

## Ch11. 雜湊表 Hash Table

接下來進入雜湊表的章節，首先一樣會先簡介什麼是雜湊表，以及雜湊表的特色跟原理，再來會實作出雜湊表。

在實作雜湊表的時候，不可避免的會遇到「碰撞」的問題，因此要講解如何處理碰撞，接著，來瞭解一下在實作雜湊表的時候有什麼課題需要面對，最後來看 C++ STL 裡面的雜湊表。

### 課程大綱

- A. 雜湊表簡介與雜湊函式 Hash Function
- B. 實作雜湊表
- C. 碰撞 Collision 處理
- D. 雜湊表的挑戰
- E. Map 與 Dict

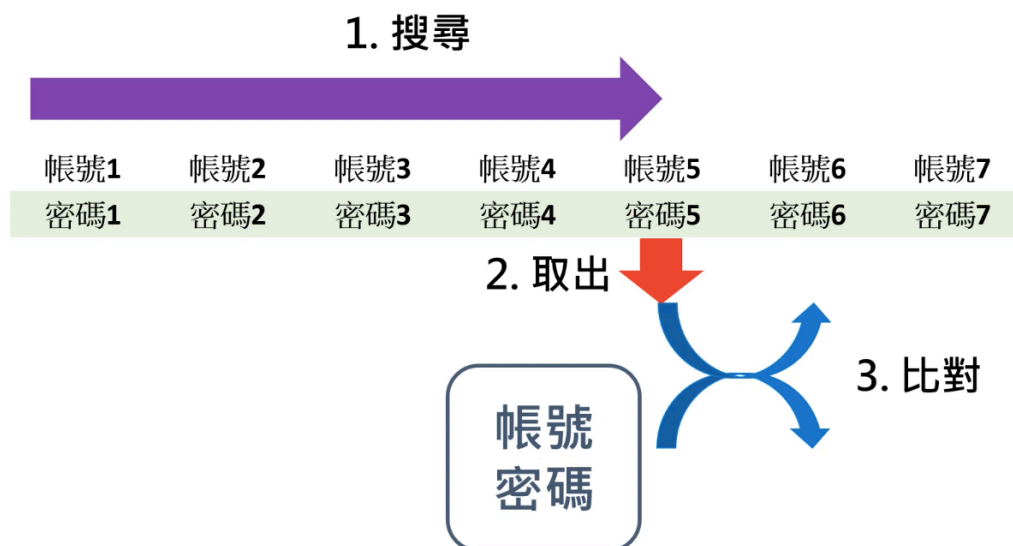
### 第一節：雜湊表簡介

先前在簡介資料結構的時候，已經有稍微提過：如果要知道特定的資料在陣列的哪個位置，必須從第一筆資料開始一路找到最後一筆資料才能確定，所以大部分的時間都花在搜尋該元素對應的索引值，花費時間為  $O(n)$ 。



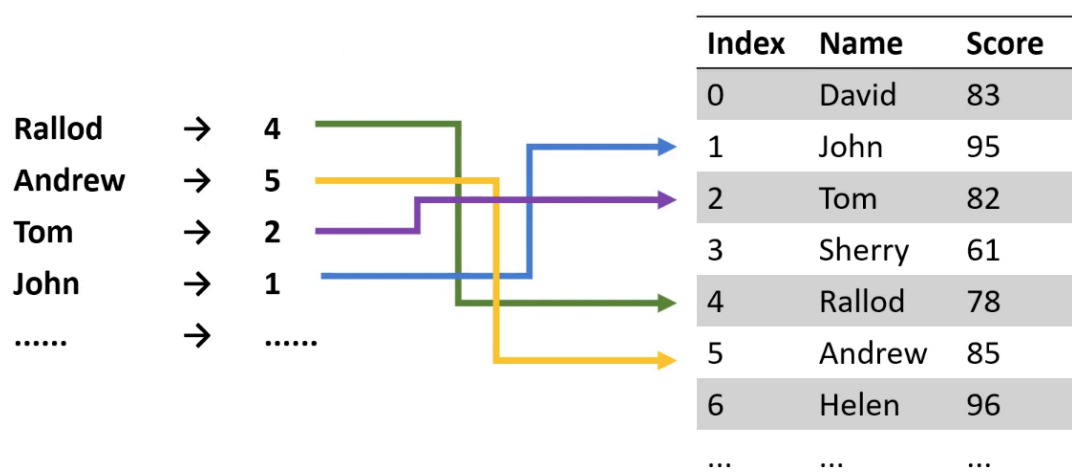
在陣列中尋找特定資料	
1	int search(int *p, int len, int value)
2	{
3	for(int i=0;i<len;i++){
4	if(*(p+i)==value)
5	return i;
6	}
7	return -1;
8	}

另外，在處理網站登入的時候，也會遇到一樣的問題，要判斷一次登入嘗試是否成功，就要找到使用者輸入的帳號對應到哪個密碼，必須從陣列的第一筆資料開始一筆一筆往後，需要的時間也是  $O(n)$ ，花費大量的時間在尋找正確索引值上。



## 1. 雜湊表的功能

可以想像，如果有一個方法可以「把查詢值直接對應到索引值」，那包含「搜尋、新增、刪除」等操作的複雜度就降到  $O(1)$ ，而非之前的  $O(n)$ 。



上圖中，目標是找到 Rallod、Andrew、Tom、John 四筆資料的位置，如果可以用某種「神奇的方法」，把 Rallod 直接對應到索引值 4（而不用從索引值 0、1、2、3 一個一個找），就節省了從開頭遍歷陣列的時間。

同樣的，用這個方法直接把 Andrew 對應到 5，直接到索引值 5 的位置來取出相應的資料；這個方法也可以把 Tom 對應到 2，代表 Tom 的資料在 2 的位置。

如果可以直接轉換：

查詢的值 Value -> 索引值 Index

上面轉換的實現即下面的等式：

$$\text{index} = \text{hash}(\text{Name})$$

上式中，把姓名 Name 透過「雜湊函式」直接轉換成索引值 index。

如果這種轉換能成功進行，搜尋就變得非常方便迅速，這就是雜湊表的基本原理與目標：透過一個雜湊函式，把想要查詢的資料直接轉成索引值，立刻知道資料儲存的位置。

## 2. 雜湊函式 Hash Function

### (1) 雜湊函式的定義

#### A. 雜湊函式可以定義如下

給定任意的輸入 input，輸出 output 都必須介於 0 和  $m-1$  之間

其中  $m$  是陣列的長度，只要符合定義，就是一個雜湊函式。

#### B. 雜湊函式的數學定義

雜湊函式把字集合  $U$ （任意輸入）轉換成  $0 \sim m-1$  的整數。

$\text{hash} : U \rightarrow \{0, 1, \dots, m-1\}$ （多對一函式）

注意到，因為字集有無限多種可能性（涵蓋任意輸入），一定會發生「碰撞」的問題，也就是「不同的資料」根據雜湊函式對應到「同樣的索引值」，後面會介紹這個問題如何解決。

## (2) 雜湊函式的需求

### A. 雜湊函式需要有兩個特性

- a. 計算簡單：盡量接近  $O(1)$
- b. 平均分布

會使用雜湊函式，就是想要把搜尋、新增的複雜度從  $O(n)$  降成  $O(1)$ ，所以如果用雜湊函式轉換本身就需要大量時間，就失去意義了。

另外一方面，也會希望雜湊函式的轉換能夠平均分佈，比如函式把所有的資料映射到 0 到  $m-1$  之間，「每個整數對應的輸入」其數量如果接近平均分佈，就能減少碰撞的發生。

## (3) 常見的雜湊函式

常見的雜湊函式有以下幾種，我們會一一介紹

- A. 除法 Division
- B. 乘法 Multiplication
- C. Mid-square
- D. Folding addition
- E. Digit analysis

### 3. Division

$$\text{hash}(\text{key}) = \text{key} \% m$$

透過把資料 key 對 m 取餘數的做法，將所有輸入壓回到  $0 \sim m - 1$ 。

#### (1) Division 的優缺點

- A. 優點：計算快速簡便
- B. 缺點：很難決定如何取 m
  - a. 如果 m 取很大， $0 \sim m - 1$  的陣列會占很大的空間
  - b. 如果 m 取很小，則陣列一下就放滿了

c. m 怎麼取是 Division 算法最大的難題

## (2) Division 的實例

Character	D	a	v	i	d
Ascii code	68	97	118	105	100
Total	488				

實際看一下 Division 的進行方式，如果想要把字串「David」轉換成索引值，該如何進行呢？

假設現在陣列的長度是 8，也就是索引值只有 0 到 7，字串是由 'D'、'a'、'v'、'i'、'd' 五個字元組成。

$if\ m = 8$

$hash(488) = 488 \% 8 = 0$

$hash(David) = 0$

也就是 David 這筆資料（key）映射到的 m 值為 0。

先把每個字元都轉換成 Ascii 碼，分別為 68、97、118、105、100，這五個數字的總和是 488，把 488 對 8 取餘數，得到 0，這也就代表 David 這個字串透過雜湊函式得到的「雜湊值」是 0，應該到 0 這個索引值去取用資料。

## (3) Division 的處理過程

整理一下 Division 的步驟

- 把輸入字串中的每個字元轉換成 Ascii 碼（整數）
- 把得到的所有整數加總
- 把加總得到的值對陣列長度（上例中為 8）取餘數
- 得到的餘數就是索引值

這樣的雜湊函式有沒有缺點？最明顯的缺點之一就是「字串中字母的順序」不會影響雜湊值，所以任意打亂字母的順序，得到的雜湊值都會完全一樣，所以當字串中字母的順序很重要時，上述的雜湊函式不甚理想。

Character	d	D	a	v	i
Ascii code	100	68	97	118	105
Total	488				

#### (4) 考慮字母順序

試著設計一個新的方式來解決這個問題：因為電腦是二進位，所以可以轉而使用下面的函式，先試著把每個字元的 Ascii 碼根據字元的位置乘上  $256 (2^8)$  的不同次方，再對 8 取餘數。

Character	D	a	v	i	d
Ascii code	68	97	118	105	100
Total	293692926308				

$$m = 8, 2^8 = 256$$

$$68 \times 256^4 + 97 \times 256^3 + 118 \times 256^2 + 105 \times 256^1 + 100 \times 256^0 = 293692926308$$

$$\text{hash}(293692926308) = 293692926308 \% 8 = 4$$

意即 David 對應到索引值 4，David -> 4。



上面的做法是否完全解決了字母順序調換的問題呢？因為除了  $d$  乘上的數字是  $256^0$ ，也就是 1，其他幾個位置乘上的數字都是 256 的次方數，同時也是 8 的倍數，因此透過一些數學計算，可以證明出只要最後一個字元  $d$  的位置不變，前面四個字元任意調換仍然不會改變得到的索引值。

A. 通常表格長度  $m = 2^n$

$$\text{hash}(\text{key}) = \text{key} \% 2^n$$

B.  $n = 3$  時：

$$\begin{aligned}\text{hash}(\text{key}) &= \text{key} \% 2^3 \\ &= \text{key} \% 8\end{aligned}$$

由上可知， $m$  盡量不要等於  $2^n$ ，才能避免正好整除而無法反映「字母順序」的問題；只是，表格長度設定為  $2^n$  是常見的做法，因此 Division 方法很難達到完全考慮「字母順序」的效果。

#### 4. 乘法 Multiplication

##### (1) Multiplication 的進行方式

乘法 Multiplication 的處理步驟（假定  $m = 2^n$ ）

A. 選擇一個常數  $C$ ， $0 < C < 1$

B. 將輸入的鍵值  $\text{Key}$  與  $C$  相乘，得到  $\text{Key} \times C$

C. 取得  $\text{Key} \times C$  的小數點後部分  $\text{frac}$ ，其中

$$\text{frac} = \text{Key} \times C - \lfloor \text{Key} \times C \rfloor$$

（這步的目的是把  $\text{Key} \times C$  的值壓到 0 到 1 之間）

D. 把  $\text{frac}$  與  $m$  相乘，得到  $m \times \text{frac}$

（這步的目的是把 0~1 映射到  $0 \sim m - 1$ ）

E.  $\text{Hash} = \lfloor m \times \text{frac} \rfloor$

首先選擇一個介於 0 與 1 之間的常數  $C$ ，再來，把常數  $C$  乘上想要算的資料  $\text{Key}$ （剛剛的例子裡，就是人名  $\text{David}$ ，也就是要轉成索引值的輸入資料）。

