

Хеш таблиці

Уявіть, що ви продавець в маленькому магазинчику. Коли клієнт купує товари, ви перевіряєте їх ціну за книгою. Якщо записи в книзі не впорядковані за алфавітом, то пошук слова «апельсини» в кожному рядку займе дуже багато часу. Фактично доведеться проводити простий пошук, а для цього потрібно перевірити кожний запис. Пам'ятайте, скільки часу це займе? $O(n)$. Якщо ж книга впорядкована за алфавітом, зможете скористатися бінарним пошуком, час якого складає всього $O(\log n)$.

Але пошук даних в книзі - головний біль для касира, навіть якщо її вміст відсортовано. Поки ви перегортаєте сторінки, клієнт потихеньку починає виходити з себе. Набагато зручніше було б завести помічницю, яка пам'ятає усі назви товарів і ціни. Тоді нічого шукати взагалі не доведеться: ви питаєте помічницю, а вона миттєво відповідає.

Кількість елементів у книзі	Простий пошук $O(n)$	Бінарний пошук $O(\log n)$	Мегі $O(1)$
100	10 с	1 с	миттєво
1000	1,6 хв	1 с	миттєво
10000	16,6 хв	2 с	миттєво

Звернемося до структур даних. Поки вам відомі дві структури даних: масиви та списки. Книгу можна реалізувати у вигляді масиву.

(яблука, 2.49)	(молоко, 1.49)	(груши, 0.79)
----------------	----------------	---------------

Кожен елемент масиву насправді складається з двох елементів: назви товару і його ціни. Якщо впорядкувати масив за ім'ям, ви зможете провести по ньому бінарний пошук для визначення ціни товару. Це означає, що пошук буде виконуватися за час $O(\log n)$. Але необхідно, щоб пошук виконувався за час $O(1)$. У цьому вам допоможуть хеш-функції.

Хеш-функція це функція, яка отримує рядок та повертає число.

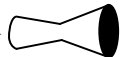
Хеш-функція повинна відповідати деяким вимогам:

- вона повинна бути послідовною. Припустимо, ви передали їй рядок «апельсини» і отримали 4. Це означає, що кожен раз в майбутньому, передаючи їй рядок «апельсини», будете отримувати 4. Без цього хеш - таблиця марна
- різними словами повинні відповідати різні числа. Наприклад, хеш - функція, яка повертає 1 для кожного отриманого слова, нікуди не годиться. В ідеалі кожне вхідне слово повинно відображатися на своє число.

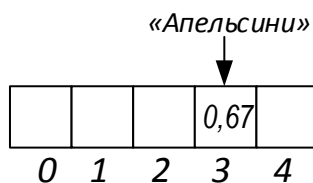
Отже, хеш-функція пов'язує рядки з числами. Почнемо з пустого масиву:

0	1	2	3	4

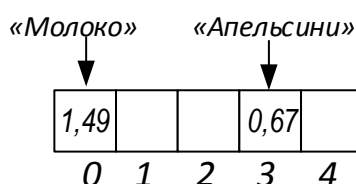
Всі ціни будуть зберігатися в цьому масиві; передамо хеш-функції рядок «Апельсини».

«Апельсини» →  → 3

Хеш-функція видає значення «3». Збережемо ціну апельсинів в елементі масиву з індексом 3.



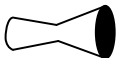
Додамо молоко. Передамо хеш-функції рядок «молоко».



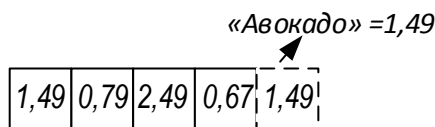
Продовжуйте діяти так, і з часом весь масив буде заповнений цінами на товари.

1,49	0,79	2,49	0,67	1,49
------	------	------	------	------

А тепер ви питаєте: скільки коштує авокадо? Шукати в масиві нічого не потрібно, просто передайте рядок «авокадо» хеш-функції.

«Авокадо» →  → 4

Результат показує, що значення зберігається в елементі з індексом 4.



Хеш-функція повідомляє, де зберігається ціна, і вам взагалі не потрібно нічого шукати! Таке рішення працює, тому що:

- хеш-функція незмінно зв'язує назву з одним індексом. Кожен раз, коли вона викликається для рядка «авокадо», ви отримуєте назад одне і те ж число. При першому виклику цієї функції ви дізнаєтеся, де слід зберегти ціну авокадо, а при наступних викликах вона повідомляє, де взяти цю ціну;
- хеш-функція пов'язує різні рядки з різними індексами. «Авокадо» зв'язується з індексом 4, а «молоко» - з індексом 0. Для кожного рядка знаходиться окрема позиція масиву, в якій зберігається ціна цього товару;
- хеш-функція знає розмір масиву і повертає тільки дійсні індекси. Таким чином, якщо довжина масиву дорівнює 5 елементів, хеш-функція не поверне 100, тому що це значення не є дійсним індексом в масиві.

