що для кожного ключа $k$ $\langle h(k,0),h(k,1),\ldots,h(k,m-1)\rangle$ є перестановкою $\langle 0,1,\ldots,m-1\rangle$. Однак ці методи не задовольняють припущенню про рівномірний хешуванні, тому що жодний з них не в змозі згенерувати більш $m^2$ різних послідовностей досліджень (замість $m!$, що вимагаються для рівномірного хешування). Найбільша кількість послідовностей досліджень дає подвійне хешування і, як і випливало очікувати, дає найкращі результати.

### Лінійне дослідження

Нехай задана звичайна хеш-функція $h':U\to\{0,1,\ldots,m-1\}$, що ми будемо надалі іменувати ***допоміжною хеш-функцією***. Метод ***лінійного дослідження*** для обчислення послідовності досліджень використовує хеш-функцію

$$h(k,i)=\left(h'(k)+i\right)\bmod m\ ,$$

де $i$ приймає значення від 0 до $m-1$ включно. Для даного ключа $k$ першим досліджуваним осередком є $T\left[h'(k)\right]$, тобто осередок, що дає допоміжна хеш-функція. Далі ми досліджуємо осередок $T\left[h'(k)+1\right]$ і далі послідовно усі до осередку $T\left[m-1\right]$, після чого переходимо в початок таблиці і послідовно досліджуємо осередку $T\left[0\right]$, $T\left[1\right]$, і так до осередку $T\left[h'(k)-1\right]$. Оскільки початковий досліджуваний осередок однозначно визначає всю послідовність досліджень цілком, усього мається $m$ різних послідовностей.

Лінійне дослідження легке реалізується, однак з ним зв'язана проблема ***первинної кластеризації***, зв'язаної зі створенням довгих послідовностей зайнятих осередків, що, саме собою зрозуміло, збільшує середній час пошуку. Кластери виникають у зв'язку з тим, що

імовірність заповнення порожнього осередку, який передують *i* заповнених осередків, дорівнює $(i+1)/m$. Таким чином, довгі серії заповнених осередків мають тенденцію до усе більшого подовження, що приводить до збільшення середнього часу пошуку.

### Квадратичне дослідження

***Квадратичне дослідження*** використовує хеш-функцію виду

$$h(k,i) = \left(h'(k) + c_1 i + c_2 i^2\right) \bmod m, \tag{5}$$

де $h'$ — допоміжна хеш-функція, $c_1$ і $c_2 \neq 0$ — допоміжні константи, а *i* приймає значення від 0 до $m-1$ включно. Початковий досліджуваний осередок — $T\left[h'(k)\right]$; інші досліджувані позиції зміщені щодо її на величини, що описуються квадратичною залежністю від номера дослідження *i*. Цей метод працює істотно краще лінійного дослідження, але для того, щоб дослідження охоплювало всі осередки, необхідний вибір спеціальних значень $c_1$, $c_2$ і *m*. Крім того, якщо два ключі мають одну і те ж початкову позицію дослідження, то однакові і послідовності дослідження в цілому, тому що з $h_1(k,0) = h_2(k,0)$ випливає $h_1(k,i) = h_2(k,i)$. Ця властивість приводить до більш м'якої ***вторинної кластеризації***. Як і у випадку лінійного дослідження, початковий осередок визначає всю послідовність, тому усього використовується *m* різних послідовностей дослідження.

### Подвійне хешування

Подвійне хешування являє собою один з найкращих способів використання відкритої адресації, оскільки одержувані при цьому перестановки мають багатьма характеристики випадково обираних перестановок. ***Подвійне хешування*** використовує хеш-функцію виду

$$h(k,i) = \left(h_1(k) + i h_2(k)\right) \bmod m,$$

де $h_1$ і $h_2$ — допоміжні хеш-функції. Початкове дослідження виконується в позиції $T\left[h_1(k)\right]$, а зсув кожної з наступних досліджуваних осередків щодо попередньої дорівнює $h_2(k)$ по модулю *m*. Отже, на відміну від лінійного і квадратичного дослідження, у даному випадку послідовність дослідження залежить від ключа *k* по двох параметрах — у плані вибору початкового досліджуваного осередку і відстані між сусідніми досліджуваними осередками, тому що обоє ці параметра залежать від значення ключа.