下一步是取出  $Key \times C$  的小數部分，以數學方式表達就是把  $Key \times C$  減去  $Key \times C$  的「向下取整（最接近但不大於某數的整數）」，接下來，把得到的小數  $frac$  乘上陣列大小  $m$ ，最後的雜湊值是  $m \times frac$  的整數部分。

範例（假定  $m = 2^3 = 8$ ,  $Key = 488$ ）

A. 任意選擇常數  $C = \frac{7}{16}$ ,  $0 < C < 1$ ）

B. 將輸入的鍵值  $Key$  與  $C$  相乘， $488 \times C = 488 \times \frac{7}{16}$

C. 取得  $Key \times C$  的小數點後部分  $frac$

$$frac = 488 \times \frac{7}{16} - \left\lfloor 488 \times \frac{7}{16} \right\rfloor = \frac{8}{16}$$

D. 把  $frac$  與  $m$  相乘， $8 \times \frac{8}{16} = 4$

E.  $Hash = \left\lfloor 8 \times \frac{8}{16} \right\rfloor = 4$

乘法 **Multiplication** 方法的核心精神是把  $Key$  轉換成一個 0 到 1 之間的隨機小數（當然並非完全隨機）。

## (2) Multiplication 的特性

A. Multiplication 的數學表示法如下：

$$hash(key) = \lfloor m((key \times C) \% 1) \rfloor$$

其中  $(key \times C) \% 1$  指的是  $key \times C$  小數點後的部分。

**Multiplication** 方法的特性是使用到了  $Key$  中每個 **bit** 的資料，不會發生剛剛 **Division** 例子中順序調換完全不影響，或者只有部分影響雜湊值的情形。

**Knuth**，一個電腦科學領域著名的學者建議常數  $C$  應該取  $\frac{\sqrt{5}-1}{2} \sim 0.618$ ，只是這個運算有點複雜，因此有時會改採位元平移（**Bit shifting**）方式處理，比較省時。



## 5. 其他常見的雜湊函式

### (1) Mid-Square

- A. 把 Key 平方，取中間幾位數當 index
- B. Key = 488 ,  $488^2 = 238,144$
- C. 00000000 0000001[1 10]100010 01000000
- D. 取第 15 個 bit 到第 17 個 bit：二進位的 [110] 相當於 6
- E. David 這個 Key 被映射到索引值 6，David -> 6

### (2) Folding addition：切割後再相加

- A. Key = 488
- B.  $4 + 8 + 8 = 20$ （也可以只取其中某幾位數，例如只取奇數、只取偶數，或者只取特定位數）
- C.  $20 \% 8 = 4$ （8 是陣列長度）
- D. David -> 6

## 6. 利用除法 Division 設計雜湊函式

- A. 輸入：字串
- B. 輸出：整數（即索引值）

除法 Division	
1	#include <iostream>
2	using namespace std;
3	
4	// input 是 Key，m 是陣列長度
5	int Hash_Func_Div(string input, int m){
6	
7	// sum 使用 long long int 型態，可以存放的數字較大
8	long long int sum = 0;
9	
10	// 把輸入 input 中的每個字元取出來

11	for(char c:input){
12	// sum 是每個字元 ascii 碼的總和
13	// 因為 sum 是 int , c 會被自動轉換
14	sum += c;
15	}
16	
17	// 回傳值是 sum 對 m 取餘數
18	return sum%m;
19	}
20	
21	int main()
22	{
23	// 讓使用者輸入一個字串 str
24	string str;
25	cout << "Please enter a str:" << endl;
26	cin >> str;
27	
28	// 輸出經過雜湊函式轉換得到的索引值，假設陣列長度是 8
29	cout << "Index:" << Hash_Func_Div(str, 8);
30	
31	return 0;
32	}
執行結果	
Please enter a str:	
>> Hello	
Index: 4	

上面的函式讓使用者傳入一個字串，並且回傳字串對應到的索引值是多少。

## 7. 利用乘法 Multiplication 設計雜湊函式

A. 輸入：字串

B. 輸出：整數

## 乘法 Multiplication

```
1 int Hash_Func_Mul(string input, int m){
2
3     // 一樣用 sum 來存放加總結果
4     long long int sum = 0;
5
6     // 常數 c 採用剛才介紹過的值  $\frac{\sqrt{5}-1}{2}$ 
7     double c = (sqrt(5)-1)/2.0;
8
9     // 讓每個字元乘上 tmp (256 的次方數) 以把字元次序納入考慮
10    int tmp = 1;
11    for(char ch:input){
12        sum += tmp * ch;
13        tmp *= 256;
14    }
15
16    // 取出 c*sum 的小數部分
17    double frac = c*sum - (int)(c*sum);
18
19    // 把 frac 從 0 到 1 之間映射到 0 到 m-1
20    int index = frac * m;
21
22    return index;
23 }
24
25 int main()
26 {
27     // 讓使用者輸入一個字串 str
28     string str;
29     cout << "Please enter a str:" << endl;
30     cin >> str;
31
32     // 輸出經過雜湊函式轉換得到的索引值，假設陣列長度是 8
```

33	cout << "Index:" << Hash_Func_Mul(str, 8);
34	return 0;
	}
執行結果	
Please enter a str:	
>> Hello	
Index: 7	

這樣就完成了兩個最基本的雜湊函式了。

## 第二節：實作雜湊表

接下來，我們要來實作出雜湊表。

比如想要用某個學生的名字（**Key**）來查詢這個學生的分數（**Value**），使用一般的陣列時要從第一筆資料慢慢往後找，但有了雜湊表之後，則可以直接把要查詢的學生名字透過雜湊函式轉換，得到該學生資料所在的索引值，過程只需要  $O(1)$  時間，代表不管陣列有多長，搜尋、新增、刪除的速度都是一樣的。

A. Key 跟 Value 的資料型態可以不同（學生的名字是字串，成績是整數）

Key -> Value

B. 常見的操作：查詢、新增、刪除

C. 實作步驟

a. 宣告結構存放 Key 與 Value

Key：學生姓名，Value：學生成績

Key：客戶姓名，Value：存款餘額

b. 以 Array 存放結構：像一個櫃子，學生或客戶的資料都放到這個櫃子中

c. 使用雜湊函式把「要搜尋的資料」對應到「索引值」

d. 把資料存到對應的索引值中：之後搜尋可以直接使用雜湊函式得到資料存放的位置

陣列

結構
結構
結構
結構
結構
結構
結構
結構

## 1. 實作雜湊表

實作一個銀行的系統，這個系統可以（透過 `Multiplication` 雜湊函式）在  $O(1)$  的時間內取出客戶的資料。

結構

- A. Key （客戶姓名）
- B. Value （客戶存款）

`Unordered_Map` 類別

- A. 建構式
- B. 查詢
- C. 新增
- D. 刪除

### (1) 建立銀行系統的雜湊表

- A. 把字串對應到存簿餘額
- B. 撰寫雜湊表內的搜尋函式

擴寫先前的 `main.cpp`：

class Unordered_Map	
1	#include <iostream>
2	#include <math.h>
3	#include <stdlib.h>
4	using namespace std;
5	
6	// 存放客戶姓名和餘額的結構
7	// 目標是透過雜湊函式轉換，只要傳入 Key 就回傳 Value
8	template<typename T1, typename T2>
9	struct Data{
10	T1 Key;
11	T2 Value;
12	};
13	
14	// 目標是用 Balance["Rallod"] 的方式取出 Rallod 的存簿餘額
15	// 需要重載中括號 [] 運算子
16	
17	template<typename T1, typename T2>
18	class Unordered_Map{
19	private:
20	// 宣告指向 Data 結構的 Pointer
21	Data<T1,T2>* Pointer;
22	int len;
23	// 利用模板撰寫，雜湊函式的傳入參數型態為 T1
24	// 函式回傳值（資料所在的索引值）型態為 int
25	int Hash_Func_Div(T1);
26	int Hash_Func_Mul(T1);
27	
28	public:
29	// 重載中括號運算子
30	T2& operator[] (T1);
31	// 建構式：預設長度是 128
32	Unordered_Map(int=128);
33	

34	}
35	...     // 函式定義在這裡
36	

建構式	
1	template<typename T1, typename T2>
2	Unordered_Map<T1,T2>::Unordered_Map(int m){
3	
4	// 把 m 賦值給私有成員 len
5	len = m;
6	
7	// 讓 Pointer 指向長度為 len 的 Data 陣列
8	// 前面是資料轉型，後面是資料的空間大小
9	Pointer = (Data<T1,T2>*)malloc(sizeof(Data<T1,T2>)*len);
10	
11	}

重載中括號 [] 運算子，假設傳入資料都是字串	
1	template<typename T1, typename T2>
2	T2& Unordered_Map<T1,T2>::operator[](T1 input){
3	
4	// 使用 Multiplication 雜湊函式取出 input 對應到的索引值
5	int index = Hash_Func_Mul(input);
6	
7	// 中括號用作輸入一筆資料時：把資料存入陣列中索引值 index 處
8	Pointer[index].Key = input;
9	
10	// 中括號用作查詢時：取得陣列中索引值 index 處的資料並回傳
11	return Pointer[index].Value;
12	
13	}

雜湊函式 Hash\_Func\_Div：可以直接取用 len，因此不需傳入陣列長度 m

```
1  template<typename T1, typename T2>
2  int Unordered_Map<T1,T2>::Hash_Func_Div(T1 input){
3
4      long long int sum=0;
5
6      // 假設輸入是字串，sum 把每個字元的 ascii 值加總
7      for(char c:input){
8          sum += c;
9      }
10
11     // 改對 len 取餘數
12     return sum%len;
13
14 }
```

雜湊函式 Hash\_Func\_Mul

```
1  template<typename T1, typename T2>
2  int Unordered<T1,T2>::Hash_Func_Mul(T1 input){
3
4      long long int sum = 0;
5      double c = (sqrt(5)-1)/2.0;
6
7      int tmp = 1;
8      for(char ch:input){
9          sum += tmp*c;
10         tmp *= 256;
11     }
12
13     double frac = c*sum-(int)(c*sum);
14     int index = frac*len;
15     return index;
16 }
```



測試 unordered_map	
1	int main()
2	{
3	// 輸入客戶名字
4	string str;
5	cout << "Please enter client name:" << endl;
6	cin >> str;
7	
8	// 輸入客戶存簿餘額
9	int number;
10	cout << "Please enter a balance:" << endl;
11	cin >> number;
12	
13	// 宣告把字串映射成整數的 unordered_map，變數名稱為 balance
14	unordered_map<string,int> balance;
15	
16	// 儲存一筆客戶資料的結構
17	// 客戶名字(Key)為 str，餘額(Value)為 number
18	balance[str] = number;
19	
20	// 輸出陣列中 Key 為 str 的這個結構中的 Value 值
21	// 即 str 這個客戶的存簿餘額
22	cout << "Balance of client" << str << "is: " << balance[str];
23	
24	return 0;
25	}
執行結果	
Please enter client name:	
>> LKM	
Please enter a balance:	
>> 50	
Balance of client LKM is: 50	