Хеш-таблиці також відомі під іншими назвами: «асоціативні масиви», «словники», «відображення», «хешкарти» або просто «хеші». Хеш-таблиці виключно швидко працюють! Хеш-таблиці використовують масиви для

зберігання даних, тому при зверненні до елементів вони не поступаються масивам.

У будь-якому пристойній мові програмування існує реалізація хештаблиць.

c++: `std::unordered_set` / `std::unordered_map`

java: `java.util.HashMap <K, V>`

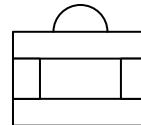
c#: `System.Collections.Hashtable, System.Collections.Dictionary <K, V>`

python: `dict`

php: `array ()`

В Python теж є хеш-таблиці ; вони називаються словниками. Нова хеш-таблиця створюється функцією `dict` :

```
>>> book=dict()
```



Порожня
Хеш-таблиця

`book` - нова хеш-таблиця. Додамо в `book` кілька цін:

```
>>> book["orange"] = 0.67 ← Апельсини стоять 67 центів
```

```
>>> book["milk"] = 1.49 ← Молоко коштує 1 долар 49 центів
```

```
>>> book["avocado"] = 1.49
```

```
>>> print book
```

```
{'Avocado': 1.49, 'orange': 0.67, 'milk': 1.49}
```

А тепер запитаємо ціну авокадо:

```
>>> print book["avocado"]
```

```
1.49 ← Ціна авокадо
```

Хеш-таблиці складається з ключів та значень. У хеш `book` імена продуктів є ключами, а ціни - значеннями. Хеш-таблиця пов'язує ключі зі значеннями .

Приклади використання

Використання хеш-таблиць для пошуку

У телефоні є зручна вбудована телефонна книга. З кожним ім'ям зв'язується номер телефону.

Телефона книга повинна підтримувати такі функції:

- додавання імені людини і номера телефону, пов'язаного з цим ім'ям;
- отримання номера телефону, пов'язаного з введенням ім'ям.

Таке завдання ідеально підходить для хеш-таблиць! Хеш-



таблиці відмінно працюють, коли ви хочете :

- створити зв'язок, що відображає один об'єкт на інший;
- знайти значення в списку.

Побудувати телефонну книгу нескладно. Почніть з створення нової хеш-таблиці:

```
>>> phone_book = dict ( ) або >>> phone_book = {}  
>>> phone_book [ "jenny" ] = 8675309  
>>> phone_book [ "emergency" ] = 911
```

Тепер припустимо, що ви хочете знайти номер телефону Дженні (Jenny). Просто передайте ключ хешу :

```
>>> print phone_book [ "jenny" ]  
8675309
```

Хеш-таблиці використовуються для пошуку відповідності в набагато більшому масштабі. Наприклад, уявіть, що ви хочете перейти на веб-сайт - припустимо, `http : //adit.io` . Ваш комп'ютер повинен перетворити символічне ім'я `adit.io` в IP -адресу. Для будь-якого відвідуваного веб-сайту його ім'я перетворюється в IP-адресу :

ADIT.IO	173.255.258.255
GOOGLE.COM	74.125.239.133
FACEBOOK.COM	173.252.120.6
SCRIBD.COM	23.235.47.175

Цей процес називається перетворенням DNS. Хеш-таблиці - всього лише один із способів реалізації цієї функціональності.

Вилучення дублікатів

Припустимо, ви керівник виборчою дільницею. Природно, кожен виборець може проголосувати лише один раз. Як перевірити, що він не голосував раніше? Коли людина приходить голосувати, ви дізнаєтеся його повне ім'я, а потім перевіряєте за списком вже виборців, що проголосували. Якщо ім'я входить в список, значить, ця людина вже проголосувала. В іншому випадку ви додасте ім'я в список і дозволяєте йому проголосувати. Тепер припустимо, що бажаючих проголосувати багато і список вже тих, хто проголосував досить великий. Кожен раз, коли хтось приходить голосувати, ви змушені переглядати: скористатися хешем !



Спочатку створимо хеш для зберігання інформації про вже тих, хто проголосував людей:

```
>>> voted = {}
```

Коли хтось приходить голосувати, перевірте, чи присутній його ім'я в хеші:

```
>>> value = voted.get ( " tom ")
```

Функція `get` повертає значення, якщо ключ `"tom"` присутній в хеш - таблиці. В іншому випадку повертається `None`. За допомогою цієї функції можна перевірити, голосував виборець раніше чи ні!

```
voted = {}
```

```
def check_voter ( name ):
```

```
    if voted.get ( name ):
```

```
        print "kick them out !"
```

```
    else :
```

```
        voted [ name ] = True
```

```
        print "let them vote !"
```

Тестуємо

```
>>> check_voter ( " tom ")
```

```
let them vote !
```

```
>>> check_voter ( " mike ")
```

```
let them vote !
```

```
>>> check_voter ( " mike ")
```

```
kick them out !
```

Використання хеш-таблиці як кеша

Уявіть, що у вас є племінниця, яка пристає до вас з питаннями про планети: «Скільки кілометрів від Землі до Марса?», «А скільки кілометрів до Місяця?», «А до Юпітера?» Кожен раз ви вводите запит в Google і повідомляєте їй відповідь. На це йде пара хвилин. А тепер уявіть, що вона завжди питає: «Скільки кілометрів від Землі до Місяця?» Досить швидко ви запам'ятовуєте, що Місяць знаходиться на відстані 384 400 кілометрів від Землі. Шукати інформацію в Google не потрібно ... Ви просто запам'ятовуєте і видаєте відповідь. Ось так працює механізм кешування: сайт просто запам'ятовує дані, замість того щоб перераховувати їх заново.