На рис. 5 показано приклад вставки при подвійному хешуванні. Ви бачите хеш-таблиц. розміром 13 осередків, у якій використовуються допоміжні хеш-функції $h_1(k) = k \bmod 13$ і $h_2(k) = 1 + (k \bmod 11)$. Тому що $14 \equiv 1 \pmod{13}$ і $14 \equiv 3 \pmod{11}$, ключ 14 вставляється в порожній осередок 9, після того як при дослідженні осередків 1 і 5 з'ясовується, що ці осередки зайняті.

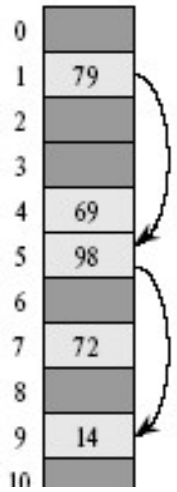


Рис. 25.5. Вставка при подвійному хешуванні

Для того щоб послідовність дослідження могла охопити всю таблицю, значення $h_2(k)$ повинне бути взаємно простим з розміром хеш-таблиці *m*. Зручний спосіб забезпечити виконання цієї умови складається у виборі числа *m*, рівного ступеня 2, і розробці хеш-функції $h_2$ таким чином, щоб вона повертала тільки непарні значення. Ще один спосіб складається у використанні в якості *m* простого числа і побудові хеш-функції $h_2$ такий, щоб вона завжди повертала натуральні числа, менші *m*. Наприклад, можна вибрати просте число в якості *m*, а хеш-функції такими:

$$h_1(k) = k \bmod m,$$
$$h_2(k) = 1 + (k \bmod m'),$$

де $m'$ повинно бути небагато менше *m* (наприклад, $m - 1$). Скажемо, якщо $k = 123456$, $m = 701$, а $m' = 700$, то $h_1(k) = 80$ і $h_2(k) = 257$, так що першої досліджуваний буде осередок у 80-й позиції, а потім буде досліджуватися кожна 257-я (по модулю *m*) осередок, поки не буде виявлений порожній осередок, чи поки не виявляться досліджені всі осередки таблиці.

Подвійне хешування перевершує лінійне чи квадратичне дослідження в розумінні кількості $\Theta\left(m^2\right)$ послідовностей досліджень, у той час як у згаданих методів ця кількість дорівнює $\Theta\left(m\right)$. Це зв'язано з тим, що кожна можлива пара $\left(h_1\left(k\right),h_2\left(k\right)\right)$ дає свою послідовність досліджень, що відрізняється від других. У результаті ефективність подвійного хешування досить близька до продуктивності "ідеальної" схеми рівномірного хешування.

### Аналіз хешування з відкритою адресацією

Аналіз відкритої адресації, як і аналіз методу ланцюжків, виконується з використанням коефіцієнта заповнення $\alpha = n/m$ хеш-таблиці при *n* і *m*, що прагнуть до нескінченності. Саме собою зрозуміло, при використанні відкритої адресації в нас може бути не більш одного елемента на осередок таблиці, так що $n \le m$ і, отже, $\alpha \le 1$.

Будемо вважати, що використовується рівномірне хешування. При такій ідеалізованій схемі послідовність досліджень $\left\langle h\left(k,0\right),h\left(k,1\right),\ldots,h\left(k,m-1\right)\right\rangle$, використовувана для чи вставки пошуку кожного ключа *k*, з рівною імовірністю є однієї з можливих перестановок $\left\langle 0,1,\ldots,m-1\right\rangle$. Зрозуміло, з кожним конкретним ключем зв'язана єдина фіксована послідовність досліджень, так що при розгляді розподілу імовірностей ключів і хеш-функцій усі послідовності досліджень виявляються рівноймовірними.

Зараз ми проаналізуємо математичне очікування кількості досліджень для хешування з відкритою адресацією в припущенні рівномірного хешування, і почнемо з аналізу кількості досліджень у випадку неуспішного пошуку.

