**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

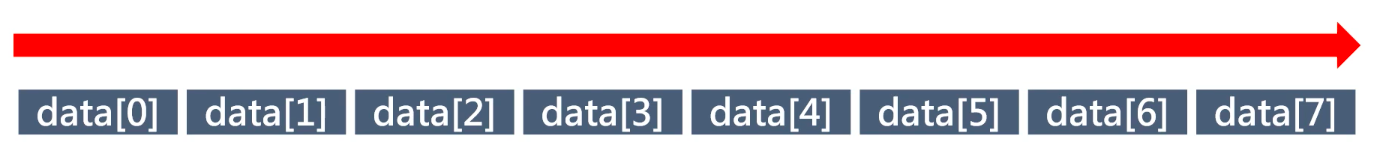
C. 碰撞 Collision 處理

D. 雜湊表的挑戰

E. Map 與 Dict

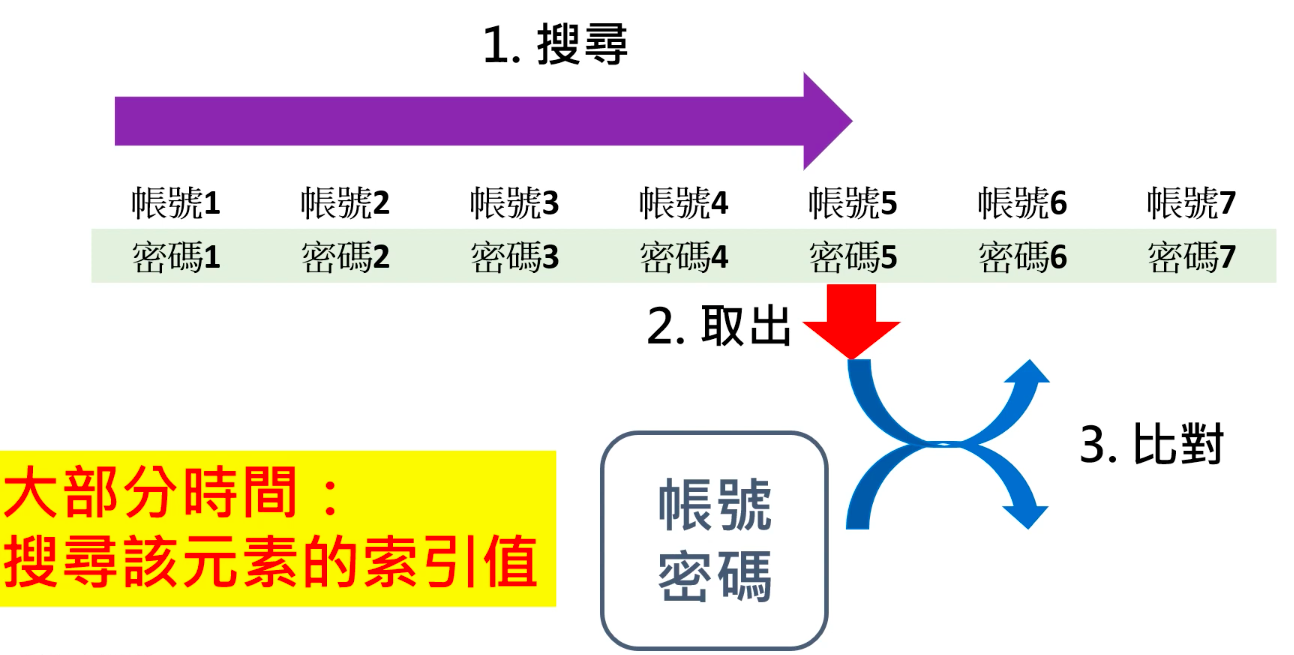
**第一節：雜湊表簡介**

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 。

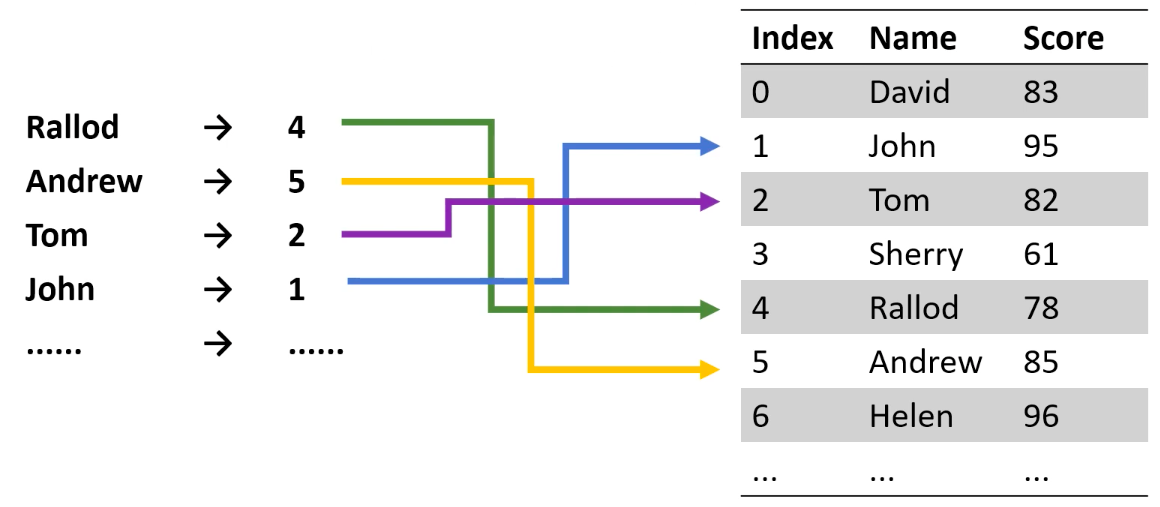


|  |  |
| --- | --- |
| 在陣列中尋找特定資料 | |
| 1  2  3  4  5  6  7  8 | int search(int \*p, int len, int value)  {  for(int i=0;i<len;i++){  if(\*(p+i)==value)  return i;  }  return -1;  } |

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 ，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 ，而非之前的 。

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：

index = hash(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

雜湊函式把宇集合 U（任意輸入）轉換成 的整數。

（多對一函式）

注意到，因為宇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

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 降成 ，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