Якщо ви увійшли на Facebook, то весь контент, який ви бачите, адаптований спеціально для Вас. Кожен раз, коли ви заходите на facebook.com, серверам доводиться думати, який контент вас цікавить. Якщо ж не ввели облікові дані на Facebook, то ви бачите сторінку входу. Всі користувачі бачать одну і ту ж сторінку входу. Facebook постійно отримує однакові запити: «Я ще не ввійшов на сайт, видайте мені домашню сторінку». Сервер перестає

виконувати зайву роботу і генерувати домашню сторінку знову і знову. Замість цього він запам'ятовує, як виглядає домашня сторінка, і відправляє її вам .



Такий механізм зберігання називається кешуванням. Він володіє двома перевагами:

- ви отримуєте веб-сторінку набагато швидше;
- доводиться виконувати менше роботи.

Кешування – стандартний спосіб прискорення роботи. Всі великі вебсайти застосовують кешування. А кешовані дані зберігаються в хеше !

Facebook не просто кешує домашню сторінку. Також кешуються сторінки «Про нас», «Умови використання» і багато інших. Отже, необхідно створити зв'язок URL-адреса сторінки і даних сторінки.

facebook.com/about → *дані сторінки з інформацією про facebook*

facebook.com → *дані домашньої сторінки*

Ось як це виглядає в коді:

```
cashe = {}  
def get_page (url):  
    if cashe.get (url):  
        return cashe[url]          ← Повертаються кешовані дані  
    else :  
        data = get_data_from-sever(url)  
        cashe[url]=data            ← Дані спочатку зберігаються в кеше  
        return data
```

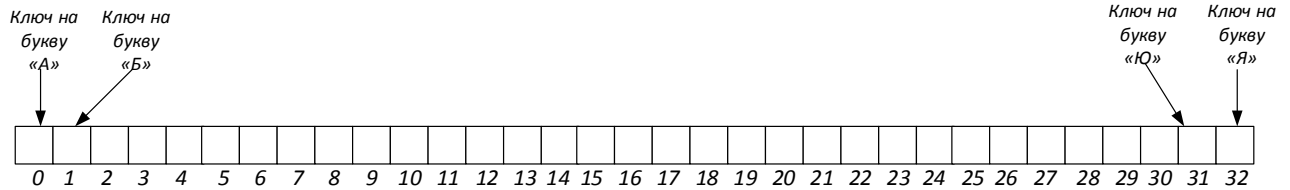
Тут сервер виконує роботу тільки в тому випадку, якщо URL не зберігається в кеші. Однак перед тим, як повертати дані, ви зберігаєте їх в кеші. Коли користувач в наступний раз запросить ту ж URL-адресу , дані можна відправити з кешу (замість того щоб заставляти сервер виконувати роботу).

Колізії

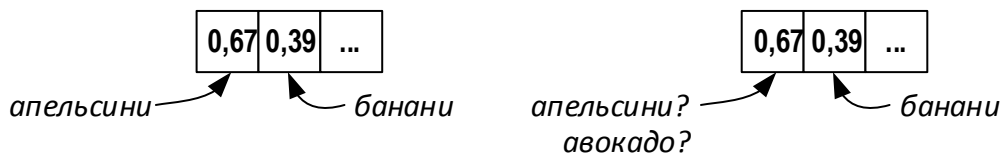
У попередньому матеріалі акцентувалася, що хеш-функція завжди відображає різні ключі на різні позиції в масиві.

Насправді написати таку хеш-функцію майже неможливо. Розглянемо простий приклад: припустимо, масив складається всього з 33 комірок.

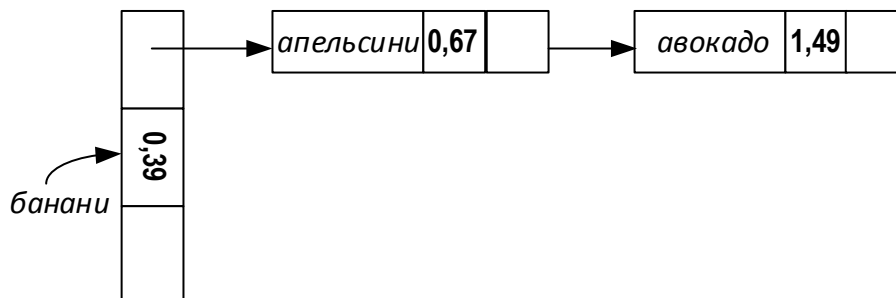
І хеш-функція дуже проста: елемент масиву просто призначається по алфавітному ознакою.