一筆資料存入 unordered\_map 後，就可以隨心所欲的取用（與 Python 的

Dictionary 的使用方法相同)。

#### A. 存入資料方式

```
balance["Mick"] = 50;
```

```
balance["John"] = 100;
```

#### B. 取出資料方式

```
cout << balance["Mick"];
```

```
cout << balance["John"];
```

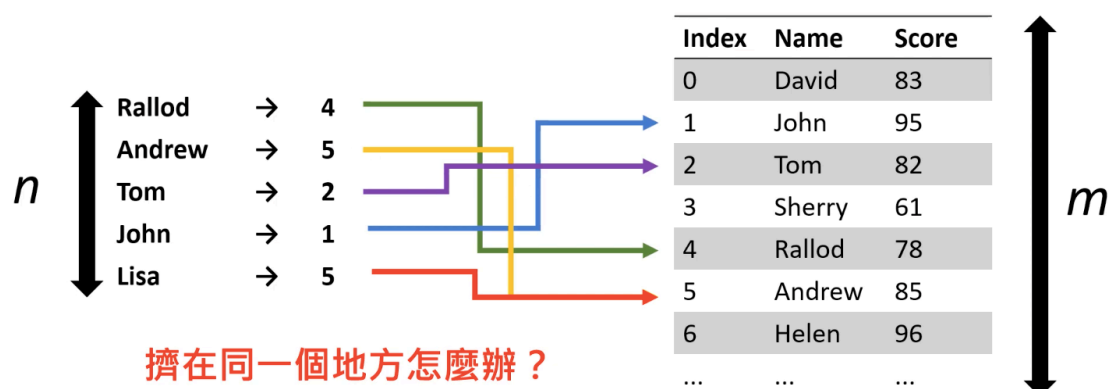
### 第三節：碰撞處理

講解完整個雜湊表的架構之後，接著就可以來處理「碰撞」的課題。

#### 1. 為什麼會發生碰撞

為什麼會有碰撞的發生呢？因為雜湊函式是把「輸入」轉換成「輸出」，輸入是查詢的鍵值 **Key**，輸出是索引值 **index**， $m$  是陣列長度。

有一個很明顯的問題：輸入是字集合，所以輸入會有無限多種狀況，輸出則是  $0 \sim m - 1$ ， $m$  是陣列長度。輸入有無限多種，輸出則只有  $m$  種，代表這是一個多對一的函式：很多種輸入可能對應到同一個索引值。



比如 Rallod 被轉換成索引值 5，Lisa 也被轉換成索引值 5，兩筆資料對應到了陣列中的同一個位置。

然而陣列中每個位置預設只能放一筆資料，不能同個位置既放 Rallod 又放 Lisa，這就是我們要處理的「碰撞」問題。

該如何處理這種「不同的資料被放在同一個索引值」的情形呢？

## 2. Load factor

**Load factor** 值被用來描述雜湊表裡的值是多還是少：當  $\frac{n}{m}$  越大的時候，代表資料越多，當  $\frac{n}{m}$  接近 0 的時候，代表裡面沒什麼資料。

$$Load\ factor(\alpha) = \frac{n}{m}$$

m：陣列的大小；n：陣列裡資料的個數。

為什麼會有碰撞發生？這是因為雜湊函式是一個多對一函式，很多種不同的輸入可能會對應到同一個輸出（索引值）。

當 n 遠大於 m 的時候，*Load factor* 的值很大，代表陣列中資料放得非常滿，很容易就會發生碰撞。

## 3. 處理碰撞的方式

處理碰撞的方式大致上可以分成下列三種：

### A. Open Addressing

- a. Linear Probing
- b. Quadratic Probing
- c. Double Hashing

### B. Perfect hashing

### C. Chaining

A. **Open Addressing** 是指當一個索引值已經有資料時仍要放入另一組資料，就往陣列後方一直找，直到找到一個空位可以放入，「此地不留爺，自有留爺處」，類似「發現一間客棧滿了，就去下一間看」的想法。

B. **Perfect Hashing** 是指用兩個雜湊表做成一個雜湊表。

C. **Chaining** 則是讓陣列中的每一筆資料都是一個鏈結串列 **Linked List**，這樣陣列中的每一個位置都可以放很多筆不同鍵值對應的資料。

#### 4. Open Addressing

先來看第一種 **Open Addressing**，當發現一個位置已經被佔用，就往後找到下一個位置。

**Open Addressing** 的各種做法中，最簡單最容易實作的是 **Linear Probing**，指的是用「線性」的方式去找下一個空著的位置。

##### (1) Linear Probing

**Open Addressing : Linear Probing**

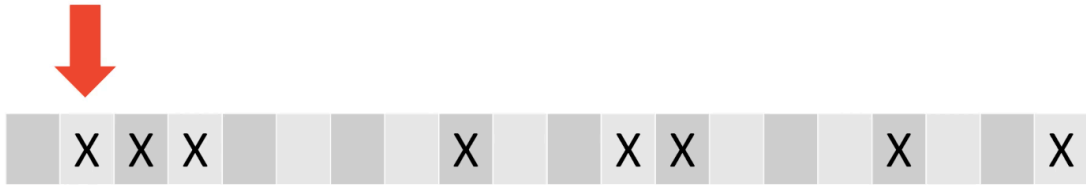
$$\text{hash}(\text{key}, i) = (\text{hash}(\text{key}, 0) + i) \% m$$

$i = \text{collision times} = 0$

在 **linear probing** 中，想知道某個 **key** 值對應到的索引值的話，要多加一個參數  $i$ ， $i$  代表的是「碰撞次數」。

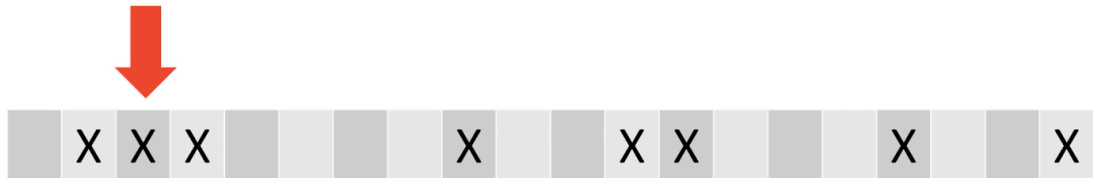
怎麼知道鍵值碰撞  $i$  次之後的雜湊值應該是多少呢？其實就是沒有發生任何碰撞時的索引值  $\text{hash}(\text{key}, 0)$  再加上  $i$ ，代表碰撞第一次的時候會右移一格，又碰撞第二次的話，就再右移一格。

Collision



比方說，如果一開始要把資料插入到圖中最左邊的 X 處，但發現要插入的位置裡面已經有資料了（用 X 表示），這時發生了第一次碰撞，因此右移一格。

Collision



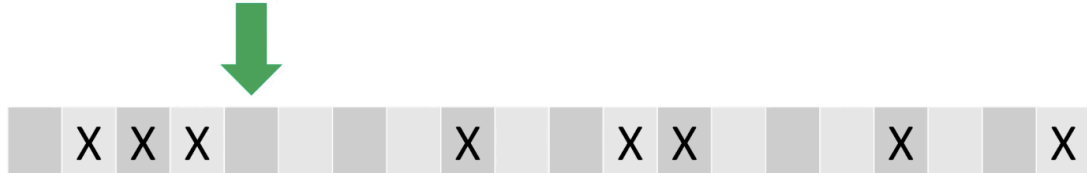
然而右移之後，裡面仍然有資料，那就再往右一格。

Collision



因為裡面還是有資料，所以再往右一格。

Okay !



右移了三次後，終於有空位出現，此時就可以放入資料並決定「碰撞次數  $i$  為 3」。

Linear Probing 是否有問題呢？最明顯的問題就是這種做法會導致「資料密集在同一個區間插入」，結果一個雜湊表裡面有一區滿滿都是資料，另一區空空如

也，因為新增資料都發生在同一區塊。

這要如何解決呢？可以簡單改一下  $i$ ，讓  $i$  代表的意義不要是線性的，而是一個二次函式  $ai^2 + bi$ ，每次要往右移動幾格都由  $ai^2 + bi$  決定，係數  $a$  跟  $b$  的值可以自由決定。

## (2) Quadratic Probing

### Open Addressing: Quadratic Probing

$$\text{hash}(\text{key}, i) = (\text{hash}(\text{key}, 0) + ai^2 + bi) \% m$$

$i = \text{collision times} = 0$

$(a = 1, b = 2 \text{ in this case})$

假設取  $a=1$ 、 $b=2$ ，當我們還沒有處理碰撞的時候，起始位置一樣是圖中最左邊的  $X$ ，第一次碰撞的時候把  $i$  代 1，第二次碰撞的時候把  $i$  代 2。

$$ai^2 + bi = i^2 + 2$$

$$i = 1 : 1^2 + 2 = 3$$

$$i = 2 : 2^2 + 2 = 6$$



用上面的方法，第一次就會往右移動 3 格，直接找到了一個空位。



Linear Probing 跟 Quadratic Probing 最基本的差異是前者使用  $i$  的「線性方程式」決定如何移動，後者則是用  $i$  的「二次方程式」決定。

### (3) Double Hashing

Quadratic Probing 仍然存在問題： $a$  和  $b$  的值應該怎麼取呢？如果  $a$  和  $b$  取得太小，資料還是會擠在同一個區間。

第三種做法叫做 Double Hashing，使用兩個雜湊函式去算出索引值，第一個雜湊函式  $hash1$  是指如果沒有發生任何碰撞的話，輸入會落在哪個位置，第二個函式  $hash2$  則是用來決定當碰撞發生的時候會落在哪個位置。

#### Open Addressing：Double Hashing

$$hash(key, i) = (hash1(key) + ihash2(key)) \% m$$

$i$  = collision times

Double Hashing 的數學寫法如上式：

$key$  的第  $i$  次碰撞的索引值 =

「沒有發生任何碰撞時的索引值」+「碰撞次數  $i$  乘上  $hash2$ 」

$hash2$  負責指出發生碰撞的時候，應該要往下找幾格，因為這種做法用到了「兩個」雜湊函式，所以叫做 Double Hashing。

## 5. Perfect Hashing

### (1) Universal Hashing 通用雜湊

「通用雜湊」是避免所有鍵值被指派到同一索引值的一種方式。

#### Universal Hashing

- A. 隨機選擇哈希函式：準備十幾二十，甚至上百個雜湊函式，每次要使用的時候，隨機取用不同的函式
- B. 同樣的鍵值可能對應到不同索引值

C. 讓碰撞的發生機率固定在  $\frac{n}{m}$  (n 是資料筆數、m 是資料大小)

## (2) Perfect Hashing 的實現方式

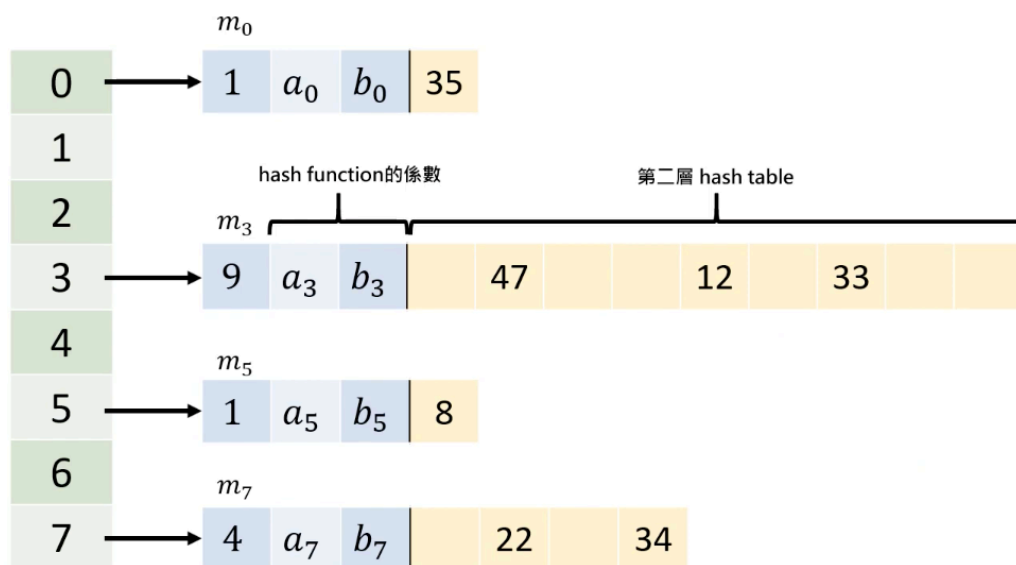
### A. 利用兩次 Universal Hashing

- 第一次 hash 到的是另一個 hash table
- 第二次 hash 到的才是真正的 value

### B. Worst case 下的運算複雜度為 $O(1)$

### C. 可透過挑選適合的 hash function 使得空間複雜度仍為 $O(1)$

Perfect Hashing 的想法是使用兩次 Universal Hashing，第一次雜湊映射到一個雜湊表，根據這個雜湊表進行第二次雜湊才會取得 Value。