透過把資料 key 對 m 取餘數的做法，將所有輸入壓回到 。

(1) Division 的優缺點

A. 優點：計算快速簡便

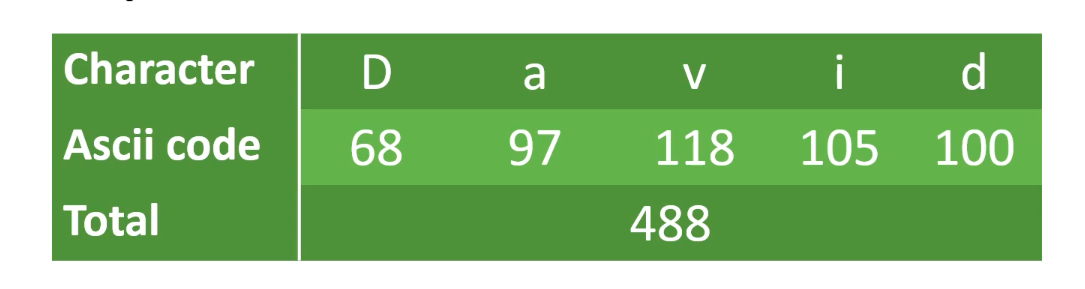
B. 缺點：很難決定如何取 m

a. 如果 m 取很大， 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



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。

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

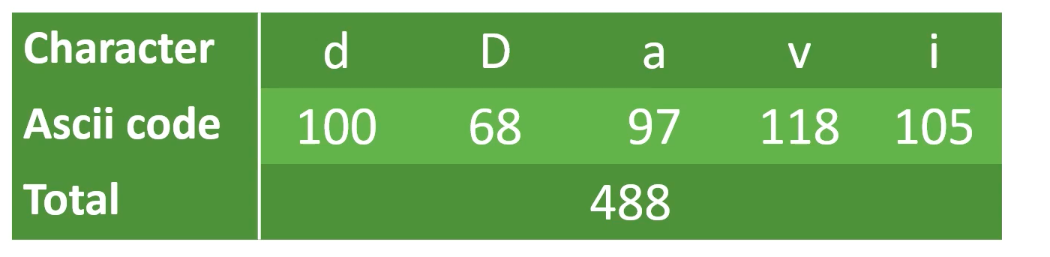
A. 把輸入字串中的每個字元轉換成 Ascii 碼（整數）

B. 把得到的所有整數加總

C. 把加總得到的值對陣列長度（上例中為8）取餘數

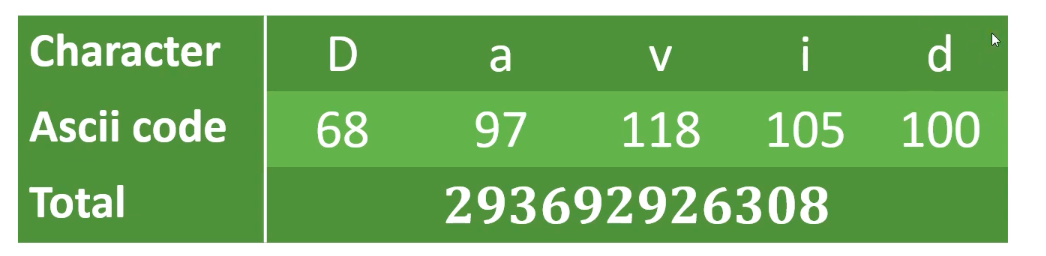
D. 得到的餘數就是索引值

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。

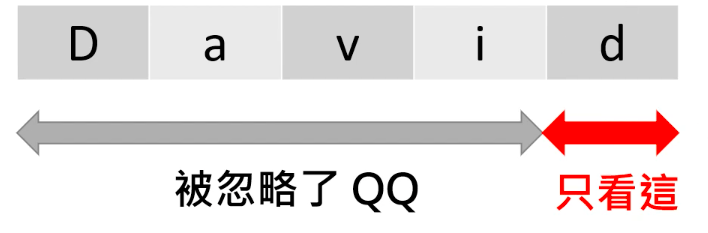


(4) 考慮字母順序

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（）的不同次方，再對 8 取餘數。



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 ，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

B. n = 3 時：

由上可知，m 盡量不要等於 ，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 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 ）

A. 選擇一個常數 C，

B. 將輸入的鍵值 Key 與 C 相乘，得到

C. 取得 的小數點後部分 ，其中

（這步的目的是把 的值壓到 0 到 1 之間）

D. 把 frac 與 m 相乘，得到

（這步的目的是把 映射到 ）

E.

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 Key（剛剛的例子裡，就是人名 David，也就是要轉成索引值的輸入資料）。

下一步是取出 的小數部分，以數學方式表達就是把 減去 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 的整數部分。

範例（假定 , ）

A. 任意選擇常數 ）

B. 將輸入的鍵值 Key 與 C 相乘，

C. 取得 的小數點後部分

D. 把 frac 與 m 相乘，

E.

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：

其中 指的是 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 ，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，

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  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | #include <iostream>  using namespace std;  // input 是 Key，m 是陣列長度  int Hash\_Func\_Div(string input, int m){  // sum 使用 long long int 型態，可以存放的數字較大  long long int sum = 0;  // 把輸入 input 中的每個字元取出來  for(char c:input){  // sum 是每個字元 ascii 碼的總和  // 因為 sum 是 int，c 會被自動轉換  sum += c;  }  // 回傳值是 sum 對 m 取餘數  return sum%m;  }  int main()  {  // 讓使用者輸入一個字串 str  string str;  cout << "Please enter a str:" << endl;  cin >> str;  // 輸出經過雜湊函式轉換得到的索引值，假設陣列長度是 8  cout << "Index:" << Hash\_Func\_Div(str, 8);  return 0;  } |
| 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  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | int Hash\_Func\_Mul(string input, int m){  // 一樣用 sum 來存放加總結果  long long int sum = 0;  // 常數 c 採用剛才介紹過的值  double c = (sqrt(5)-1)/2.0;    // 讓每個字元乘上 tmp（256 的次方數）以把字元次序納入考慮  int tmp = 1;  for(char ch:input){  sum += tmp \* ch;  tmp \*= 256;  }    // 取出 c\*sum 的小數部分  double frac = c\*sum - (int)(c\*sum);    // 把 frac 從 0 到 1 之間映射到 0 到 m-1  int index = frac \* m;  return index;  }  int main()  {  // 讓使用者輸入一個字串 str  string str;  cout << "Please enter a str:" << endl;  cin >> str;  // 輸出經過雜湊函式轉換得到的索引值，假設陣列長度是 8  cout << "Index:" << Hash\_Func\_Mul(str, 8);  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 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

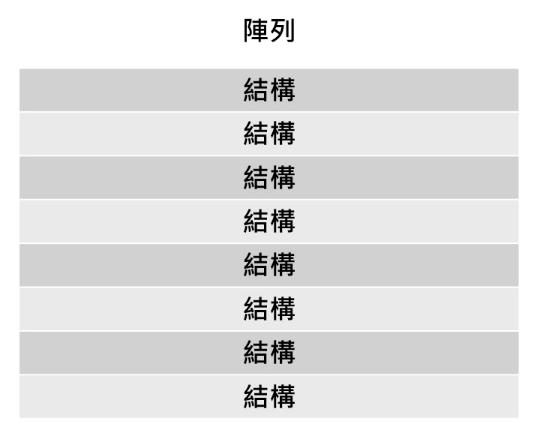
Key：客戶姓名，Value：存款餘額

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



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 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  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | #include <iostream>  #include <math.h>  #include <stdlib.h>  using namespace std;  // 存放客戶姓名和餘額的結構  // 目標是透過雜湊函式轉換，只要傳入 Key 就回傳 Value  template<typename T1, typename T2>  struct Data{  T1 Key;  T2 Value;  };  // 目標是用 Balance["Rallod"] 的方式取出 Rallod的存簿餘額  // 需要重載中括號 [] 運算子  template<typename T1, typename T2>  class Unordered\_Map{  private:  // 宣告指向 Data 結構的 Pointer  Data<T1,T2>\* Pointer;  int len;  // 利用模板撰寫，雜湊函式的傳入參數型態為 T1  // 函式回傳值（資料所在的索引值）型態為 int  int Hash\_Func\_Div(T1);  int Hash\_Func\_Mul(T1);  public:  // 重載中括號運算子  T2& operator[](T1);  // 建構式：預設長度是 128  Unordered\_Map(int=128);  }  … // 函式定義在這裡 |

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

|  |  |
| --- | --- |
| 重載中括號 [] 運算子，假設傳入資料都是字串 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | template<typename T1, typename T2>  T2& Unordered\_Map<T1,T2>::operator[](T1 input){  // 使用 Multipliction 雜湊函式取出 input 對應到的索引值  int index = Hash\_Func\_Mul(input);  // 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  Pointer[index].Key = input;  // 中括號用作查詢時：取得陣列中索引值 index 處的資料並回傳  return Pointer[index].Value;  } |