Ви хочете помістити ціну апельсинів в хеш. Для цього виділяється перша комірка. Після апельсинів в хеш заноситься ціна бананів. Для бананів виділяється друга комірка.



Тепер в хеш необхідно включити ціну авокадо. І для авокадо знову виділяється перша комірка. Елемент вже зайнятий апельсинами! Що ж робити? Така ситуація називається колізією: двом ключам призначається один елемент масиву. Виникає проблема: якщо зберегти в цьому елементі ціну авокадо, то вона запишеться на місце ціни апельсинів. Колізії - неприємна штука, і вам доведеться якось розбиратися з ними. Існує багато різних стратегій обробки колізій. Найпростіша з них виглядає так: якщо кілька ключа показує на один елемент, в цьому елементі створюється зв'язаний список.



У цьому прикладі і «Апельсини», і «Авокадо» відображаються на один елемент масиву, тому в елементі створюється зв'язаний список. Якщо вам буде потрібно дізнатися ціну бананів, ця операція, як і раніше виконається швидко. Якщо буде потрібно дізнатися ціну апельсинів, робота піде трохи повільніше. Вам доведеться провести пошук по зв'язаному списку, щоб знайти в ньому «апельсини». Якщо зв'язаний список малий, це не так страшно - пошук буде

обмежений трьома або чотирма елементами. Якщо кожний елемент хеш-таблиці зберігається в зв'язаному списку, то робота з даними сповільнюється.

З цього прикладу слідують два важливих моменти:

- вибір хеш-функції дійсно важливий. Хеш-функція, яка відображає всі ключі на один елемент масиву, нікуди не годиться. В ідеалі хеш-функція повинна розподіляти ключі рівномірно по всьому хешу;
- якщо пов'язані списки стають занадто довгими, робота з хеш-таблицею сильно сповільнюється.

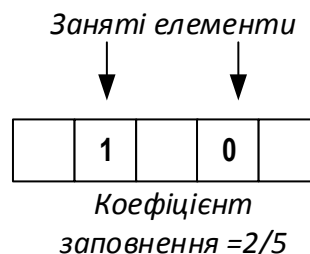
Гарна хеш-функція створює мінімальне число колізій. Як же вибрати хорошу хеш-функцію?

Коефіцієнт заповнення

Коефіцієнт заповнення хеш-таблиці обчислюється за простою формулою .

$$\frac{\text{Кількість елементів в хеш-таблиці}}{\text{Загальна кількість елементів}}$$

Хеш-таблиці використовують масив для зберігання даних, тому для обчислення коефіцієнта заповнення можна підрахувати кількість заповнених елементів в масиві. Наприклад, в наступній хеш-таблиці коефіцієнт заповнення дорівнює 2/5, або 0,4.



Припустимо, в хеш-таблиці потрібно зберегти ціни 100 товарів і хеш - таблиця складається з 100 елементів. У кращому випадку кожному товару буде виділений окремий елемент. Коефіцієнт заповнення цієї хеш-таблиці дорівнює 1. А якщо хеш-таблиця складається всього з 50 елементів? Тоді її коефіцієнт заповнення буде дорівнює 2. Виділити під кожен товар окремий елемент ні за яких умов не вдасться, тому що елементів попросту не вистачить! Коефіцієнт заповнення більше 1 означає, що кількість товарів перевищує кількість елементів в масиві.

З ростом коефіцієнта заповнення в хеш-таблицю доводиться додавати нові елементи, тобто змінювати її розмір. Уявімо, що ця хеш-таблиця наближається до заповнення. Хеш-таблицю необхідно розширити. Розширення починається зі створення нового масиву більшого розміру. Зазвичай в такому випадку створюється масив вдвічі більшого розміру.

4	3	1	
---	---	---	--

Коефіцієнт
заповнення=3/4

--	--	--	--	--	--	--	--

	4		1	3			
--	---	--	---	---	--	--	--

Коефіцієнт заповнення=3/8

Тепер всі ці елементи необхідно заново вставити в нову хеш-таблицю функцією hash. Нова таблиця має коефіцієнт заповнення 3/8. З меншим коефіцієнтом завантаження число колізій зменшується, і таблиця починає працювати більш ефективно. Гарне наближене правило: змінюйте розмір хеш-таблиці, коли коефіцієнт заповнення перевищує 0,7. Але ж на зміну розмірів йде багато часу! Так, зміна розмірів вимагає значних витрат ресурсів, тому воно не повинно відбуватися занадто часто.

Гарна хеш-функція повинна забезпечувати рівномірний розподіл значень в масиві. Погана хеш-функція створює скупчення і породжує безліч колізій. Гарна хеш-функція реалізує алгоритм SHA .