綠色的陣列是第一層的雜湊表，每個索引值裡面存放的是另一個雜湊表，第二層雜湊表裡面才是存放的資料。

Perfect Hashing 在最壞的情況下時間複雜度仍然是  $O(1)$ ，因為第一層雜湊的轉換複雜度是  $O(1)$ ，第二層雜湊的轉換複雜度也是  $O(1)$ ，兩個  $O(1)$  的操作前後接在一起仍然是  $O(1)$ ： $O(1) + O(1) = O(1)$ 。

## (3) Perfect Hashing 的碰撞機率

讓  $m = n^2$  可以確保：碰撞機率  $< 1/2$ 。



證明

A. 總共有  $C_2^n$  對鍵值可能碰撞

B. 每對鍵值以  $\frac{1}{m}$  的機率碰撞

$$P(\text{collision}) = \frac{C_2^n}{n^2}$$

$$= \frac{n^2 - n}{2} \times \frac{1}{n^2} < \frac{1}{2}$$

總共有  $C_2^n$  組鍵值可能發生碰撞（ $n$  個鍵值裡兩個一組可以取出的組數），分母則是陣列（插槽）的大小，也就是  $m$ 。

雖然根據以上證明，知道只要讓插槽的數目是資料個數的平方，碰撞機率就會小於  $\frac{1}{2}$ ，但是這樣一來資料有一百筆時，插槽要有一萬個，資料有一萬筆時，插槽就要有一億個。改為使用兩層雜湊表，則可以大幅減少需要的空間。

比如說，如果我索引值 0 裡要放的資料只有一筆（圖中的 35），那索引值 0 對應到的陣列長度只需要  $1^2 = 1$ ，而索引值 3 裡放的資料有 3 筆，對應到的陣列長度是  $3^2 = 9$ ，這樣一來，Perfect Hashing 所需要的空間就可以被大幅壓低。

比較一下兩種做法在資料量  $n$  為 10,000 時需要的空間：

A. 只有一層雜湊表：要確保  $m > n^2$ ，陣列長度要有一億

B. 使用兩層雜湊表時，假設

a. 第一層的長度為一百

b. 第二層的每個雜湊表平均需要放  $\frac{10000}{100} = 100$  筆資料

c. 每個第二層雜湊表長度需要  $100^2 = 10,000$

d. 100 個長度為 10000 的陣列，總長只有一百萬

e. 加上第一層雜湊表用到的長度 100，所需空間也只有第一種做法的

$$\text{約 } \frac{1}{100}$$

## 6. Chaining

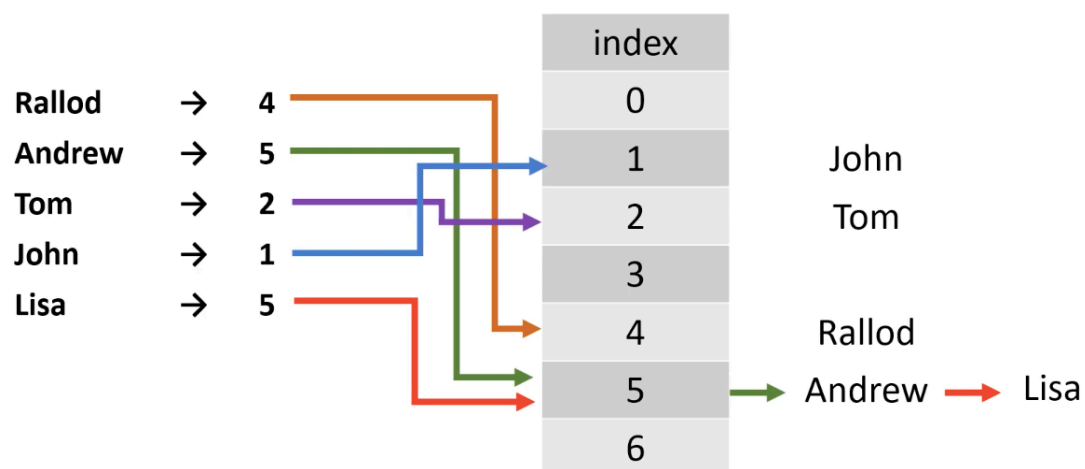
再來介紹第三種方式：Chaining。

Chaining 是一種比較簡單的方法，稍後實作時也會以 Chaining 為主，所以如果讀者覺得前面的數學太繁雜，只把 Chaining 搞懂也可以。

Chaining 是使用 Linked List 來做連結，比如剛剛的例子裡，Andrew 和 Lisa 都被轉成 5，兩者都需要放在索引值 5 當中的話，應該怎麼處理呢？

### (1) Chaining 的基本概念

陣列裡的每個位置可以不再放單一的資料，而是放入鏈結串列，當發現超過一筆資料需要放在陣列的同一個索引值當中的時候，就把這些資料都存在同一個鏈結串列裡。



索引值 5 已經放入 Andrew 的話，要再把 Lisa 也放入索引值 5 時，就加到 Andrew 的後面，意即每筆碰撞後的資料都被加到同一個鏈結串列裡。

### (2) Chaining 的時間複雜度

Chaining 的時間複雜度在最差的情況下是  $O(n)$ ，發生在所有的 Key 都被 Hash function 分到同一個索引值的時候，所有資料等同於放在單一個 Linked list 裡。這時要搜尋這個鏈結串列，自然就需要  $O(n)$ 。

Chaining 的時間複雜度平均而言是  $O(1 + \alpha)$ ，其中  $Load\ factor(\alpha) = \frac{n}{m}$ 。

- A. Load factor 是資料數量  $n$  與 slot 數量  $m$  的比例，也是每個 slot 裡 linked list 的平均長度（平均會有多少筆資料落到同一個插槽內）
- B.  $O(1)$  是先把輸入值轉成索引值
- C.  $O(\alpha)$  是遍歷一個長度  $\alpha$  的 Linked List

當 Load factor 的值不大時， $\alpha$  是一個較小的值， $O(1 + \alpha) = O(1)$ 。

總結來說，碰撞無法被完全避免，只能藉由一些數學函式想辦法「降低碰撞的可能」，比如以乘法 multiplication 方法取代除法 division 方法，減少碰撞次數來增加效能（每次遇到碰撞繼續往後尋找都會花費時間）。

## 7. 雜湊表的效能

$\alpha$ : load factor

	Open Addressing Uniform Probing	Open Addressing Linear Probing	Chaining
Successful Search	$\frac{1}{\alpha} \times \ln\left(\frac{1}{1-\alpha}\right)$	$\frac{1}{2}\left(1 + \left(\frac{1}{1-\alpha}\right)^2\right)$	$1 + \alpha$
Unsuccessful Search	$\frac{1}{1-\alpha}$	$\frac{1}{2}\left(1 + \left(\frac{1}{1-\alpha}\right)\right)$	$1 + \alpha$

### (1) Successful Search 和 Unsuccessful Search

#### A. Successful Search 成功的搜尋

刪除時找到有值的位置：要刪除一筆特定的資料，需要先找到那筆資料的位置

#### B. Unsuccessful Search 失敗的搜尋

插入時找到空的值以插入：找到一個還是空的位置可以放入資料

## (2) Open Addressing

- A. 當一個位置滿了去找下一個位置
- B. 不需要頻繁更改記憶體
- C. 不像 Chaining 要放資料到鏈結串列裡，可能要頻繁調整長度

當  $\alpha = \frac{n}{m} \sim 1$  時， $\frac{1}{1-\alpha} \sim \infty$ ：當  $n$  接近  $m$ ，雜湊表快滿的時候，發生碰撞的機率會很大，比如一個雜湊表中有 100 個空間，而其中 99 個空間裡面已經有資料了，這樣要再放一筆資料進去，可以預期會發生大量的碰撞。

## (3) Open Addressing Uniform Probing：使用 Double Hashing

- A. 對於所有的雜湊函式  $h(k,0)$ 、 $h(k,1)\dots$ ， $k$  是鍵值

後面的數字是碰撞發生的次數，雜湊函式間無關，也就是 Universal Hashing。

- B. 對於陣列中任意的空間，平均來說
  - a. 已有資料的機率： $\alpha$
  - b. 尚未有資料的機率： $1 - \alpha$

有  $\frac{n}{m} = \alpha$  的空間已經有資料了，也就是說，隨機挑選一個位置時，挑到已經被佔用的位置的機率是  $\alpha$ ，而挑到還沒被佔用的位置的機率則是  $1 - \alpha$ 。

- C. Unsuccessful Search 的複雜度： $\frac{1}{1-\alpha}$

$$1 + \alpha + \alpha^2 + \alpha^3 + \dots = \frac{1}{1-\alpha}$$

$\alpha < 0.6$ , for all small constants.

對於不成功的搜尋（找到一個還沒被佔用的位置）而言，所有鍵值都至少需要先做一次搜尋，該次搜尋有  $\alpha$  的機率會發生第一次碰撞，而導致需要進行第二次搜尋，那麼有多少比例的鍵值需要做第三次搜尋呢？就是前兩次都發生碰撞

者，其中第一次發生碰撞的機率為  $\alpha$ ，第二次發生碰撞的機率為  $\alpha^2$ ，所以

所有 Key 都要做第一次搜尋：1

+  $\alpha$  比例的 Key 要做第二次搜尋： $\alpha$

+  $\alpha^2$  比例的 Key 要做第三次搜尋： $\alpha^2$

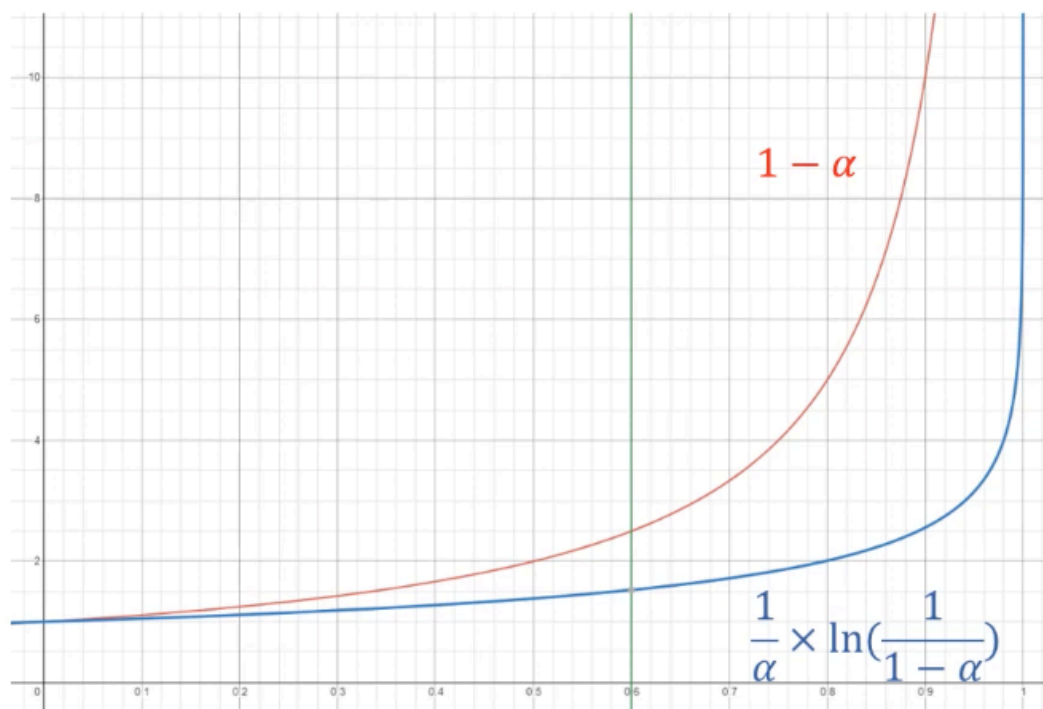
+  $\alpha^3$  比例的 Key 要做第四次搜尋： $\alpha^3$

...

...