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

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

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

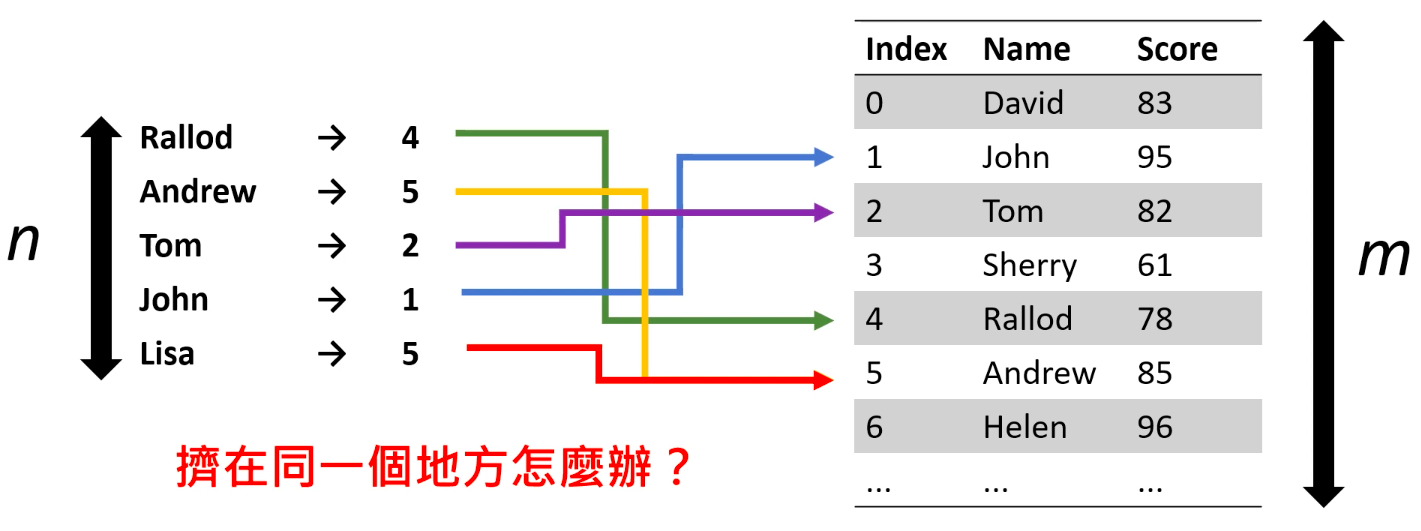
**第三節：碰撞處理**

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：輸入是宇集合，所以輸入會有無限多種狀況，輸出則是 ，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 越大的時候，代表資料越多，當 接近 0 的時候，代表裡面沒什麼資料。

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， 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。

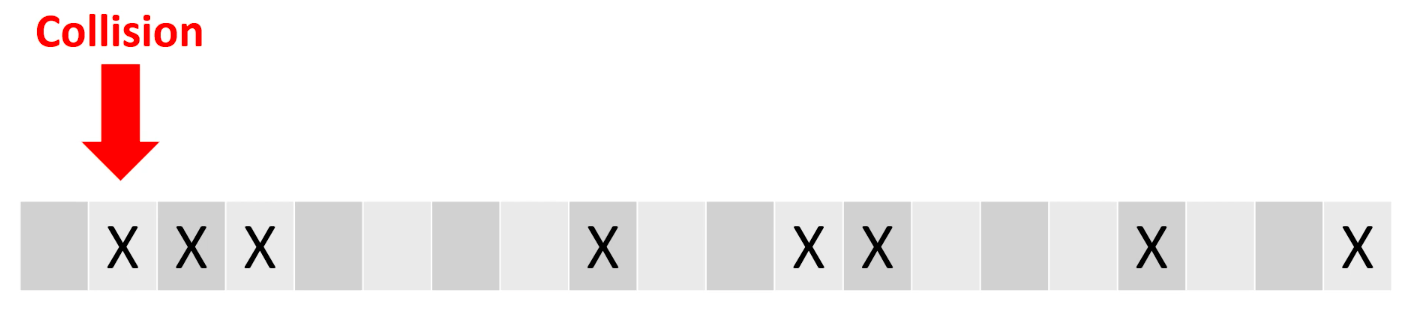
(1) Linear Probing

Open Addressing：Linear Probing

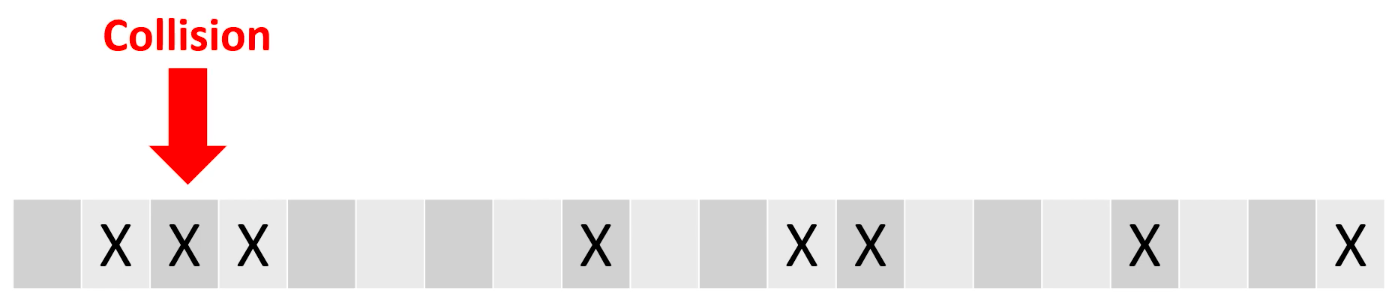
i = collision times = 0

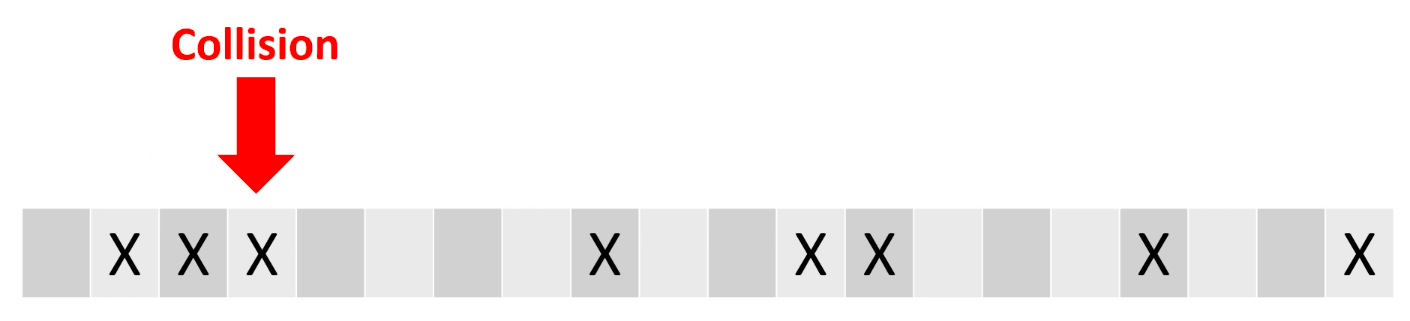
在 linear probing 中，想知道某個 key 值對應到的索引值的話，要多加一個參數 i，i 代表的是「碰撞次數」。