Метод поділу

Нехай k - ключ (той, що необхідно хешувати), а N - максимально можливе число хеш -кодів. Тоді метод хешування за допомогою поділу полягатиме у взятті залишку від ділення k на N :

$h(k) = k \bmod N$, де \bmod - операція взяття залишку від ділення.

Наприклад, на вхід подаються наступні ключі: 3, 6, 7, 15, 32, 43, 99, 100, 133, 158.

Визначимо N рівним 10, тому можливі значення хеш лежать в діапазоні 0 ... 9. Використовуючи цю функцію, отримаємо наступні значення хеш -коду:

$h(3) = 3, h(6) = 6, h(7) = 7, h(15) = 5, h(32) = 2, h(42) = 2, h(99) = 9, h(100) = 0, h(133) = 3, h(158) = 8$.

Програма, що виконує хешування методом розподілу

```

1  #include "stdafx.h"
2  #include <iostream>
3  using namespace std;
4  int HashFunction(int k)
5  {
6  return (k%10);
7  }
8  void main()
9  {
10 setlocale(LC_ALL, "Rus");
11 int key;
12 cout<<"Ключ > "; cin>>key;
13 cout<<"HashFunction("<<key<<")="<<HashFunction(key)<<endl;
14 system("pause>>void");
15 }

```

<http://kvodo.ru/hesh-funktsii.html>

Припустимо, у вас є невелика компанія з 20 співробітників і вам хотілося б отримувати інформацію про кожного з них за ідентифікаційним номером (ІН). Один із способів зберегти дані - сформувати масив з 100 елементів, в якому ІН будуть відповідати позиціях $N \bmod 100$. Наприклад, співробітник з ІН 2190 буде розміщений в позиції 90, з ІН 2817 - в позиції 17, а з ІН 3078 - в позиції 78. Щоб знайти певного працівника, досить скористатися формулою $ІН \bmod 100$ і переглянути відповідну запис в масиві. Подібна операція займе час $O(1)$, тобто на перевірку виявиться швидшим, ніж інтерполяційний пошук.

На практиці не все так просто. Якщо співробітників виявиться досить багато, то серед них знайдуться двоє, у яких ІН буде відповідати одному і тому ж значенню. Скажімо, ІН 2817 і 1 317 займуть в таблиці позицію 17.

Метод множення

Отримати з вихідної послідовності ключів послідовність хеш-кодів, використовуючи метод множення (мультиплікативний метод), значить скористатися хеш-функцією:

$$h(k) = \lfloor N * \{k * A\} \rfloor$$

Тут A - раціональне число, по модулю менше одиниці ($0 < A < 1$), а k і N позначають те саме, що і в попередньому методі: ключ і розмір хеш-таблиці. Також права частина функції містить три пари дужок:

- $()$ - дужки пріоритету;
- $\lfloor \rfloor$ - дужки взяття цілої частини;
- $\{ \}$ - дужки взяття дробової частини.

Аргумент хеш-функції k ($k \geq 0$) в результаті дасть значення хеш-коду $h(k) = x$, що лежать в діапазоні $0 \dots N-1$. Для роботи з негативними числами можна число x взяти по модулю.

Від вибору A і N залежить те, наскільки оптимальним виявиться хешування множенням на певній послідовності. Не маючи відомостей про вхідні ключі, як N слід вибрати одну зі ступенів двійки, т. як множення на 2^m рівносильно зсуву на m розрядів, що комп'ютером проводиться швидше. Непоганим значенням для A (в загальному випадку) буде $(\sqrt{5} - 1) / 2 \approx 0,6180339887$. Воно засноване на властивостях золотого перетину.

При такому A , хеш-коди розподілятися досить рівномірно, але багато що залежить від початкових значень ключів.

Для демонстрації роботи мультиплікативного методу, припустимо $N = 13$, $A = 0,618033$. Як ключі візьмемо числа: 25, 44 і 97. Підставимо їх в функцію:

$$h(k) = \lfloor 13 * \{25 * 0,618033\} \rfloor = \lfloor 13 * \{15,450825\} \rfloor = \lfloor 13 * 0,450825 \rfloor = \lfloor 5,860725 \rfloor = 5$$

$$h(k) = \lfloor 13 * \{44 * 0,618033\} \rfloor = \lfloor 13 * \{27,193452\} \rfloor = \lfloor 13 * 0,193452 \rfloor = \lfloor 2,514876 \rfloor = 2$$

$$h(k) = \lfloor 13 * \{97 * 0,618033\} \rfloor = \lfloor 13 * \{59,949201\} \rfloor = \lfloor 13 * 0,949201 \rfloor = \lfloor 12,339613 \rfloor = 12$$

Реалізація методу на C ++ з використанням N і A:

```

1  #include "stdafx.h"
2  #include <iostream>
3  using namespace std;
4  int HashFunction(int k)
5  {
6  int N=13; double A=0.618033;
7  int h=N*fmod(k*A, 1);
8  return h;
9  }
10 void main()
11 {
12 setlocale(LC_ALL, "Rus");
13 int key;
14 cout<<"Ключ > "; cin>>key;
15 cout<<"HashFunction("<<key<<")="<<HashFunction(key)<<endl;
16 system("pause>>void");
17 }

```

Пряме зв'язування

У хеш-таблиці з прямим зв'язуванням (рис. 1) значення ключів зберігаються в спеціальних наборах записів, званих блоками. Кожен з них є вершиною зв'язного списку, в якому знаходяться прив'язані до блоку елементи.