### Теорема 25.6.

Математичне очікування кількості досліджень при невдалому пошуку в хеш-таблиці з відкритою адресацією і коефіцієнтом заповнення $\alpha = n/m < 1$ в припущенні рівномірного хешування не перевищує $1/\left(1-\alpha\right)$.

***Доведення***. При невдалому пошуку кожна послідовність досліджень завершується на порожньому осередку. Визначимо випадкову величину *X* як рівну кількості досліджень, виконаних при неуспішному пошуку, і події $A_i$ $(i = 1,2,\ldots)$, що полягають у тім, що було виконано *i-me* дослідження, і воно довелося на зайнятий осередок. Тоді подія $\left\{X \ge i\right\}$ являє

собою перетинання подій $A_1 \bigcap A_2 \bigcap \ldots \bigcap A_{i-1}$. Обмежимо імовірність $\Pr\{X \ge i\}$ шляхом обмеження імовірності $\Pr\{A_1 \bigcap A_2 \bigcap \ldots \bigcap A_{i-1}\}$. Тоді,

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

Оскільки існує *n* елементів і *m* осередків, $\Pr\{A_1\} = n/m$. Імовірність того, що буде виконане *j-те* дослідження ( $j > 1$ ) і що воно буде проведено над заповненим осередком (при цьому перші $j-1$ досліджень проведені над заповненими осередками), дорівнює $(n-j+1)/(m-j+1)$. Ця імовірність визначається в такий спосіб: ми повинні перевірити один з $(n-(j-1))$ елементів, що залишилися, а всього недосліджених до цього часу залишається осередків $(m-(j-1))$. Відповідно до припущення про рівномірне хешуванні, імовірність дорівнює відношенню цих величин. Скориставшись тим фактом, що з $n < m$ для усіх $0 \le j < m$ випливає співвідношення $(n-j)/(m-j) \le n/m$, для усіх $1 \le i \le m$ ми одержуємо:

$$\Pr\{X \ge i\} = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \ldots \frac{n-i+2}{m-i+2} \le \left(\frac{n}{m}\right)^{i-1} = \alpha^{i-1} .$$

Тепер ми можемо обмежити математичне очікування кількості досліджень:

$$\mathrm{E}[X] = \sum_{i=1}^{\infty} \Pr\{X \ge i\} \le \sum_{i=1}^{\infty} \alpha^{i-1} = \sum_{i=0}^{\infty} \alpha^{i} = \frac{1}{1-\alpha} . \qquad \blacksquare$$

Отримана границя $1 + \alpha + \alpha^2 + \alpha^3 + \ldots$ має інтуїтивну інтерпретацію. Одне дослідження виконується завжди. З імовірністю, приблизно рівної $\alpha$, перше дослідження проводиться над заповненим осередком, і потрібно виконання другого дослідження. З імовірністю, приблизно рівної $\alpha^2$, два перші осередки виявляються заповненими і потрібно проведення третього дослідження, і т.д.

Якщо $\alpha$ — константа, то з теореми 6 випливає, що неуспішний пошук виконується за час $O(1)$. Наприклад, якщо хеш-таблиця заповнена наполовину, та середня кількість досліджень при неуспішному пошуку не перевищує $1/(1-0.5) = 2$. При заповнюванні хеш-

таблиці на 90% середню кількість досліджень не перевищує $1/(1-0.9)=10$. Теорема 25.6 практично безпосередньо дає нам оцінку ефективності процедури HASH_INSERT.

**Теорема 25.7.**

Вставка елемента в хеш-таблицю з відкритою адресацією і коефіцієнтом заповнення $\alpha$ в припущенні рівномірного хешування, вимагає в середньому не більш $1/(1-\alpha)$ досліджень.

***Доведення***. Елемент може бути вставлений у хеш-таблицю лише в тому випадку, якщо в ній є вільне місце, так що $\alpha<1$. Вставка ключа вимагає проведення неуспішного пошуку, за яким випливає розміщення ключа в знайденому порожньому осередку. Отже, математичне очікування кількості досліджень не перевищує $1/(1-\alpha)$. ■

Обчислення математичного очікування кількості досліджень при успішному пошуку вимагає небагато більше зусиль.