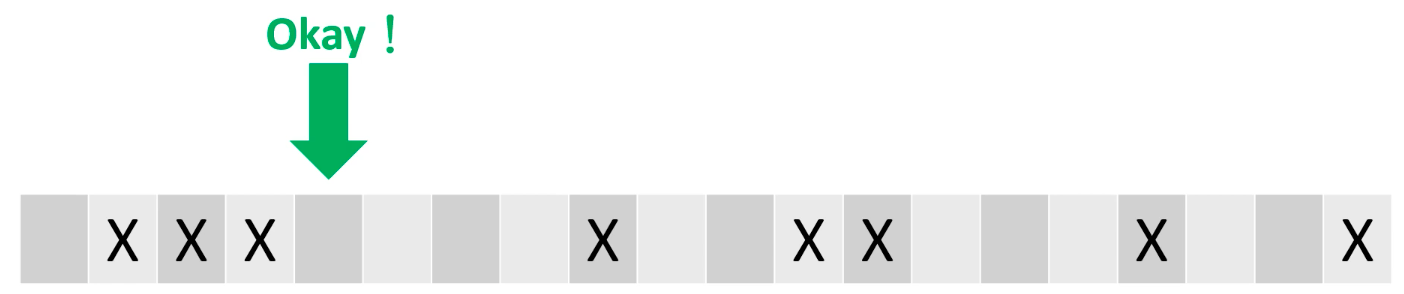
怎麼知道鍵值碰撞 i 次之後的雜湊值應該是多少呢？其實就是沒有發生任何碰撞時的索引值 再加上 i，代表碰撞第一次的時候會右移一格，又碰撞第二次的話，就再右移一格。



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。



然而右移之後，裡面仍然有資料，那就再往右一格。

因為裡面還是有資料，所以再往右一格。

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。

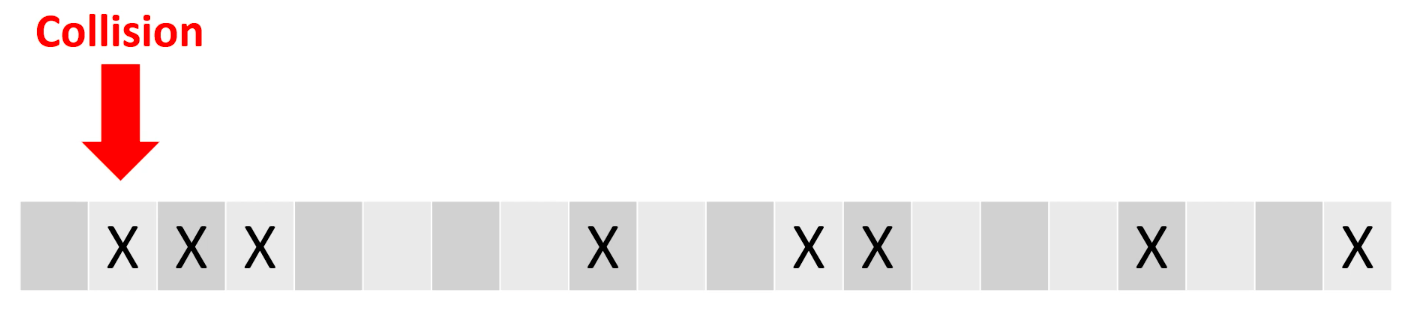
這要如何解決呢？可以簡單改一下 i，讓 i 代表的意義不要是線性的，而是一個二次函式 ，每次要往右移動幾格都由 決定，係數 a 跟 b 的值可以自由決定。

(2) Quadratic Probing

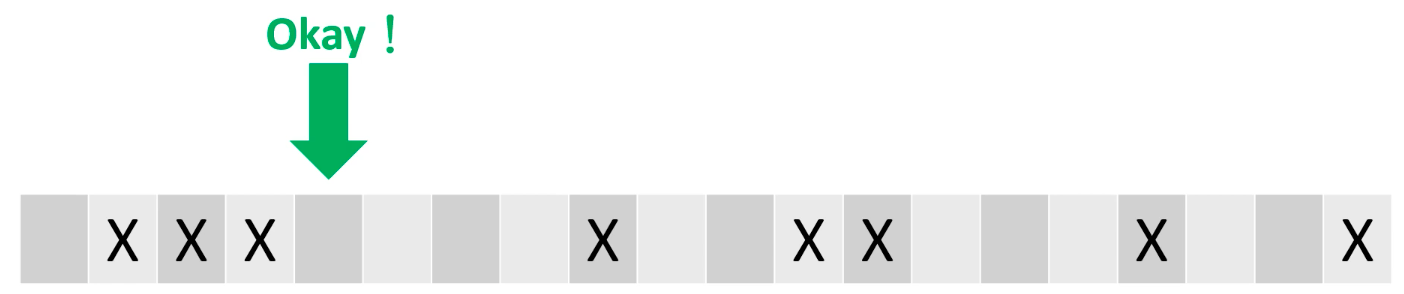
Open Addressing: Quadratic Probing

i = collision times = 0

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。



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

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 （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

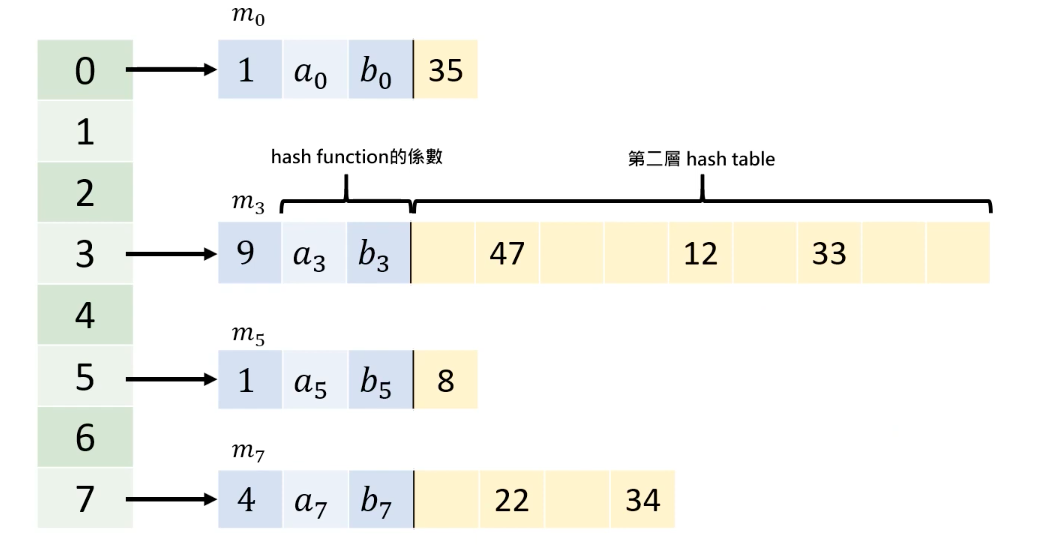
a. 第一次 hash 到的是另一個 hash table

b. 第二次 hash 到的才是真正的 value

B. Worst case 下的運算複雜度為

C. 可透過挑選適合的 hash function 使得空間複雜度仍為

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 ，因為第一層雜湊的轉換複雜度是 ，第二層雜湊的轉換複雜度也是 ，兩個 的操作前後接在一起仍然是 ：。

(3) Perfect Hashing 的碰撞機率

讓 可以確保：碰撞機率 < 1/2。

證明

A. 總共有 對鍵值可能碰撞

B. 每對鍵值以 的機率碰撞

總共有 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 ，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 ，而索引值 3 裡放的資料有 3 筆，對應到的陣列長度是 ，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 ，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 筆資料