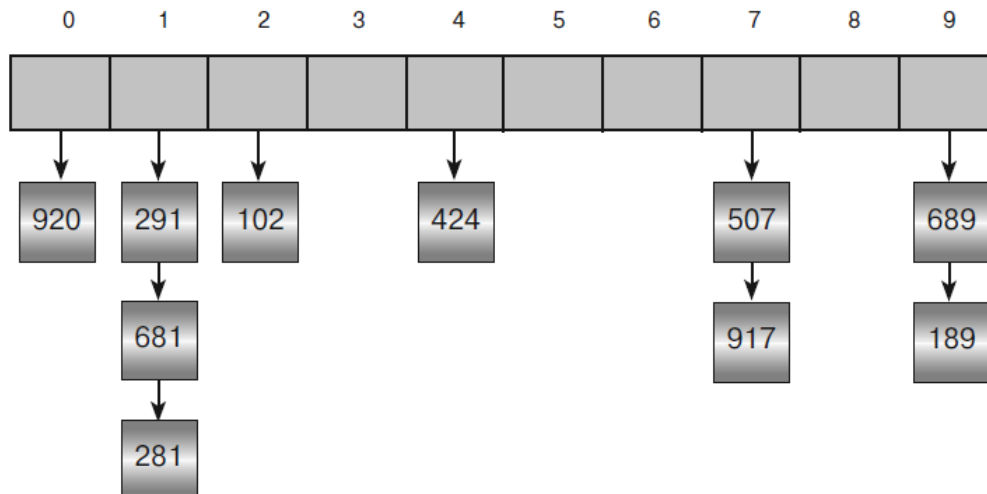


Рис. 1. Хеш-таблиця з прямим зв'язуванням

Зазвичай блоки розташовані в масиві таким чином, що можна використовувати просту функцію хешування для визначення їх ключа. Наприклад, якщо у вас N блоків, а ключі числові, зв'яжіть ключ K з блоком під номером $K \bmod N$. Щоб додати ключ до хеш-таблиці, його потрібно спочатку співставити з блоком через функцію хешування, а потім вставити нову комірку в верхню частину списку. Кожна з двох операцій потребують $O(1)$ кроків, тобто займе дуже мало часу. За визначенням хеш-таблиця не повинна містити дубльованих значень. Якщо в ній B блоків і N елементів, які розподілені досить

рівномірно, то кожен зв'язний список блоку буде включати приблизно N / B елементів. Таким чином, щоб перевірити, чи присутній новий елемент в блоці, знадобиться $O(N/B)$ кроків. Це означає, що для додавання елемента в хеш-таблицю потрібно всього $O(1) + O(N/B) = O(N/B)$ кроків.

Щоб знайти потрібний елемент, програмі необхідно хешувати ключ і визначити, в якому з блоків він може міститися, а потім рухатися по зв'язаному списку до тих пір, поки не буде досягнутий його кінець або не виявиться шукане. Якщо ви доберетеся до кінця списку, значить, запитуваного елемента в хеш-таблиці немає. Як і у випадку з додаванням елемента, належить виконати $O(N/B)$ кроків.

Видалити елемент з хеш-таблиці з прямим зв'язуванням також нескладно: потрібно хешувати ключ елемента, щоб знайти потрібний блок, і виконати відповідну операцію в зв'язковому списку. Для хешування знадобиться $O(1)$ кроків, безпосередньо видалення займе $O(N/B)$ кроків. Таким чином, загальне час складе $O(N/B)$. Хеш-таблиці з прямим зв'язуванням може розширюватися і стискатися в міру необхідності, тому вам не потрібно спеціально змінювати її розмір. Однак якщо зв'язані списки стануть занадто довгими, то пошук і видалення елементів займуть багато часу. В цьому випадку вам знадобиться збільшити таблицю, щоб створити більше блоків. Оскільки при рехешуванні таблиці не потрібно проводити пошук дублікатів до кінця зв'язного списку в кожному блоці, повністю впоратися з операцією можна за час $O(N)$.

Відкрита адресація

Ще один спосіб реалізації хеш-таблиць - відкрита адресація. У цьому випадку значення зберігаються в масиві, а функція хешування являє собою незначні розрахунки. Наприклад, в масиві з M записами проста функція хешування може зв'язати значення ключа K з позицією $K \bmod M$.

У різних видах відкритої адресації використовуються різні функції хешування. Неоднакова і політика вирішення колізій, але в загальному випадку вона виглядає так: *для кожного значення в масиві підбирається кілька комірок, і якщо перша вже зайнята, алгоритм пробує використовувати другу, потім третю і так до тих пір, поки не знайде вільну або не прийде до висновку, що такої немає.*

Серія комірок, яку алгоритм підбирає для значення, називається пробної послідовністю. За її середньою довжиною хорошо оцінювати наповненість хеш таблиці. В ідеалі пробна послідовність повинна дорівнювати 1 або 2, великі цифри говорять про повну таблиці.