預期需要做的搜尋次數就是等比級數公式：

$\frac{\text{首項}}{1-\text{公比}}$ ，也就是  $\frac{1}{1-\alpha}$ ，時間複雜度是  $O(\frac{1}{1-\alpha})$ 。



從圖上可以看出（橫軸是 Load factor  $\alpha$ ，縱軸是需要進行的搜尋次數），當  $\alpha$  的值小於 0.6 時， $\frac{1}{1-\alpha}$  的值是可以接受的。

D. Successful Search 的複雜度：成功搜尋，要找到資料之後做刪除

對於第  $(i + 1)^{st}$  筆插入到雜湊表中的資料來說

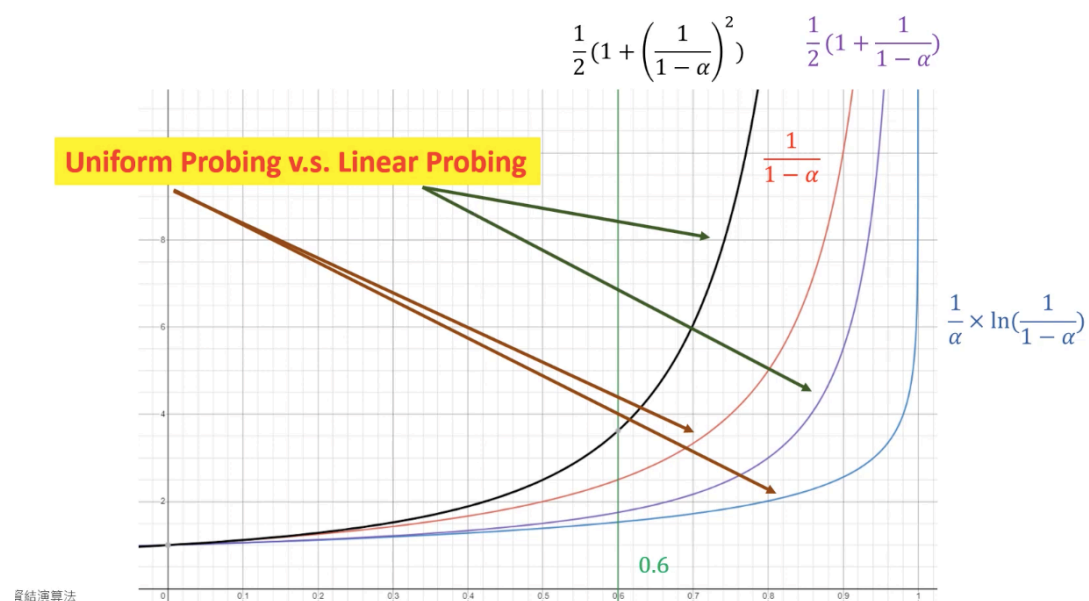
$$\begin{aligned}
 & \frac{1}{n} \sum_{i=0}^{n-1} \frac{1}{1 - \frac{i}{m}} \\
 &= \frac{1}{n} \sum_{i=0}^{n-1} \frac{m}{m-i} \\
 &= \frac{m}{n} \sum_{i=0}^{n-1} \frac{1}{m-i} \\
 &= \frac{1}{\alpha} \sum_{k=m-n+1}^m \frac{1}{k} \\
 &\leq \frac{1}{\alpha} \int_{m-n}^m \frac{1}{x} dx \\
 &= \frac{1}{\alpha} \ln \frac{m}{m-n} \\
 &= \frac{1}{\alpha} \ln \frac{1}{1-\alpha}
 \end{aligned}$$

最前面的  $\frac{1}{n}$  代表我們想要得到的是全部  $n$  筆資料的平均時間，所以先把所有時間用  $\Sigma$  加起來之後，再除以  $n$ 。

第  $i$  次插入沒有發生碰撞的機率是  $\frac{1}{1 - (\frac{i}{m})}$ ，中間有一些積分的計算我們不細

講，實務上只要知道當  $\text{load factor} < 0.6$  的時候，Open Addressing 的複雜度是可以接受的。

(6) 比較複雜度



比較 Uniform Probing 和 Linear Probing，Uniform Probing 是用兩個雜湊表、Double Hashing 的方式做，Linear Probing 是線性往後找的方式。

觀察上圖中曲線的意義，紅線和藍線是 Uniform Probing 的表現，黑線和紫線是 Linear Probing 的表現，我們發現 Uniform Probing 的表現比 Linear Probing 來得好，在設計雜湊函式的時候，可以盡量採用 Universal Probing。

## 8. 實作 Chaining

直接使用 STL 中的 vector 和 list 來實作 Chaining。

基本想法是一個雜湊表本身是向量 vector，但是向量中每個位置存的是鏈結串列 Linked list，這樣當不同的資料透過雜湊函式被轉換成同樣的索引值，它們就會被放到同一個鏈結串列裡，這樣一來每個插槽（slot）都可以放很多筆資料。

Chaining 版 Unordered Map	
1	// 引入 vector 和 list
2	#include <iostream>
3	#include <math.h>
4	#include <vector>
5	#include <list>
6	#include <stdlib.h>
7	using namespace std;
8	
9	template<...>
10	struct Data{...};
11	...

修改 Unordered\_Map 類別中的 Data

```
vector<list<Data<T1,T2>>>> data;
```

改寫後的 data 本身是一個向量，裡面每一個空間指向一個 Linked list，每個

Linked list 的節點資料則存放結構 Data。

#### Chaining 版建構式

```
1  template<typename T1,typename T2>
2  unordered_map<T1,T2>::unordered_map(int m){
3      len = m;
4      // 把 data 最外面的向量大小設成 m，總共有 m 個鏈結串列
5      data.resize(m);
6  }
```

#### Chaining 版中括號重載

```
1  T2& unordered_map<T1,T2>::operator[](T1 input){
2
3      int index = Hash_Func_Mul(input);
4
5      // 遍歷 data[index] 這個 Linked List
6      for(auto iter = data[index].begin();iter!=data[index].end();iter++){
7          // 比對 iter 中的人名與 input 是否相同
8          if((*iter).Key == input){
9              return (*iter).Value;
10         }
11     }
12
13     // 如果上面迴圈結束還沒有回傳
14     // 代表目前沒有 Key 為 input 的這筆資料
15     // 在 data[index] 這個 Linked list 後面加上新的資料
16     data[index].push_back(Data<T1, T2>{input,0});
17
18     // 回傳新增資料的 Value 成員
19     return data[index].back().Value;
20
21 }
```



測試 Chaining	
1	int main()
2	{
3	// 把大小改成 1，使得兩筆資料必定會發生碰撞
4	Unordered_Map<string,int> balance(1);
5	
6	balance["Mick"] = 50;
7	balance["John"] = 100;
8	
9	balance["Mick"] += 1;
10	cout << balance["Mick"] << endl;
11	cout << balance["John"] << endl;
12	
13	return 0;
14	}
執行結果	
51	
100	

加上 Chaining 後，即使發生碰撞仍然可以順利執行，兩筆資料可以被放在同一個 Linked list 中。

#### 第四節：Rehashing

本節來處理雜湊表的另一個挑戰：Rehashing

雜湊函式需要計算簡單：雜湊表的存在意義就是要減低搜尋的時間

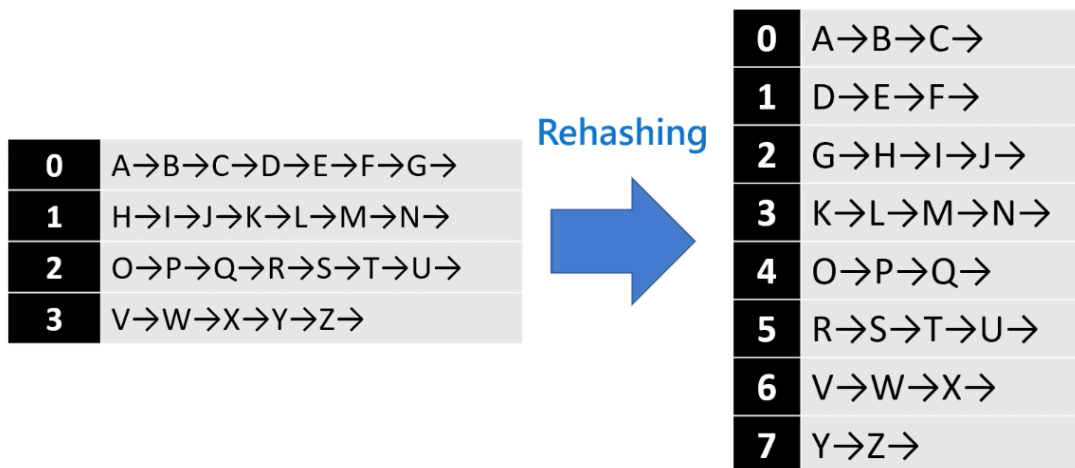
- A. 盡量避免碰撞 Collision Avoidance：減少碰撞發生的次數和機率
- B. 解決碰撞 Collision Resolution：碰撞仍然發生時採用的解決方式
  - a. Universal Hashing
  - b. Perfect Hashing
  - c. Rehashing：長度  $m$  不夠用

## 1. Rehashing

- A. 當不斷塞入資料導致原有的 Hash table 過小
  - a. 碰撞不停發生
  - b. 每個 slot 裡頭被塞入許多資料，導致效率不佳
- B. 重新配置 Hash table 大小
  - a. 把 hash table 的大小擴增為原有兩倍
  - b. 再把資料重新 hash 後移入

一開始幫雜湊表選擇了某個固定的大小，但是當資料不停的被塞入這個雜湊表中，會發生雜湊表「過小」的問題，此時碰撞會不停發生，效率也就減低。

一旦發現雜湊表的大小不夠用了，就應該重新配置雜湊表的大小，像 **vector** 一樣，當大小不夠用了，就把雜湊表的大小擴增成原本的「兩倍」，隨後需要重新計算目前所有資料在擴增後的雜湊表中新索引值，再一一把原有的資料放入。



舉個例子說明，原本的雜湊表的大小是 4，裡面只有 4 個插槽（slot），因為已經放入了許多筆資料，所以開始發生大量的碰撞，每個插槽中的 linked list 都很長。**Rehashing** 就是 **load factor** 太大、平均每個插槽分到的資料量太多時的解決方式，讓我們得以擴充雜湊表的大小。

使用 **Rehashing** 把雜湊表的大小擴增成兩倍的 8 後，Linked list 的平均長度就直接減半，因為 **load factor** 是  $\frac{n}{m}$ ，當插槽個數  $m$  變成兩倍，而資料筆數  $n$

不變時，**load factor** 自然就變為原本的  $\frac{1}{2}$ 。

## 2. 實作 Rehashing 函式

下面的實作中，當 `load factor > 5`（也可以取任意其他值）時就會自動呼叫 `Rehashing` 函式。

### Rehashing 函式的宣告

```
1  template<typename T1, typename T2>
2  class Unordered_Map{
3      private:
4          ...
5          // Rehashing 只需要由內部呼叫
6          void Rehashing();
7      public:
8          ...
9  };
```

### Rehashing 函式的定義

```
1  template<typename T1, typename T2>
2  void Unordered_Map<T1,T2>::Rehashing(){
3
4      // 建立一個新的兩倍大小的 vector
5      len *= 2;
6      vector<list<Data<T1, T2>>> new_data(len);
7
8      // 逐一把資料由原本的 vector 放入新 vector
9      for(int i=0;i<data.size();i++){
10
11          // 每個 data[i] 都是一個 linked list
12          for(auto iter = data[i].begin();iter!=data[i].end();iter++){
13
14              // 取得一筆資料的 key 和 value
15              T1 key = (*iter).Key;
16              T2 value = (*iter).Value;
17          }
```

18	// 計算新的雜湊值 index
19	int index = Hash_Func_Mul(key, len);
20	
21	// 把資料放到擴充後的陣列中
22	new_data[index].push_back(Data<T1, T2>{key, value});
23	
24	}
25	} // end of for
26	
27	// 把雜湊表改指向新的 vector
28	data = new_data;
29	
30	} // end of Rehashing

為了方便測試，另外寫一個 STL 裡沒有的 Print 函式：

Print	
1	template<typename T1, typename T2>
2	void Unordered_Map<T1,T2>::Print(){
3	
4	for(int i=0;i<data.size();i++){
5	// data[i]: linked list
6	cout << i << "-th:";
7	
8	for(auto iter = data[i].begin();iter!=data[i].end();iter++){
9	// 取出迭代器 iter 指到的 Key 與 Value
10	T1 key = (*iter).Key;
11	T2 value = (*iter).Value;
12	cout << "{" << key << "," << value << "}";
13	}
14	cout << endl;
15	}
16	}

### 3. 自動進行 Rehashing

接下來我們要實作一旦「Load factor > 5」就進行 Rehashing 的功能。

首先加上一個代表「資料個數」的成員 amount：

在 unordered_map 類別中加入 amount 成員	
1	template<typename T1, typename T2>
2	class unordered_map{
3	private:
4	int amount;
5	...
6	public:
7	...
8	};