**Теорема 8.**

Математичне очікування кількості досліджень при вдалому пошуку в хеш-таблиці з відкритою адресацією і коефіцієнтом заповнення $\alpha<1$, у припущенні рівномірного хешування і рівноімовірного пошуку кожного з ключів, не перевищує

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

***Доведення***. Пошук ключа *k* виконується з тією же послідовністю досліджень, що і його вставка. У відповідності з теоремою 7, якщо *k* був $(i+1)$-м ключем, вставленим у хеш-таблицю, то математичне очікування кількості проб при пошуку *k* не перевищує $1/(1-i/m)=m/(m-i)$. Усереднення по всіх *n* ключах у хеш-таблиці дає нам середня кількість досліджень при вдалому пошуку:

$$\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i}=\frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i}=\frac{1}{\alpha}\left(H_m-H_{m-n}\right),$$

де $H_i=\sum_{j=1}^{i}1/j$ являє собою *i-те* гармонійне число. Скориставшись методом обмеження суми інтегралом, одержуємо верхню границю математичного очікування кількості досліджень при вдалому пошуку:

$$\frac{1}{\alpha}\left(H_m - H_{m-n}\right) = \frac{1}{\alpha}\sum_{k=m-n+1}^{m}\frac{1}{k} \le \frac{1}{\alpha}\int_{m-n}^{m}(1/x)\,dx = \frac{1}{\alpha}\ln\frac{m}{m-n} = \frac{1}{\alpha}\ln\frac{1}{1-\alpha}\ . \ \blacksquare$$

Якщо хеш-таблиця заповнена наполовину, очікуване кількість досліджень при успішному пошуку не перевищує 1.387; при заповнюванні на 90% цю кількість не перевищує 2.559.

### 25.5. Ідеальне хешування

Хоча найчастіше хешування використовується через чудову *середню* продуктивність, можлива ситуація, коли реально одержати чудову продуктивність хешування в *найгіршому* випадку. Такою ситуацією є ***статична*** множина ключів, тобто коли після того як усі ключі збережені в таблиці, їх множина ніколи не змінюється. Ряд застосувань у силу своєї природи працює зі статичними множинами ключів. Як приклад можна навести множину зарезервованих слів мови програмування чи множину імен файлів на компакт-диску. Ідеальним ***хешуванням*** ми називаємо методику, що у найгіршому випадку виконує пошук за $O(1)$ звертань до пам'яті.

Основна ідея ідеального хешування досить проста. Ми використовуємо дворівневу схему хешування з універсальним хешуванням на кожнім рівні (рис. 25.6). Перший рівень по суті той же, що й у випадку хешування з ланцюжками: *n* ключів хешуються в *m* осередків з використанням хеш-функції *h*, ретельно обраної із сімейства універсальних хеш-функцій. Однак замість того, щоб створювати список ключів, хешованих в осередок *j*, ми використовуємо маленьку ***вторинну хеш-таблицю*** $S_j$ зі своєю хеш-функцією $h_j$. Шляхом точного вибору хеш-функції $h_j$ ми можемо гарантувати відсутність колізій на другому рівні.

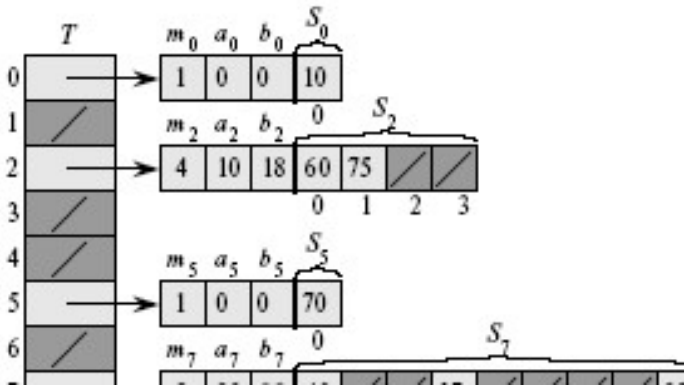


Рис. 25.6. Використання ідеального хешування для збереження множини $K = \{10, 22, 37, 40, 60, 70, 75\}$