Іноді політика вирішення колізій така, що для елемента може не знайтися вільної комірки, навіть коли вона є. Якщо пробна послідовність повторює саму

себе перед тим, як перевірити чергову запис, деякі записи можуть залишитися невикористаними.

Щоб знайти елемент в хеш-таблиці, алгоритм слідує за пробною послідовністю, поки не відбудеться одна з трьох подій.

1. Якщо пробна послідовність зуміла відшукати елемент, завдання виконано.

2. Якщо пробна послідовність знаходить порожній запис в масиві, елемента немає.

3. Пробна послідовність перевіряє M записів (за розміром масиву) - і алгоритм приходить до виводу, що значення відсутнє. Послідовність може перебрати не усі елементи, але якщо пройде по всім, то ви будете знати, що вони точно переглянуті або що цільовий елемент не знайдений. Вона також може перевірити в циклі одну і ту ж позицію кілька разів. У будь-якому випадку значення не має бути присутнім, оскільки інакше воно б додавалося до масиву з використанням тієї ж пробної послідовності.

При розумному заповненні хеш-таблиці відкрита адресація працює дуже швидко. Якщо довжина пробної послідовності дорівнює 1 або 2, додавання і знаходження елементів виконуються за час $O(1)$. Але якщо масив з N елементів істотно переповнений, продуктивність знижується. У найгіршому разі алгоритм прийде до виводу, що елемента в масиві немає, за час $O(N)$. Пошук присутності елементів також буде виконуватися вкрай повільно.

Ви можете збільшити розмір масиву, що б зменшити коефіцієнт наповнення хеш-таблиці. Для цього створіть новий масив і рехешуйте елементи в ньому. Для кожного з них операція займе $O(1)$ часу, а загальна продуктивність алгоритму складе $O(N)$.

Лінійне пробірування

У лінійному пробіруванні політика вирішення колізій додає до кожної комірки постійне число (найчастіше 1), яке називається кроком по індексу, яке генерує пробну послідовність. При кожному черговому додаванні береться розмір масиву по модулю, при необхідності послідовність повертається до початку масиву.

Припустимо, в хеш-таблиці 100 елементів, а правило хешування звучить так: N пов'язано з коміркою $N \bmod 100$. Тоді пробна послідовність для значення 2197 перевіряє комірки 97, 98, 99, 0, 1, 2 і т. д.

На рис. 2 представлений масив з 10 записів, який вже містить деякі значення. Щоб додати в нього нове значення 71, використовуючи лінійну пробну послідовність, потрібно зв'язати його з осередком $71 \bmod 10 = 1$. Але ця комірка вже зайнята значенням 61, тому алгоритм переходить до осередку 2,

який теж заповнений. Наступною повинна бути комірка 3 - вона вільна, і алгоритм розміщає там 71.

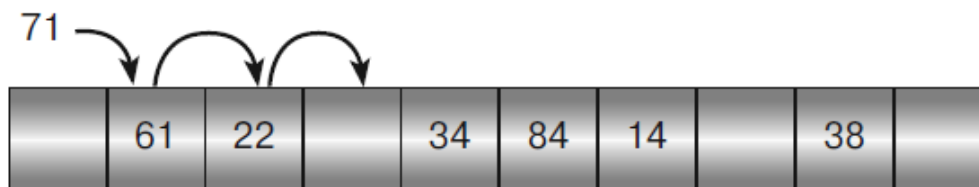


Рис. 2. Лінійна пробна послідовність

Перевага даного методу - простота. Якщо необхідно, пробна послідовність пройде по кожній комірці масиву і вставить елемент у вільне місце, якщо воно ще залишилося. Але є і супутній недолік – так звана первинна кластеризація, яка проявляється в утворенні великих блоків суміжних записів і призводить до довгих пробних послідовностей.

У результаті при додаванні нового елемента і його хешуванні до будь-якого запису в кластері пробна послідовність змушена пройти через весь кластер, щоб знайти вільну комірку.

Квадратичне пробірування

Виникнення великих кластерів при лінійному пробіруванні пов'язано з тим, що нові елементи зв'язуються з комірками, що стоять в кінці групи, і поступово збільшують її. Запобігти такій ситуації допомагає квадратичне пробірування. Для створення пробної послідовності в якості кроку за індексом береться квадрат кількості комірок. Іншими словами, якщо в лінійному пробіруванні існує послідовність $K, K + 1, K + 2, K + 3, \dots$ то в квадратичному варіанті вона буде виглядати так: $K, K + 1^2, K + 2^2, K + 3^2, \dots$ у цьому випадку, якщо два елементи виявляться пов'язаними з різними позиціями в одному і тому ж кластері, вони не обов'язково будуть притримуватись однієї пробної послідовності і потраплять в кінець кластера.