c. 每個第二層雜湊表長度需要

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

約

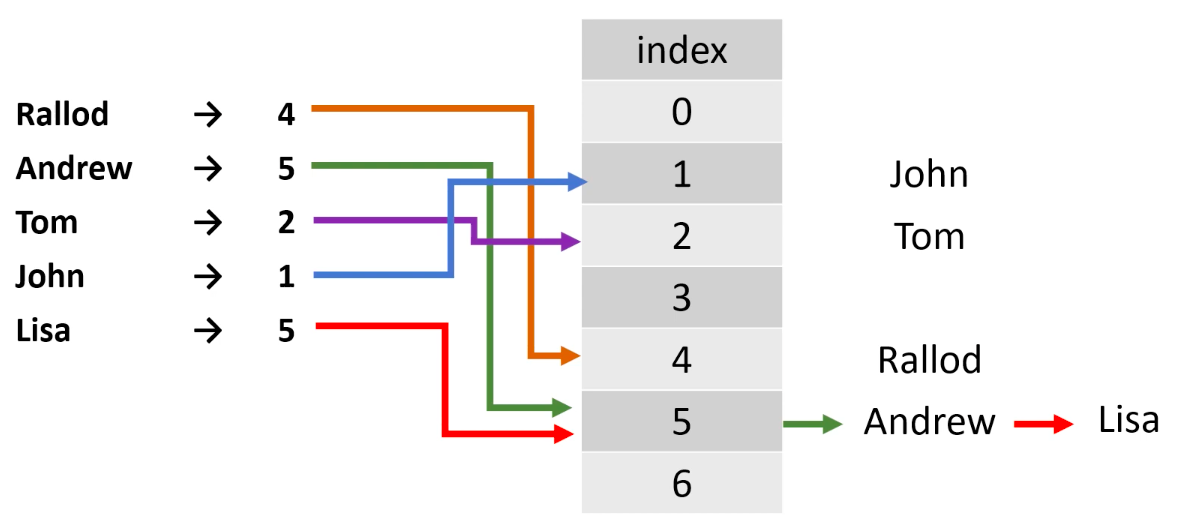
6. Chaining

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 ，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 。

Chaining 的時間複雜度平均而言是 ，其中 。

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. 是先把輸入值轉成索引值

C. 是遍歷一個長度 α 的 Linked List

當 Load factor 的值不大時，α 是一個較小的值，。

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

α: load factor

|  |  |  |  |
| --- | --- | --- | --- |
|  | Open Addressing Uniform Probing | Open Addressing Linear Probing | Chaining |
| Successful Search |  |  |  |
| Unsuccessful Search |  |  |  |

(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. 當一個位置滿了就去找下一個位置

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 時，：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)...，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：α

b. 尚未有資料的機率：

有 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 α，而挑到還沒被佔用的位置的機率則是 。

C. Unsuccessful Search 的複雜度：

, 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 α 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 α，第二次發生碰撞的機率為 ，所以

所有 Key 都要做第一次搜尋：1

+ 比例的 Key 要做第二次搜尋：

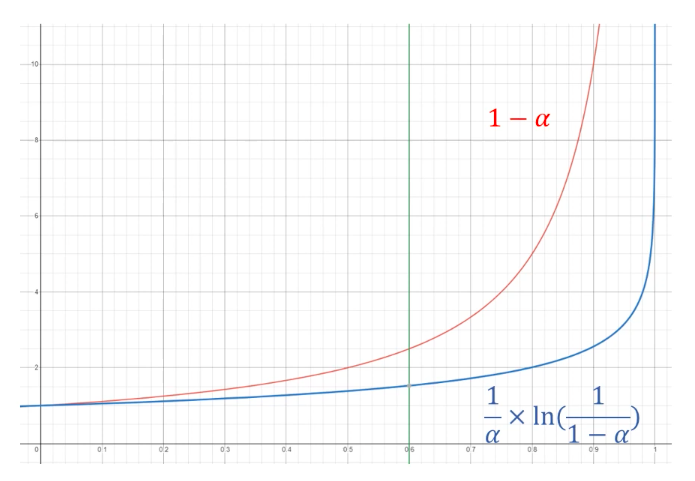
+ 比例的 Key 要做第三次搜尋：

+ 比例的 Key 要做第四次搜尋：

... ...

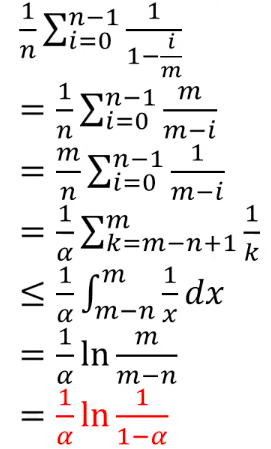
預期需要做的搜尋次數就是等比級數公式：

，也就是 ，時間複雜度是 。



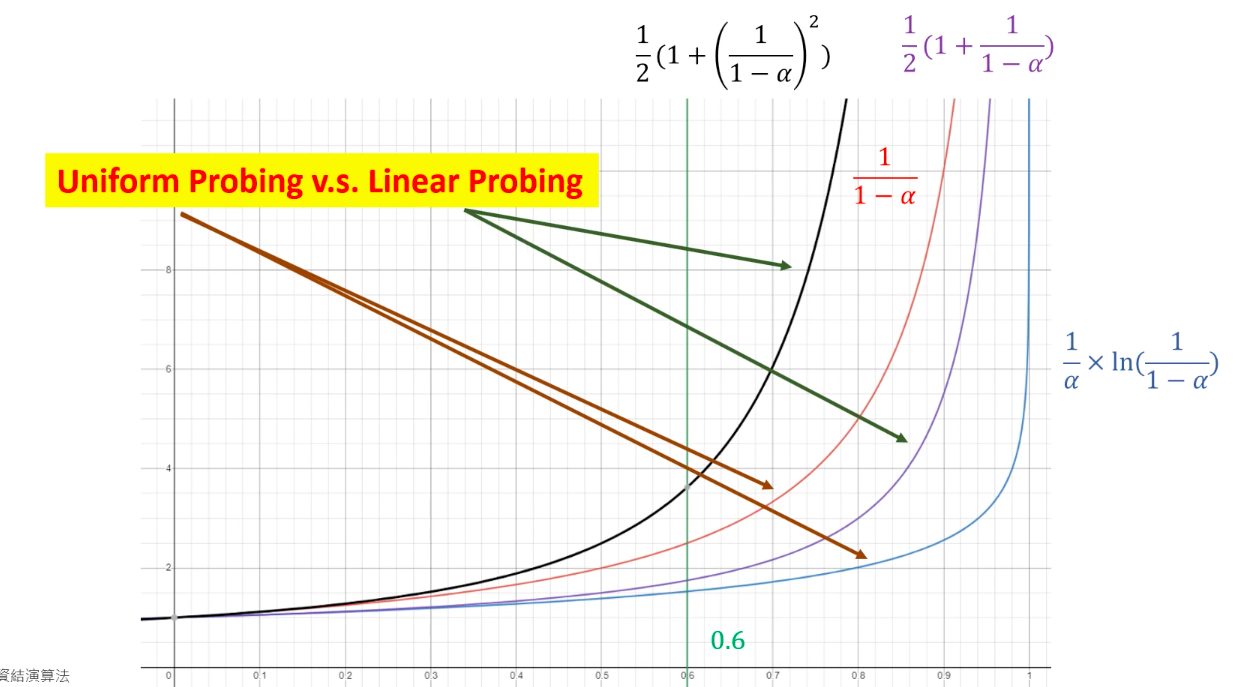
從圖上可以看出（橫軸是 Load factor ，縱軸是需要進行的搜尋次數），當 的值小於 0.6 時， 的值是可以接受的。

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

對於第 筆插入到雜湊表中的資料來說

最前面的 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 加起來之後，再除以 n。

第 i 次插入沒有發生碰撞的機率是 ，中間有一些積分的計算我們不細講，實務上只要知道當 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  2  3  4  5  6  7  8  9  10  11 | // 引入 vector 和 list  #include <iostream>  #include <math.h>  #include <vector>  #include <list>  #include <stdlib.h>  using namespace std;  template<...>  struct Data{...};  … |