Розглянемо детальніше приклад на рис. 25.6, де показане збереження статичної множини ключів $K = \{10, 22, 37, 40, 60, 70, 75\}$ у хеш-таблиці з використанням технології ідеального хешування. Зовнішня хеш-функція має вигляд $h(k) = ((ak + b) \bmod p) \bmod m$, де $a = 3$, $b = 42$, $p = 101$ і $m = 9$. Наприклад, $h(75) = 2$, так що ключ 75 хешується в осередок 2. Вторинна хеш-таблиця $S_j$ зберігає всі ключі, хешовані в осередок *j*. Розмір кожної таблиці $S_j$ дорівнює $m_j$, і з нею зв'язана хеш-функція $h_j(k) = ((a_j k + b_j) \bmod p) \bmod m_j$. Оскільки $h_2(75) = 1$, ключ 75 зберігається в осередку 1 вторинної хеш-таблиці $S_2$. В жодній із вторинних таблиць немає колізії, так що час пошуку в гіршому випадку дорівнює константі.

Для того щоб гарантувати відсутність колізій на другому рівні, потрібно, щоб розмір $m_j$ хеш-таблиці $S_j$ дорівнював квадрату числа $n_j$ ключів, хешованих в осередок *j*. Така квадратична залежність $m_j$ від $n_j$ може показатися надмірно марнотратною, однак далі ми покажемо, що при коректному виборі хеш-функції першого рівня очікувана кількість необхідної для хеш-таблиці пам'яті залишається рівною $O(n)$.

Ми виберемо хеш-функцію з універсальних множин хеш-функцій, описаних у розділі 25.3. Хеш-функція першого рівня вибирається з множини $\mathcal{H}_{p,m}$, де, як і в розділі 25.3, *p* є простим числом, що перевищує значення кожного з ключів. Ключі, хешовані в осередок *j*, потім повторно хешуються у вторинну хеш-таблицю $S_j$ розміром $m_j$ з використанням хеш-функції $h_j$, обраної з класу $\mathcal{H}_{p,m_j}$.

Робота буде виконана в два етапи. Спочатку ми з'ясуємо, як гарантувати відсутність колізій у вторинній таблиці. Потім ми покажемо, що очікувана кількість пам'яті, необхідної для первинної і вторинної хеш-таблиц, дорівнює $O(n)$.

**Теорема 25.9**

*Якщо n* ключів зберігаються в хеш-таблиці розміром $m = n^2$ з використанням хеш-функції *h*, випадково обраної з універсальної множини хеш-функцій, то імовірність виникнення колізій не перевищує $1/2$.

**Теорема 25.10**

Якщо ми зберігаємо *n* ключів у хеш-таблиці розміром $m = n$ з використанням хеш-функції *h*, обираної випадковим образом з універсальної множини хеш-функцій, то

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

де $n_j$ — кількість ключів, хешованих в осередок *j*.

**Наслідок 25.1.**

Якщо ми зберігаємо *n* ключів у хеш-таблиці розміром $m = n$ з використанням хеш-функції *h*, обираної випадковим образом з універсальної множини хеш-функцій, і встановлюємо розмір кожної вторинної хеш-таблиці рівним $m_j = n_j^2$ ( $j = 0,1,\ldots,m-1$ ), то математичне очікування кількості необхідної для вторинних хеш-таблиц у схемі ідеального хешування пам'яті не перевищує величини 2*n*.

**Наслідок 25.2.**

Якщо ми зберігаємо *n* ключів у хеш-таблиці розміром $m = n$ з використанням хеш-функції *h*, обираної випадковим образом з універсальної множини хеш-функцій, і встановлюємо розмір кожної вторинної хеш-таблиці рівним $m_j = n_j^2$ ( $j = 0,1,\ldots,m-1$ ), то імовірність того, що загальна кількість необхідної для вторинних хеш-таблиць пам'яті не менш 4*n*, менше ніж 1/2.

**Література**

Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ. — М.: Вильямс, 2005. — глава 11.