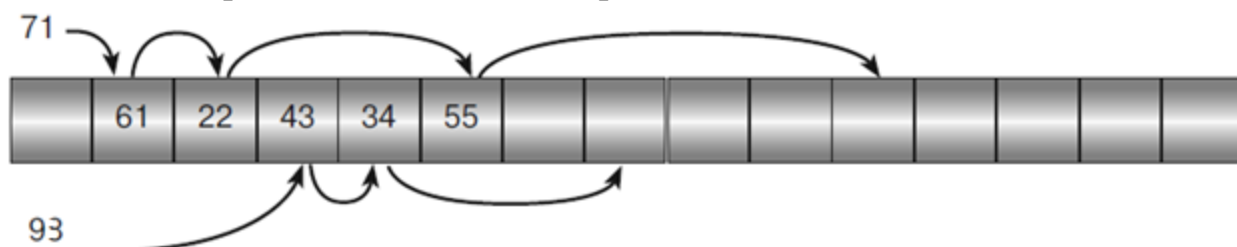


Рис. 3. Квадратичне пробірування

На рис. 3 показана хеш-таблиця, на початку якої є група з п'яти елементів. Нове значення 71 отримує пробну послідовність $1, 1 + 1^2 = 2, 1 + 2^2 = 5, 1 + 3^2 = 10$ і не додається до існуючого кластера. Значення 93 спочатку зв'язано з тим

же кластером, але згідно власної пробної послідовності 3 , $3 + 1^2 = 4$, $3 + 2^2 = 7$ теж не потрапляє в нього.

Квадратичне пробірування запобігає первинну, але не вторинну кластерізацію, при якій значення, пов'язані з однаковою початковою позицією в масиві, отримують одну і ту ж пробну послідовність, іноді дуже довгу. В результаті утворюється точно така ж група елементів, але вже не зібраних разом, а розподілених по всьому масиву.

Ще один недолік квадратичного пробірування пов'язаний з тим, що воно не зможе знайти вільну позицію, навіть якщо в хеш-таблиці їх кілька. Справа в тому, що з кожним разом переміщення по масиву відбувається все далі і далі, і незаповнена комірка просто пропускається.

Псевдовипадкове пробірування

Це пробірування подібно лінійному, за виключенням того, що крок за індексом формує псевдовипадкова функція з початково зв'язаної комірки. Припустимо, що є комірка K , тоді пробна послідовність буде виглядати так: K , $K + p$, $K + 2p$, ... де p визначається псевдовипадковою функцією.

Подібно квадратичному пробіруванню, псевдовипадкове запобігає тільки первинній кластерізації, але страждає від вторинної: значення, пов'язані з однією і тією ж початковою позицією, розміщуються в хеш-таблиці згідно з однією і тією ж пробною послідовністю. Точно так же псевдовипадкове пробірування може пропускати деякі невживані записи.

Подвійне хешування

Щоб позбутися від вторинної кластерізації значення, пов'язані з однією і тією ж початковою коміркою, повинні отримувати різні пробні послідовності. І тут стане в нагоді подвійне хешування. Воно схоже на псевдовипадкове пробірування, тільки крок для індексу задається не псевдовипадковою функцією початкової комірки, а функцією хешування.

Припустимо, значення A і B пов'язані з позицією K . У псевдовипадковому пробіруванні крок за індексом p генерується функцією $F1(K)$, потім обидва значення використовують пробну послідовність K , $K + p$, $K + 2p$, $K + 3p$, ... У подвійному хешуванні для зв'язування початкових значень A і B застосовується функція псевдовипадкового хешування $F2$. В результаті при одному і тому ж початковому K утворюються дві пробні послідовності з різними кроками по індексу: $pA = F1(A)$ і $pB = F2(B)$ відповідно.

Незважаючи на те що подвійне хешування добре справляється з первинною та вторинною кластерізацією, воно також, як і псевдовипадкове пробірування, може пропускати невживані записи.

1. Бхаргава А. Грокаем алгоритмы. Иллюстрированное пособие для программистов и любопытствующих. - СПб.: Питер, 2017 . - 288 с. (100-126)
2. Стивенс, Род. Алгоритмы. Теория и практическое применение / Род Стивенс. — Москва : Издательство «Э», 2016. — 544 с. (169-181)
3. Луридас, Панос. Алгоритмы для начинающих : теория и практика для разработчика / Панос Луридас ; [пер. с англ. Е.М. Егоровой]. — Москва : Эксмо, 2018. — 608 с. (393-447)
4. Рафгарден Тим. Совершенный алгоритм. Графовые алгоритмы и структуры данных. - СПб.: Питер, 2019. -256 с. (195-244)
5. <http://kvodo.ru/hesh-funktsii.html>
6. <http://evilcoderr.blogspot.com/2013/01/hash-table-c.html>
7. https://www.opennet.ru/docs/RUS/qt3_prog/x5383.html
8. <http://cppstudio.com/post/9535/>