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

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

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

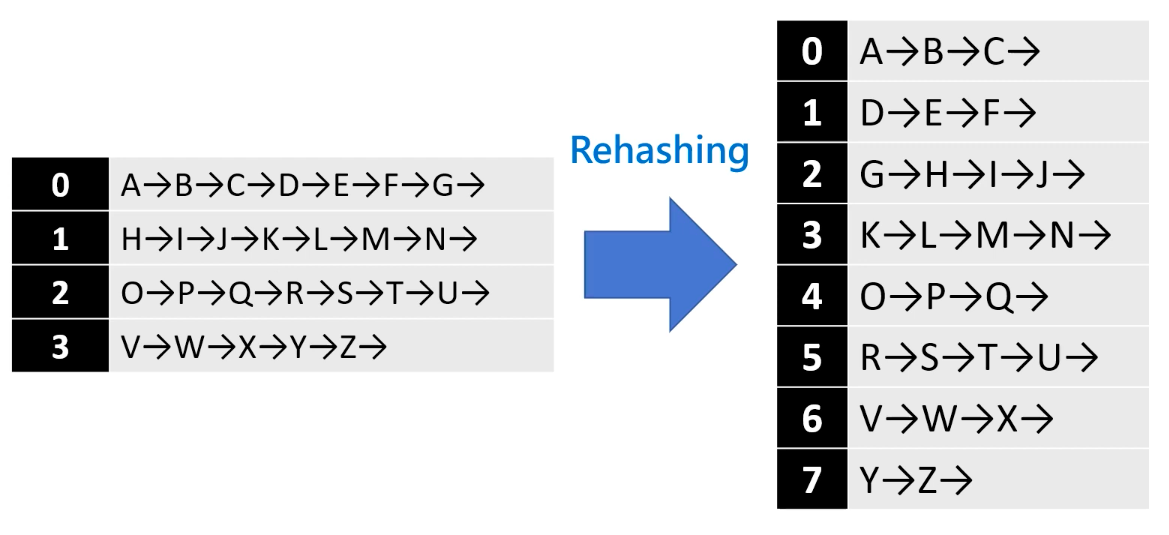
b. 每個 slot 裡頭被塞入許多資料，導致效率不佳

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 ，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 。

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

|  |  |
| --- | --- |
| Rehashing 函式的定義 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | template<typename T1, typename T2>  void Unordered\_Map<T1,T2>::Rehashing(){  // 建立一個新的兩倍大小的 vector  len \*= 2;  vector<list<Data<T1, T2>>> new\_data(len);  // 逐一把資料由原本的 vector 放入新 vector  for(int i=0;i<data.size();i++){  // 每個 data[i] 都是一個 linked list  for(auto iter = data[i].begin();iter!=data[i].end();iter++){  // 取得一筆資料的 key 和 value  T1 key = (\*iter).Key;  T2 value = (\*iter).Value;  // 計算新的雜湊值 index  int index = Hash\_Func\_Mul(key, len);  // 把資料放到擴充後的陣列中  new\_data[index].push\_back(Data<T1, T2>{key, value});  }  } // end of for  // 把雜湊表改指向新的 vector  data = new\_data;  } // end of Rehashing |

為了方便測試，另外寫一個 STL 裡沒有的 Print 函式：

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

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  2  3  4  5  6  7  8 | template<typename T1, typename T2>  class Unordered\_Map{  private:  int amount;  ...  public:  ...  }; |

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  2  3  4  5  6 | template<typename T1, typename T2>  Unordered\_Map<T1,T2>::Unordered\_Map(int m){  amount = 0;  len = m;  data.resize(m);  } |

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  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | template<typename T1, typename T2>  T2& Unordered\_Map<T1,T2>::operator[](T1 input){  ...  data[index].push\_back(...);  amount++;  // 新增資料前，先檢查新增後的 load factor  // 如果新增後 load factor 會超過 5 的話，事先進行 Rehashing  if (amount / len > 5){  // 進行 Rehashing  Rehashing();  // 取得要新增的資料的索引值  int index = Hash\_Func\_Mul(input);  }  // 新增資料進原先（或者經過 Rehashing）的陣列 data[index].push\_back(Data<T1,T2>{input,0})  return data[index].back().Value;  } |

|  |  |
| --- | --- |
| 測試 Rehashing | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | int main()  {  Unordered\_Map<string,int> balance(1);  balance["Mick"] = 50;  balance["John"] = 60;  balance["Rallod"] = 70;  balance["David"] = 80;  balance["Daphlene"] = 90;  // load factor 正好等於 5，還沒有進行 Rehashing  balance.Print();  // 加上第 6 筆資料後，會進行 Rehashing  balance["Wang"] = 100;  balance.Print();  return 0;  } |
| 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

a. Key -> Value

b. 資料結構：雜湊表或紅黑樹

c. 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

a. Value

b. 資料結構：雜湊表或紅黑樹

c. 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

b. map 則「不是」雜湊表：插入、搜尋、刪除為

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 。

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 ，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間） | 佔用多餘空間 |
| 速度 | 較快， | 較慢， |
| 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 ，map 則是 ，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  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | #include <iostream>  #include <time.h>  #include <map>  #include <unordered\_map>  using namespace std;  int main()  {  clock\_t start, finish;    // 計算 map 耗用的時間  map<int, int> data\_map;  start = clock();  for(int i=0;i<10000000;i++)  data\_map[i] = i;  finish = clock();  cout << "Time consumption of map: " << (finish - start) /  (double)CLOCKS\_PER\_SEC << " s." << endl;  // 計算 unordered\_map 耗用的時間  unordered\_map<int, int> data\_unordered\_map;  start = clock();  for(int i=0;i<10000000;i++)  data\_unordered\_map[i] = i;  finish = clock();  cout << "Time consumption of unordered map: " << (finish -  start)/(double)CLOCKS\_PER\_SEC << " s." << endl;  return 0;  } |
| 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 find | |
| 1  2  3  4  5  6  7  8  9 | // 讓 iter 指到符合該 Key 的第一筆資料  iter = map\_name.find(Key);  // Key：first  // Value：second  if (iter!=map\_name.end())  cout << "Value: " << iter->second << endl;  else  cout << "Not found in this map!" << endl; |

(2) 刪除特定項 erase

|  |  |
| --- | --- |
| 刪除特定項 erase | |
| 1  2  3  4  5  6  7  8  9 | // 先用 find 找到元素的位置  iter = map\_name.find(Key);  // 再用 erase 把該筆元素刪除  map\_name.erase(iter);  // 也可以使用一個 flag 來紀錄 erase 是否成功  // 成功時會回傳 true，失敗會回傳 false  bool flag = map\_name.erase(Key); |

(3) 全部清空 clear

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

(4) 判斷是否為空 empty

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

(5) 取出所有資料

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

(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 unordered\_map | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | #include <iostream>  #include <unordered\_map>  using namespace std;  int main()  {  // 宣告一個把字串映射到整數的 unordered\_map  // 字串（客戶名稱） -> 整數（餘額）  unordered\_map<string, int> balance;  // 新增兩筆資料  balance["Mick"] = 100;  balance.insert(pair<string, int>("Rallod", 101));  // 印出目前的所有資料  for(auto& element : balance)  cout << "Key: " << element.first << ", Value: " << element.second <<  "\n";  // 查找特定鍵值（"Mick"）是否在 unordered map 內  auto iter = balance.find("Mick");  if(iter != balance.end())  cout << "Value: " << iter->second << endl;  else  cout << "Not found in this map!" << endl;  // 刪除特定鍵值 "Mick"  balance.erase("Mick");  // 再查找一次特定鍵值 "Mick" 是否在 unordered map 內  iter = balance.find("Mick");  if(iter!=balance.end())  cout << "Value: " << iter->second << endl;  else  cout << "Not found in this map!" << endl;  // 再印出目前的所有資料  for(auto& element : balance)  cout << "Key: " << element.first << ", Value: " << element.second <<  "\n";  return 0;  } |
| 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 I，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 I 個整數，每個整數 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