再來，建構式中將 amount 初始化為 0（因為還沒有開始加入資料）：

修改 unordered_map 建構式	
1	template<typename T1, typename T2>
2	unordered_map<T1,T2>::unordered_map(int m){
3	amount = 0;
4	len = m;
5	data.resize(m);
6	}

每次新增一筆資料，就用 amount 記錄，並且檢查新增後 load factor 是否大於 5：

修改中括號 [] 運算子重載	
1	template<typename T1, typename T2>
2	T2& unordered_map<T1,T2>::operator[](T1 input){
3	...
4	data[index].push_back(...);

5	amount++;
6	// 新增資料前，先檢查新增後的 load factor
7	// 如果新增後 load factor 會超過 5 的話，事先進行 Rehashing
8	if (amount / len > 5){
9	// 進行 Rehashing
10	Rehashing();
11	// 取得要新增的資料的索引值
12	int index = Hash_Func_Mul(input);
13	}
14	
15	// 新增資料進原先（或者經過 Rehashing）的陣列
16	data[index].push_back(Data<T1,T2>{input,0})
17	return data[index].back().Value;
18	}

測試 Rehashing	
1	int main()
2	{
3	Unordered_Map<string,int> balance(1);
4	
5	balance["Mick"] = 50;
6	balance["John"] = 60;
7	balance["Rallod"] = 70;
8	balance["David"] = 80;
9	balance["Daphlene"] = 90;
10	
11	// load factor 正好等於 5，還沒有進行 Rehashing
12	balance.Print();
13	
14	// 加上第 6 筆資料後，會進行 Rehashing
15	balance["Wang"] = 100;
16	balance.Print();
17	
18	return 0;

19	
20	}
執行結果	
0-th:{Mick,50}{John,60}{Rallod,70}{David,80}{Daphlene,90}	
0-th:{Rallod,70}{David,80}{Wang,100}	
1-th:{Mick,50}{John,60}{Daphlene,90}	

原本 5 筆資料都被放在同一個 slot 裡，但是新增第 6 筆資料後，load factor 超過上限，slot 個數就透過 Rehashing 變成兩倍，資料也隨之被分別放到兩個 slot 中。

## 第五節：STL 中的雜湊表

在實作完雜湊表之後，緊接著來看 C++ STL 裡內建的雜湊表。之後讀者如果要使用雜湊表，可以直接使用 STL，不需另外用手刻。

在介紹 C++ STL 裡的雜湊表前，先介紹兩個資料結構：Map 和 Set，這兩種資料結構很容易混淆，需要特別區分。

### A. Map

- Key -> Value
- 資料結構：雜湊表或紅黑樹
- 常用來儲存「對應關係」

通常 Map 會被用來儲存對應關係，如每個客戶的存款餘額為多少、某個學生的考試成績為多少等，也就是把輸入 Key（鍵值）轉換成輸出 Value。Map 可以用雜湊表或紅黑樹（Red-Black Tree）實作出來。

### B. Set

- Value
- 資料結構：雜湊表或紅黑樹
- 常用來「分群」、「紀錄出現與否」

Set 當中只能儲存 Value，也就是說，通常用在判斷特定值存在與否，比如可以用一個 Set 來儲存所有「處理過的資料」。Set 一樣可以用雜湊表或紅黑樹實作，後續會提到如何分辨是用雜湊表還是或紅黑樹實現的。

## 1. C++ 中的 unordered\_map

### A. C++

a. unordered\_map 是雜湊表：插入、搜尋、刪除都為  $O(1)$

b. map 則「不是」雜湊表：插入、搜尋、刪除為  $O(\log_2 N)$

unordered map 中的「unordered」代表資料沒有次序之分，剛剛寫的雜湊表就沒有次序之分，所有的 input 都會被轉換成索引值，具體轉換成哪個索引值則不確定，尤其跟輸入順序「無關」，所以對於雜湊表而言，沒有所謂的「次序之分」，插入、搜尋、刪除這些處理都是  $O(1)$ 。

map 則「不是」雜湊表，而是有次序之分的。它是用二元平衡樹的「紅黑樹（Red-Black Tree）」實作而成，因為是用樹狀結構實現，所以插入、搜尋、刪除都是  $O(\log_2 N)$ ，原因會在「樹」的章節提到。

特別注意，C++ 中只有 unordered 開頭的資料結構才是用雜湊表實現的。

Python 裡常用的「字典 dictionary」資料結構也是雜湊表。

## 2. 比較 unordered\_map 與 map

	unordered_map	map
引用方式	#include<unordered_map>	#include<map>
原理	雜湊表	紅黑樹
優點	速度快	保留次序關係
缺點	沒有次序資料、空間需求更大（雜湊表中一定有空的空间）	佔用多餘空間
速度	較快， $O(1)$	較慢， $O(\log_2 N)$
適用	沒有次序的資料	有順序要求的資料



比較一下 `unordered map` 和 `map` 的各項特點，兩者的實現原理不同，但操作方式基本上是一模一樣的。

### 3. 使用 `map` / `unordered map`

#### (1) 引入函式庫

```
#include <map>
#include <unordered_map>
```

#### (2) 宣告：從 `Key(datatype_1)` 映射成 `Value(datatype_2)` 的型別

```
map<datatype_1, datatype_2> map_name;
unordered_map<datatype1, datatype_2> map_name;
```

#### (3) 迭代器

```
map<datatype_1, datatype_2>::iterator iter;
unordered_map<datatype_1, datatype_2>::iterator iter;
```

#### (4) 新增

```
map_name.insert(pair<datatype_1, datatype_2>(Key, Value));
map_name[Key] = Value;
```

要新增一筆資料時有兩種方式

##### A. 在 `map` 當中插入一個 `pair`

- a. 裡面有兩筆資料，`first` 和 `second`
- b. 對於 `map` 而言，`first` 是 `Key`，`Value` 就是 `Value`

##### B. 更簡便的方法是 `map_name[Key] = Value`。

### 4. 比較 `map` 與 `unordered_map` 的新增速度

剛剛有提到 `unordered map` 是  $O(1)$ ，`map` 則是  $O(\log_2 N)$ ，來測試一下這兩個速度的差異究竟是多少。

## 比較 map 與 unordered\_map 的新增速度

```
1  #include <iostream>
2  #include <time.h>
3  #include <map>
4  #include <unordered_map>
5  using namespace std;
6
7  int main()
8  {
9      clock_t start, finish;
10
11     // 計算 map 耗用的時間
12     map<int, int> data_map;
13     start = clock();
14     for(int i=0;i<10000000;i++)
15         data_map[i] = i;
16     finish = clock();
17     cout << "Time consumption of map: " << (finish - start) /
18     (double)CLOCKS_PER_SEC << " s." << endl;
19
20     // 計算 unordered_map 耗用的時間
21     unordered_map<int, int> data_unordered_map;
22     start = clock();
23     for(int i=0;i<10000000;i++)
24         data_unordered_map[i] = i;
25     finish = clock();
26     cout << "Time consumption of unordered map: " << (finish -
27     start)/(double)CLOCKS_PER_SEC << " s." << endl;
28
29     return 0;
30 }
```

## 執行結果

Time consumption of map : 4.256 s.

Time consumption of unordered\_map : 1.453 s.

從上面的執行結果可以發現，如果不需要次序資訊的話，應該優先使用 `unordered map`。

## 5. `map` / `unordered_map` 的操作

### (1) 搜尋 `find`

如果我們想要知道是不是有特定的 `Key` 在 `map` 當中，可以用 `find` 函式。根據「前閉後開」的規則，如果 `find(Key)` 回傳的 `iter`「不是」`map` 最後面的下一筆資料，就代表有找到 `Key` 這筆資料，如果「是」最後面的資料的後面，則代表「沒有找到 `Key` 這筆資料」。

當成功找到資料時，`iter` 指到的是一個 `map` 的元素，這個元素中 `first` 是 `Key`，`second` 是 `Value`，所以要取得 `Key`，就用 `iter->first`，要取得 `Value`，就用 `iter->second`。

搜尋 <code>find</code>	
1	<code>// 讓 iter 指到符合該 Key 的第一筆資料</code>
2	<code><u>iter = map_name.find(Key);</u></code>
3	
4	<code>// Key : first</code>
5	<code>// Value : second</code>
6	<code>if (iter!=map_name.end())</code>
7	<code>    cout &lt;&lt; "Value: " &lt;&lt; iter-&gt;second &lt;&lt; endl;</code>
8	<code>else</code>
9	<code>    cout &lt;&lt; "Not found in this map!" &lt;&lt; endl;</code>

### (2) 刪除特定項 `erase`

刪除特定項 <code>erase</code>	
1	<code>// 先用 find 找到元素的位置</code>
2	<code>iter = map_name.find(Key);</code>

3	
4	// 再用 erase 把該筆元素刪除
5	map_name.erase(iter);
6	
7	// 也可以使用一個 flag 來紀錄 erase 是否成功
8	// 成功時會回傳 true，失敗會回傳 false
9	bool flag = map_name.erase(Key);

### (3) 全部清空 clear

全部清空 clear	
1	// erase 也可以指定刪除區間
2	// 從第一筆刪掉最後一筆，等同於從 begin() 刪除到 end()
3	bool flag = map_name.erase(map_name.begin(), map_name.end());
4	map_name.clear();

### (4) 判斷是否為空 empty

判斷是否為空 empty	
1	// 如果是空的，回傳 true，如果不是則回傳 false
2	flag = map_name.empty();

### (5) 取出所有資料

取出所有資料	
1	// 把 map 中的元素一個個取出來放到 element 中
2	// first 是 Key、second 是 Value
3	for (auto& element : map_name){
4	cout << "Key: " << element.first << ", Value: " << element.second << "\n";
5	}

### (6) 其他操作

A. 設定雜湊表的槽數並搬遷資料：與剛才寫的 rehash 原理相同

```
map_name.rehash(Length);
```

- B. 擴充容量 `reserve`：確保在新增資料直到超過該容量前不需搬遷資料，節省不斷進行 `rehash` 花費的時間。

```
map_name.reserve(Length);
```

- C. 取出雜湊函式的函式指標

```
auto hash_func = map_name.hash_function();
```

- D. 取出比較 `key` 的函式指標

```
auto key_eq_func = map_name.key_eq();
```

#### (7) 使用 STL 中的 `unordered_map`

使用 STL 的 <code>unordered_map</code>	
1	<code>#include &lt;iostream&gt;</code>
2	<code>#include &lt;unordered_map&gt;</code>
3	<code>using namespace std;</code>
4	
5	<code>int main()</code>
6	<code>{</code>
7	<code>    // 宣告一個把字串映射到整數的 unordered_map</code>
8	<code>    // 字串（客戶名稱） -&gt; 整數（餘額）</code>
9	<code>    unordered_map&lt;string, int&gt; balance;</code>
10	
11	<code>    // 新增兩筆資料</code>
12	<code>    balance["Mick"] = 100;</code>
13	<code>    balance.insert(pair&lt;string, int&gt;("Rallod", 101));</code>
14	
15	<code>    // 印出目前的所有資料</code>
16	<code>    for(auto&amp; element : balance)</code>
17	<code>        cout &lt;&lt; "Key: " &lt;&lt; element.first &lt;&lt; ", Value: " &lt;&lt; element.second &lt;&lt;</code>
18	<code>        "\n";</code>
19	

20	// 查找特定鍵值 ("Mick") 是否在 unordered map 內
21	auto iter = balance.find("Mick");
22	if(iter != balance.end())
23	cout << "Value: " << iter->second << endl;
24	else
25	cout << "Not found in this map!" << endl;
26	
27	// 刪除特定鍵值 "Mick"
28	balance.erase("Mick");
29	
30	// 再查找一次特定鍵值 "Mick" 是否在 unordered map 內
31	iter = balance.find("Mick");
32	if(iter!=balance.end())
33	cout << "Value: " << iter->second << endl;
34	else
35	cout << "Not found in this map!" << endl;
36	
37	// 再印出目前的所有資料
38	for(auto& element : balance)
39	cout << "Key: " << element.first << ", Value: " << element.second <<
40	"\n";
41	
42	return 0;
43	}