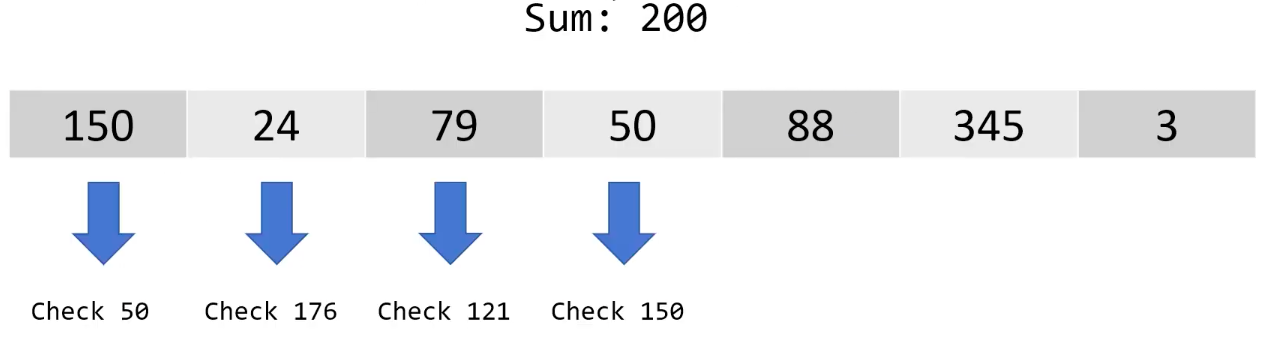
Case #3: 4 5

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。

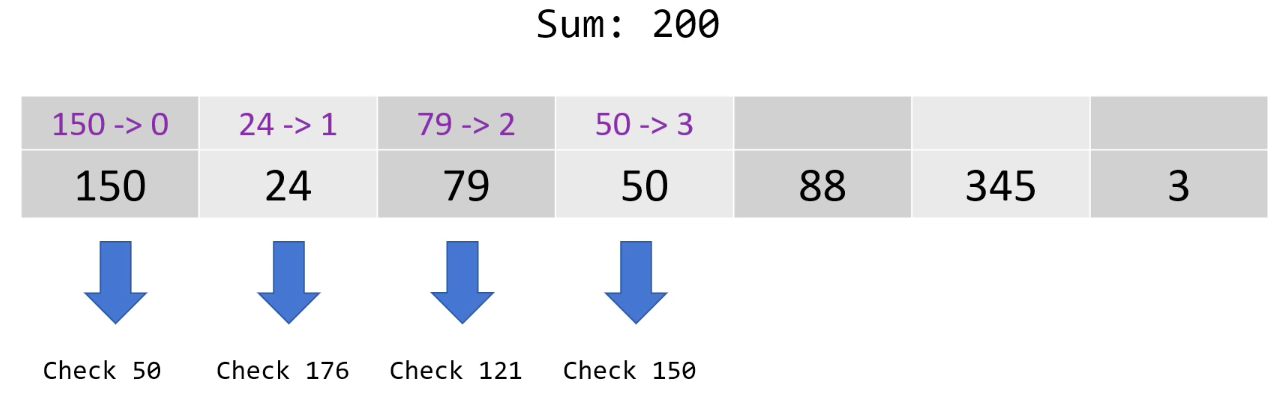
C. 解題邏輯

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 。

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 次，但是 仍然是 ，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」。



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 ，兩者相加顯示檢查一筆資料需要 。

要處理完一筆測資，最差的情況下也只需要 ，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 降為 ，這是很大幅度的改進。

|  |  |
| --- | --- |
| Google Code Jam | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69 | #include <vector>  #include <unordered\_map>  #include <stdlib.h>  using namespace std;  int main()  {  // 利用讀寫檔方式取得測資  ifstream file("A-large-practice.in");  ofstream ofile("Testoutput.txt");  // 存放資料的 vector  vector<int> data;  // 將商品價格映射到該商品索引的 unordered map  unordered\_map<int, int> number\_index;  // 當讀寫檔案成功時  if (file && ofile)  {  int dataCounts, totalPrice, stuffCounts;  // 取得總處理 case 數量  file >> dataCounts;  // 取得每個 case 對應到的三行資料  for (int i=0;i<dataCounts;i++){  file >> totalPrice >> stuffCounts;  data.resize(stuffCounts);  for (int j=0;j<stuffCounts;j++){  file >> data[j];  }  // 檢查總額減掉每筆資料的結果是否在 number\_index 中  for(int j=0;j<stuffCounts;j++){    auto iter = number\_index.find(totalPrice-data[j]);  // 如果迭代器不是 number\_index.end()  // 代表有找到資料  if (iter != number\_index.end()){  // 輸出互補商品及目前商品索引值  // 互補資料：number\_index[totalPrice-data[j]] + 1  // 目前資料：j+1  ofile << "Case #" << i+1 << ": "  << number\_index[totalPrice-data[j]]+1  << " " << j+1 << endl;  // 也可以直接 cout 以檢查  cout << "Case #" << i+1 << ": "  << number\_index[totalPrice-data[j]]+1  << " " << j+1 << endl;  break;  }  // 沒有找到互補商品的時候  // 把目前處理的 data[j] 存入 unordered map 中  // 且映射到索引值 j  else {  number\_index[data[j]] = j;  }  // 做完一個 case 後  // 清空 number\_index 以便下一輪使用  number\_index.clear();  }  } // end of for  } // 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  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | class Solution{  public:  bool containsDuplicate(vector<int>& nums){  // 宣告一個 unordered map 來存放已經出現過的數字  unordered\_map<int, bool> existed;  // 逐個檢查 nums 中的數字  for(int num: nums){  auto iter = existed.find(num);  if (iter!=existed.end()){  // 數字已經出現過  return true;  }  // 現在的數字還沒出現過時，加到 unordered map 中  existed[num] = true;  }  // 檢查完整個 vector 都沒有發現重複的數字  return false;  }  }; |

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  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | class Solution{  public:  bool containsNearbyDuplicate(vector<int>& nums, int k){  // 宣告一個 int 映射到 int 的 unordered map  // 前面的 int 是數值，後面的 int 指的是上次出現的位置  unordered\_map<int, int> index;  // 用迴圈遍歷向量 nums  for (int i=0;i<nums.size();i++){  // 檢查 nums[i] 這個數字是否已經在 index 中出現過  auto iter = index.find(nums[i]);  if (iter!=index.end()){  // 如果有找到，比對距離是否超過 k  if (i-index[nums[i]] <= k)  // 距離不超過 k，回傳 true  return true;  else  // 距離大於 k，視為沒有重複  // 更新目前處理數字 nums[i] 的索引值即可  index[nums[i]] = i;  }  else  {  // 沒有找到時代表第一次出現，把位置加到 index 中  index[num[i]]-i;  }  // 處理到陣列尾端都沒有發生「距離在 k 之內」的情形  // 回傳 false  return false;  } // end of for  } // end of containsNearbyDuplicate  }; // 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  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47 | class Solution{  public:  bool isAnagram(string s, string t){  // 如果兩個字串長度不同，一定不符合條件  if (s.size() != t.size())  return false;  // 宣告 unordered map 紀錄每個字元出現次數  unordered\_map<char, int> counts;  // 依序處理 s 中的每個字元  for (char c:s){  auto iter = counts.find(c);  if (iter!=counts.end())  // counts 中已經出現過 c 這個字元  // 把出現次數加一  counts[c]++;  else  // counts 還沒出現過 c 這個字元  // 把出現次數設定為 1  counts[c] = 1;  }  // 依序處理 t 中的每個字元  for (char c:t){  auto iter = counts.find(c);  if (iter!=counts.end())  // t 中每出現任一字元 c  // 把 counts 中 c 對應到的數字減一  if(counts[c]>0)  counts[c]--;  else  // s 中出現過的次數已經被用完了  return false;  else  // c 沒有出現在 counts 中  // 代表 t 中出現了 s 裡沒有的字元  return false;  }  // 每個字元都符合條件，回傳 true  return true;  } // end of isAnagram  }; // 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