#### 執行結果

Key: Rallod, Value: 101  
Key: Mick, Value: 100  
Value: 100  
Not found in this map!  
Key: Rallod, Value: 101

## 7. Google Code Jam

### A. 題目

有一張可以在雜貨店裡使用的儲值卡，餘額為  $C$  元。首先，走遍那家店，並且記錄下店裡全部  $L$  種商品的資訊，接下來，根據這個清單，要買兩樣商品來剛好花完儲值卡中的  $C$  元。

回傳值是該兩項商品的「位置」index，兩者中較小者先輸出。

### B. 輸入與目標輸出

輸入：

```
3
100
3
5 75 25
200
7
150 24 79 50 88 345 3
8
8
2 1 9 4 4 56 90 3
```

測試資料的第一行是「測資數」，代表整個題述的流程應進行  $N$  次。

每筆測資中有三行：

- A. 第一行為數字  $C$ ，也就是儲值卡中的餘額
- B. 第二行為數字  $L$ ，代表該商店中共有幾項商品
- C. 第三行為一串以空白分隔的  $L$  個整數，每個整數  $P_i$  代表商品的價格

經過設計，每筆測資一定會剛好有一組解。

範例輸入中，第一行的 3 代表總共做 3 輪。

每一輪有 3 行：

- A. 第一行是「手上有多少錢」：100、200、8
- B. 第二行是「有幾個物品」：3、7、8
- C. 第三行是「每個物品的價錢」

目標輸出：

Case #1: 2 3

Case #2: 1 4

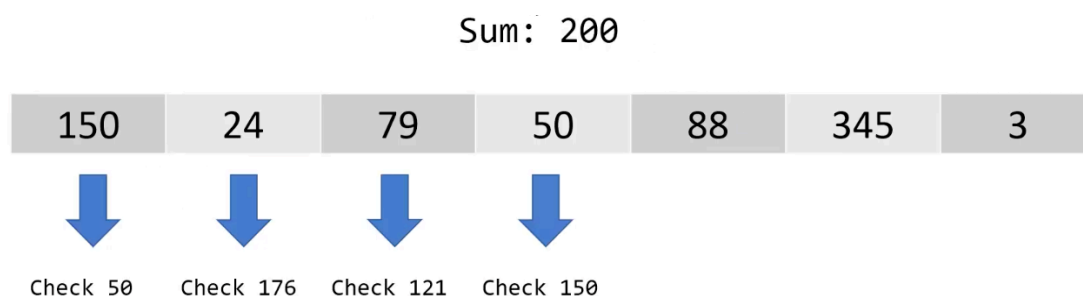
Case #3: 4 5

針對每筆測資，各輸出一行答案，格式如「Case #x:」後接該兩筆價格相加為 C 的商品的索引值，索引值小者先輸出。第一個 case 中要花 100 塊，而「第二個商品」和「第三個商品」分別為 75 和 25 塊，加起來正好符合，因此輸出 2 和 3。

### C. 解題邏輯

學雜湊表之前，這題最直觀的解法是寫兩個 for 迴圈，看哪兩個商品價格相加符合定好的數字，不過這樣的複雜度是  $O(n^2)$ 。

雖然外面那層迴圈跑到的每個數都只需要和自己後面的數字比，因此總比對次數僅為  $\frac{n^2}{2}$  次，但是  $O(\frac{n^2}{2})$  仍然是  $O(n^2)$ ，因此我們想透過雜湊表來尋找更快的方法。

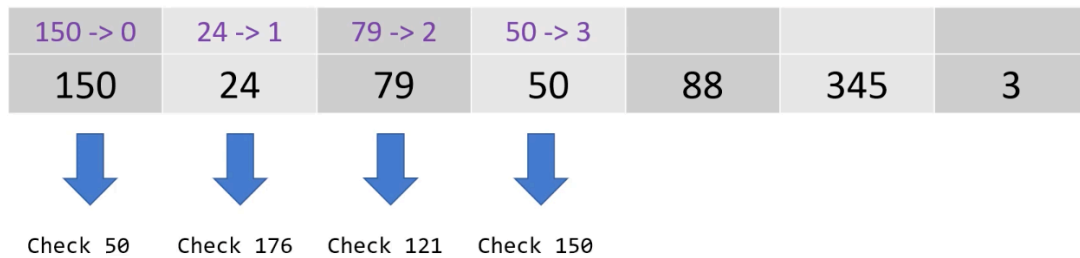


具體要怎麼做呢？以第二筆測資來舉例，若想知道哪兩個商品加起來是 200 塊，可以每遇到一筆資料，就把這筆資料和它所在的索引值記錄下來，這樣該



筆資料的「價格」就映射到「索引值」。

Sum: 200



這樣做可以有效減少比對的次數，比如遇到 150 的時候，只需檢查雜湊表裡有沒有 50，因為目前還沒有，所以把 150 映射成 0 並繼續往下處理，遇到 24 時，檢查雜湊表裡有沒有 176，因為也沒有，所以把 24 映射成 1，繼續往下進行。

直到處理到 50 時，發現雜湊表裡已經存在 150，這時去把 150 映射到的索引值取出來。過程中，檢查互補的數字是否在雜湊表中，與把該互補數字的索引值取出，兩個步驟都只需要  $O(1)$ ，兩者相加顯示檢查一筆資料需要  $O(1)$ 。

要處理完一筆測資，最差的情況下也只需要  $O(n)$ ，發生在答案的兩個商品中，其中一個在尾端的情形，複雜度成功由  $O(n^2)$  降為  $O(n)$ ，這是很大幅度的改進。

Google Code Jam	
1	#include <vector>
2	#include <unordered_map>
3	#include <stdlib.h>
4	using namespace std;
5	
6	int main()
7	{
8	// 利用讀寫檔方式取得測資
9	ifstream file("A-large-practice.in");
10	ofstream ofile("Testoutput.txt");
11	

```

12 // 存放資料的 vector
13 vector<int> data;
14 // 將商品價格映射到該商品索引的 unordered map
15 unordered_map<int, int> number_index;
16
17 // 當讀寫檔案成功時
18 if (file && ofile)
19 {
20     int dataCounts, totalPrice, stuffCounts;
21     // 取得總處理 case 數量
22     file >> dataCounts;
23
24     // 取得每個 case 對應到的三行資料
25     for (int i=0;i<dataCounts;i++){
26
27         file >> totalPrice >> stuffCounts;
28         data.resize(stuffCounts);
29         for (int j=0;j<stuffCounts;j++){
30             file >> data[j];
31         }
32
33         // 檢查總額減掉每筆資料的結果是否在 number_index 中
34         for(int j=0;j<stuffCounts;j++){
35
36             auto iter = number_index.find(totalPrice-data[j]);
37
38             // 如果迭代器不是 number_index.end()
39             // 代表有找到資料
40             if (iter != number_index.end()){
41
42                 // 輸出互補商品及目前商品索引值
43                 // 互補資料：number_index[totalPrice-data[j]] + 1
44                 // 目前資料：j+1
45                 ofile << "Case #" << i+1 << ": "

```

46	<< number_index[totalPrice-data[j]]+1
47	<< " " << j+1 << endl;
48	
49	// 也可以直接 cout 以檢查
50	cout << "Case #" << i+1 << ": "
51	<< number_index[totalPrice-data[j]]+1
52	<< " " << j+1 << endl;
53	
54	break;
55	}
56	
57	// 沒有找到互補商品的時候
58	// 把目前處理的 data[j] 存入 unordered map 中
59	// 且映射到索引值 j
60	else {
61	number_index[data[j]] = j;
62	}
63	
64	// 做完一個 case 後
65	// 清空 number_index 以便下一輪使用
66	number_index.clear();
67	}
68	} // end of for
69	} // end of if
	} // end of main
執行結果	
Case #1 : 2 3	
Case #2 : 1 4	
Case #3 : 4 5	
...	

這題的概念也被叫做「2-sum」，代表要找到哪兩個數字加起來是特定的數值，類似的還有「3-sum」、「4-sum」、...

## 8. LeetCode #217. 檢查重複元素 Contains Duplicate

### A. 題目

給定一個整數陣列 `nums`，如果陣列中任何值出現兩次以上，回傳 `true`；如果每個元素都不與其他元素的值重複，回傳 `false`。

B. 出處：<https://leetcode.com/problems/contains-duplicate/>

### C. 說明

在給定的 `vector` 中，看是否有重複來決定回傳值，有重複資料時回傳 `true`，沒有重複資料時回傳 `false`。

### D. 輸入與目標輸出

#### a. 範例一

輸入：`nums = [1,2,3,1]`

輸出：`true`

#### b. 範例二

輸入：`nums = [1,2,3,4]`

輸出：`false`

檢查重複元素 Contains Duplicate	
1	<code>class Solution{</code>
2	<code>public:</code>
3	
4	<code>bool containsDuplicate(vector&lt;int&gt;&amp; nums){</code>
5	
6	<code>// 宣告一個 unordered map 來存放已經出現過的數字</code>
7	<code>unordered_map&lt;int, bool&gt; existed;</code>
8	
9	<code>// 逐個檢查 nums 中的數字</code>

10	for(int num: nums){
11	auto iter = existed.find(num);
12	if (iter!=existed.end()){
13	// 數字已經出現過
14	return true;
15	}
16	
17	// 現在的數字還沒出現過時，加到 unordered map 中
18	existed[num] = true;
19	}
20	// 檢查完整個 vector 都沒有發現重複的數字
21	return false;
22	}
23	};

## 8. LeetCode#219. 檢查重複元素 II Contains Duplicate II

### A. 題目

給定一個整數陣列 `nums` 與一個整數 `k`，在下列條件滿足時，回傳 `true`：  
存在「`i` 不等於 `j`」，且「`(i-j 的絕對值) <= k`」，使得 `nums[i] == nums[j]`

B. 出處：<https://leetcode.com/problems/contains-duplicate-ii/>

### C. 說明

這題是檢查有沒有重複數字在陣列中的修改版，給出一個數字 `k`，要檢查在 `k` 的距離內，有沒有任意兩元素重複。

### D. 輸入與目標輸出

#### a. 範例一

輸入：`nums = [1,2,3,1], k=3`

輸出：true

b. 範例二

輸入：nums = [1,0,1,1], k=1

輸出：true

c. 範例三

輸入：nums = [1,2,3,1,2,3], k=2

輸出：false

範例一中，k 是 3，所以要檢查每個數字的前後三個數字有沒有與其重複，1 跟 1 正好距離 3，所以有重複，回傳 true。範例三中，因為 k 是 2，而每個重複的數字的距離都是 3，所以看作沒有重複，回傳 false。

E. 解題邏輯

這題只要把每個數值上一次出現的位置記錄下來即可，並把目前處理的數字，和「同樣數值」上次出現的位置做比較，如果距離在 k 以內，就回傳 true，否則更新這個數值最近一次出現的位置。

比如範例三中，第一次遇到 1 的時候，把 1 的位置紀錄成索引值 0，遇到第二個 1 的時候，看上一次出現的索引值 0 與這次出現的索引值 3 距離是否在 k=2 之內，因為沒有，所以改把 1 映射成新的值索引值 3。

這個雜湊表會不斷更新，每遇到一個數字，都和之前的資料比較，根據兩者的距離，決定直接回傳 true，或者更新雜湊表資料。

Contains Duplicate II	
1	class Solution{
2	public:
3	
4	bool containsNearbyDuplicate(vector<int>& nums, int k){
5	
6	// 宣告一個 int 映射到 int 的 unordered map

```

7      // 前面的 int 是數值，後面的 int 指的是上次出現的位置
8      unordered_map<int, int> index;
9
10     // 用迴圈遍歷向量 nums
11     for (int i=0;i<nums.size();i++){
12
13         // 檢查 nums[i] 這個數字是否已經在 index 中出現過
14         auto iter = index.find(nums[i]);
15
16         if (iter!=index.end()){
17             // 如果有找到，比對距離是否超過 k
18             if (i-index[nums[i]] <= k)
19                 // 距離不超過 k，回傳 true
20                 return true;
21             else
22                 // 距離大於 k，視為沒有重複
23                 // 更新目前處理數字 nums[i] 的索引值即可
24                 index[nums[i]] = i;
25         }
26         else
27         {
28             // 沒有找到時代表第一次出現，把位置加到 index 中
29             index[num[i]]-i;
30         }
31
32         // 處理到陣列尾端都沒有發生「距離在 k 之內」的情形
33         // 回傳 false
34         return false;
35
36     } // end of for
37
38     } // end of containsNearbyDuplicate
39 }; // end of class Solution

```

## 9. LeetCode#242 字串變形 Valid Anagram

### A. 題目

給定兩個字串  $s$  和  $t$ ，如果  $t$  是  $s$  的一個變形，回傳 `true`，否則回傳 `false`。

B. 出處：<https://leetcode.com/problems/valid-anagram/>

### C. 說明

給定兩個字串，決定字串  $s$  和字串  $t$  是否是由一樣的字元構成，只是順序有差別。比如 `anagram` 這個詞中，`a` 出現 3 次，`n`、`g`、`r`、`m` 各出現 1 次，而另一個字串如果同樣是由 3 個 `a` 與 `n`、`g`、`r`、`m` 各 1 個，以某種順序構成，我們就把後面這個字串看成前一個字串的變形，回傳 `true`，如果字母的頻率分佈不同，則回傳 `false`。

### D. 輸入與目標輸出

#### a. 範例一

輸入：`s = "anagram", t = "nagaram"`

輸出：`true`

#### b. 範例二

輸入：`s = "rat", t = "car"`

輸出：`false`

### E. 解題邏輯

試著紀錄每個字元出現的次數，來確認兩個字串中字元出現的次數分布是不是完全相同，就可以決定回傳值。

字串變形 Valid Anagram	
1	<code>class Solution{</code>
2	



3	public:
4	bool isAnagram(string s, string t){
5	
6	// 如果兩個字串長度不同，一定不符合條件
7	if (s.size() != t.size())
8	return false;
9	
10	// 宣告 unordered map 紀錄每個字元出現次數
11	unordered_map<char, int> counts;
12	
13	// 依序處理 s 中的每個字元
14	for (char c:s){
15	auto iter = counts.find(c);
16	if (iter!=counts.end())
17	// counts 中已經出現過 c 這個字元
18	// 把出現次數加一
19	counts[c]++;
20	else
21	// counts 還沒出現過 c 這個字元
22	// 把出現次數設定為 1
23	counts[c] = 1;
24	}
25	
26	// 依序處理 t 中的每個字元
27	for (char c:t){
28	auto iter = counts.find(c);
29	if (iter!=counts.end())
30	// t 中每出現任一字元 c
31	// 把 counts 中 c 對應到的數字減一
32	if(counts[c]>0)
33	counts[c]--;
34	else
35	// s 中出現過的次數已經被用完了
36	return false;

37	else
38	// c 沒有出現在 counts 中
39	// 代表 t 中出現了 s 裡沒有的字元
40	return false;
41	}
42	
43	// 每個字元都符合條件，回傳 true
44	return true;
45	
46	} // end of isAnagram
47	}; // end of class Solution

## 10. LeetCode#525. 連續陣列 Contiguous Array

### A. 題目

給定一個只含 0 與 1 的陣列 nums，回傳所有符合「含有 0 與 1 的數目相同」條件的子陣列中，最長者的長度。

B. 出處：<https://leetcode.com/problems/contiguous-array/>

### C. 說明

要找出的是「0 和 1 出現個數相同」的子陣列，並回傳這個子陣列的長度。如果有多個子陣列符合條件，回傳的是其中「長度最長者」的長度。

### D. 輸入與目標輸出

#### a. 範例一

輸入：nums = [0,1]

輸出：2

#### b. 範例二

輸入：nums = [0,1,0]

輸出：2


在範例二中，前面的 [0,1] 和後面的 [1,0] 都符合條件，回傳其中一個的長度 2。

#### E. 解題邏輯

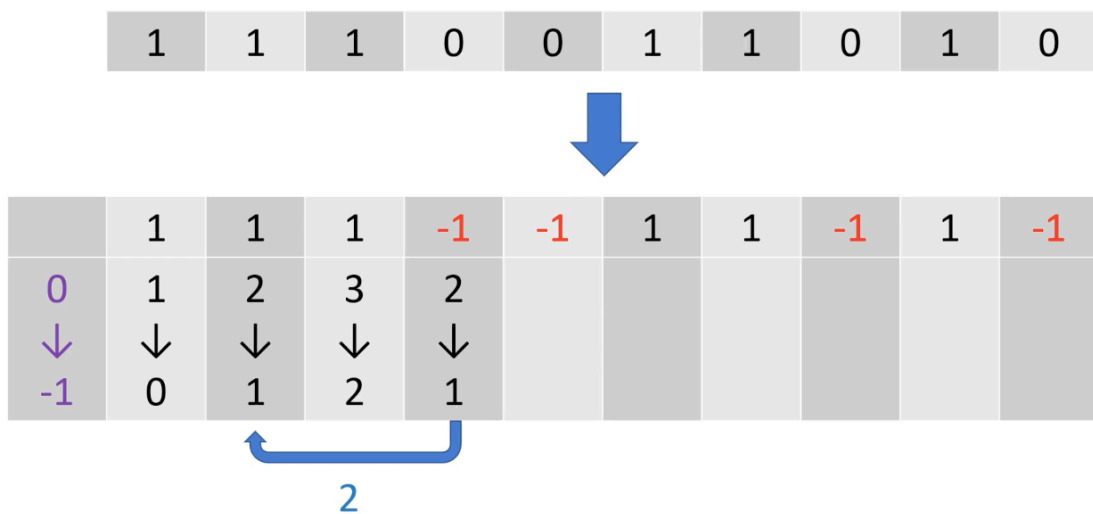
可以做一個轉換來讓檢查 0 和 1 出現次數是否相同的過程變得較為簡便：將所有的 0 都視為 -1，這樣一旦某個子陣列中 0 和 1 出現次數相同，比如 0 出現 5 次，1 也出現 5 次，這段子陣列的加總就會是

$$(-1) \times 5 + 1 \times 5 = 0$$

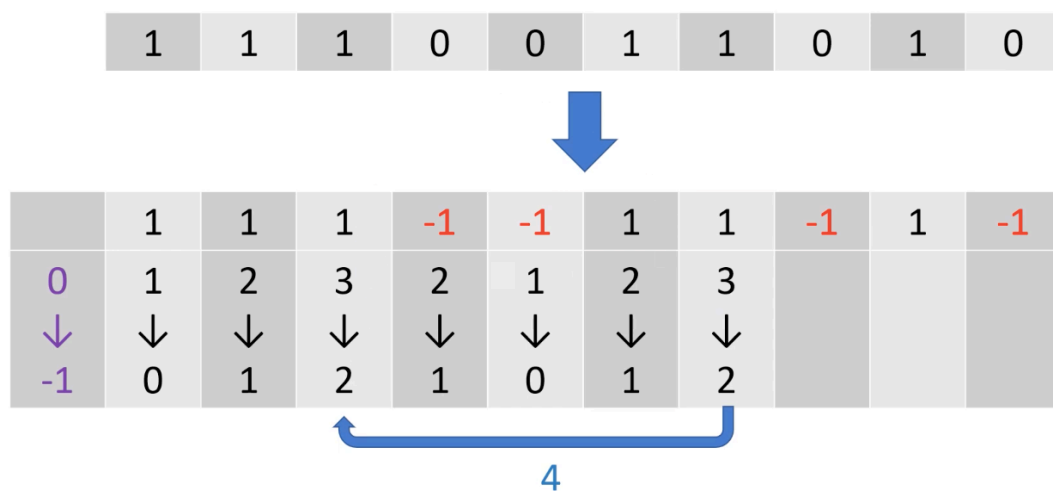
這樣一來，當任何一段子陣列開頭與結尾計算出的（從開頭開始的）加總一樣時，就代表這段子陣列符合條件。

	1	1	1	0	0	1	1	0	1	0
										
	1	1	1	-1	-1	1	1	-1	1	-1
0	1	2	3							
↓	↓	↓	↓							
-1	0	1	2							

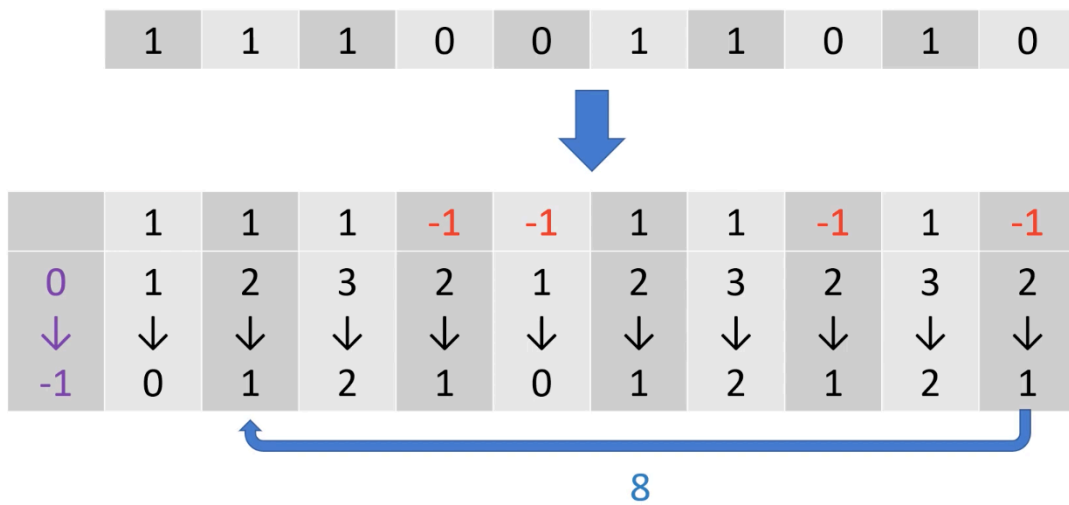
開始處理之前，先把總和 sum 設為 -1，避免開頭就誤認出現符合條件的子陣列的情形，接下來，每遇到一個數字就加上 1 或減去 1。



上面的例子中，處理到第三個 1 時，總和 sum 的值為 2，如果往後面繼續處理若干個數字後，又出現 sum 的值為 2 的情形，比如第五個 1 時 sum 正好又回到 2，就代表中間這段 [0 0 1 1] 的子陣列中 0 和 1 出現的次數相同。



這樣一來，只要記住 sum 到達過的每個數值最早出現的位置，就可以透過比對現在出現的數值，和同樣數值最早出現的位置間的距離，來決定符合條件的最長陣列長度為何。



處理到結尾時，發現  $sum = 1$  與處理到陣列中第二個 1 時的值相同，代表 [1 0 0 1 1 0 1 0] 是符合條件的子陣列。

連續陣列 Contiguous Array	
1	class Solution{
2	public:
3	
4	int findMaxLength(vector<int>& nums){
5	
6	// 宣告一個 unordered map
7	// 紀錄某個數字上一次出現在索引值多少
8	// 第一個 int 是數值，第二個 int 是上次出現的索引值
9	unordered_map<int, int> data;
10	
11	// 將開頭設定為 -1
12	data[0] = -1;
13	
14	// 儲存目前的數字加總
15	int sum = 0;
16	// 目前符合條件的最長的陣列長度
17	int longest = 0;
18	
19	// 依序處理 num 中的數字

20	for(int i=0;i<num.size();i++){
21	
22	// nums[i] 是 1
23	if (nums[i])
24	sum++;
25	// nums[i] 是 0，看成 -1
26	else
27	sum--;
28	
29	// 檢查目前為止的 sum 數值是否出現過
30	auto iter = data.find(sum);
31	
32	// 目前數值已經出現過
33	if (iter!=data.end()){
34	// 比較最長的子陣列長度 longest
35	// 與這次發現的子陣列 i-data[sum] 長度哪個較大
36	longest = longest > i-data[sum] ? longest : i-data[sum];
37	}
38	
39	// 若目前的 sum 值第一次出現：把 sum 對應目前索引值
40	else {
41	data[sum] = i;
42	}
43	
44	// 回傳得到的最長子陣列長度
45	return longest;
46	
47	} // end of for
48	
49	} // end of findMaxLength
50	}; // end of class Solution

因為處理每個數字的複雜度都是  $O(1)$ ，且總共執行陣列長度  $n$  次，上面這個算法的複雜度是  $O(n)$ 。