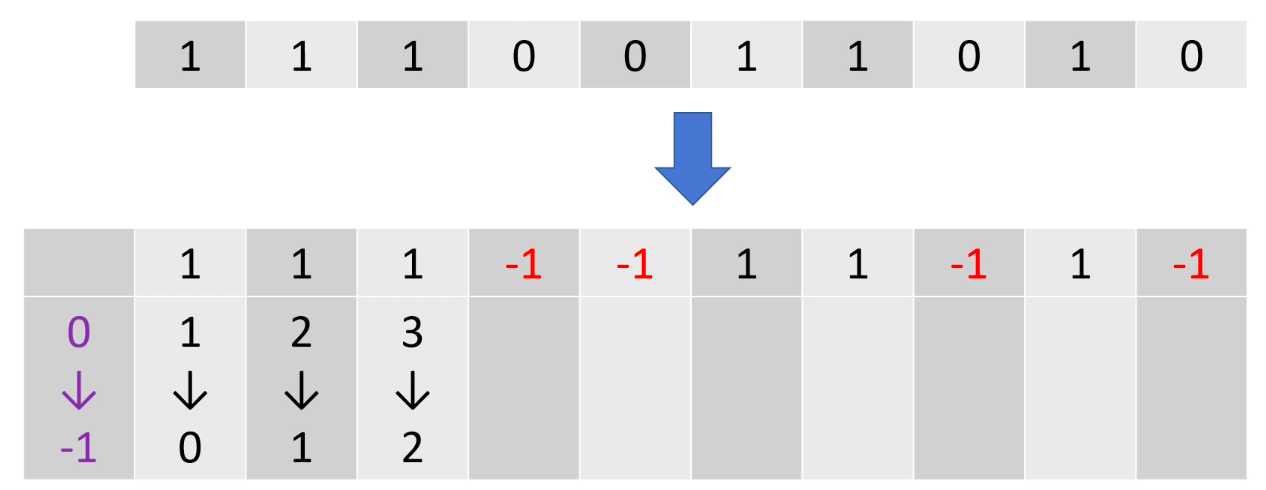
輸入：nums = [0,1,0]

輸出：2

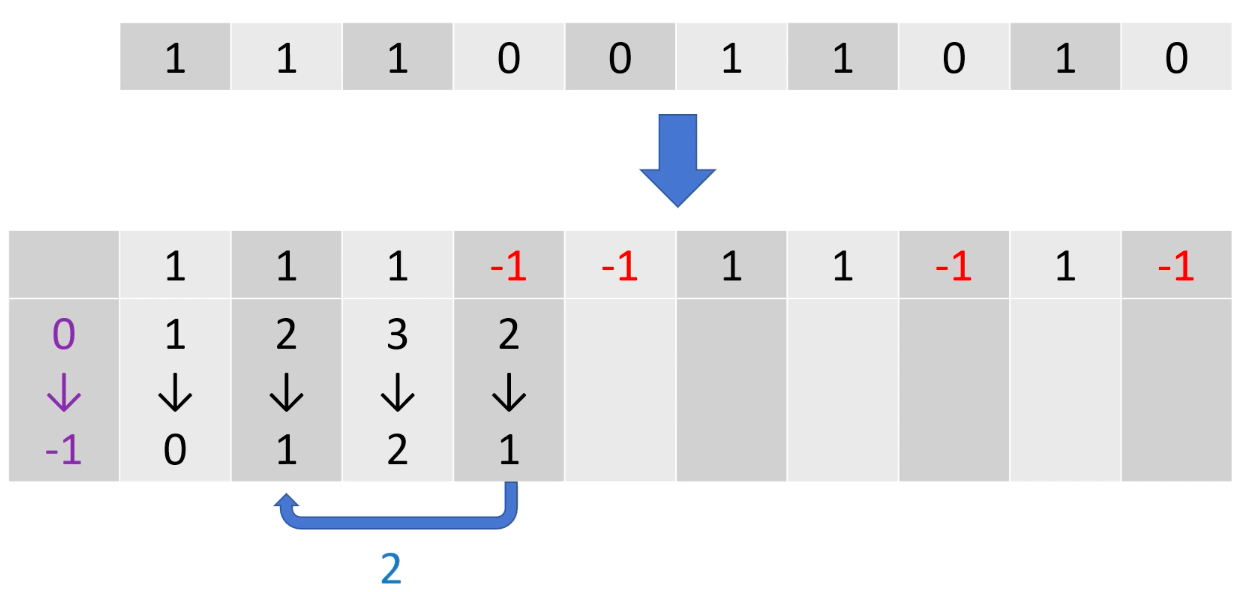
在範例二中，前面的 [0,1] 和後面的 [1,0] 都符合條件，回傳其中一個的長度 2。

E. 解題邏輯

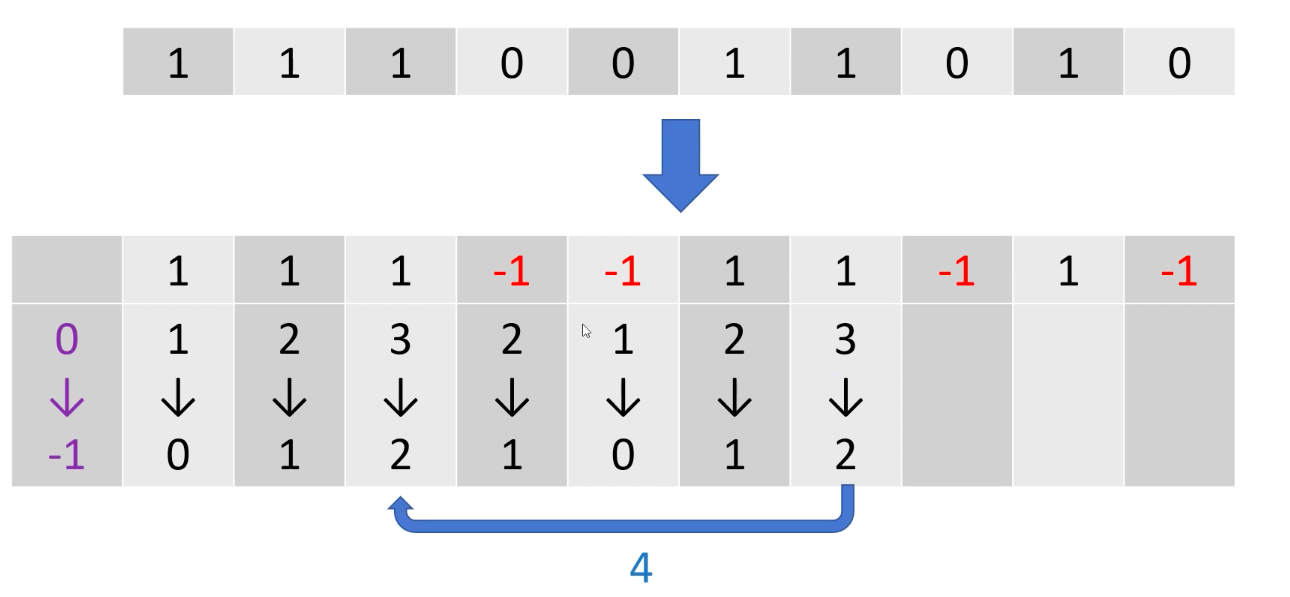
可以做一個轉換來讓檢查 0 和 1 出現次數是否相同的過程變得較為簡便：將所有的 0 都視為 -1，這樣一旦某個子陣列中 0 和 1 出現次數相同，比如 0 出現 5 次，1 也出現 5 次，這段子陣列的加總就會是

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。

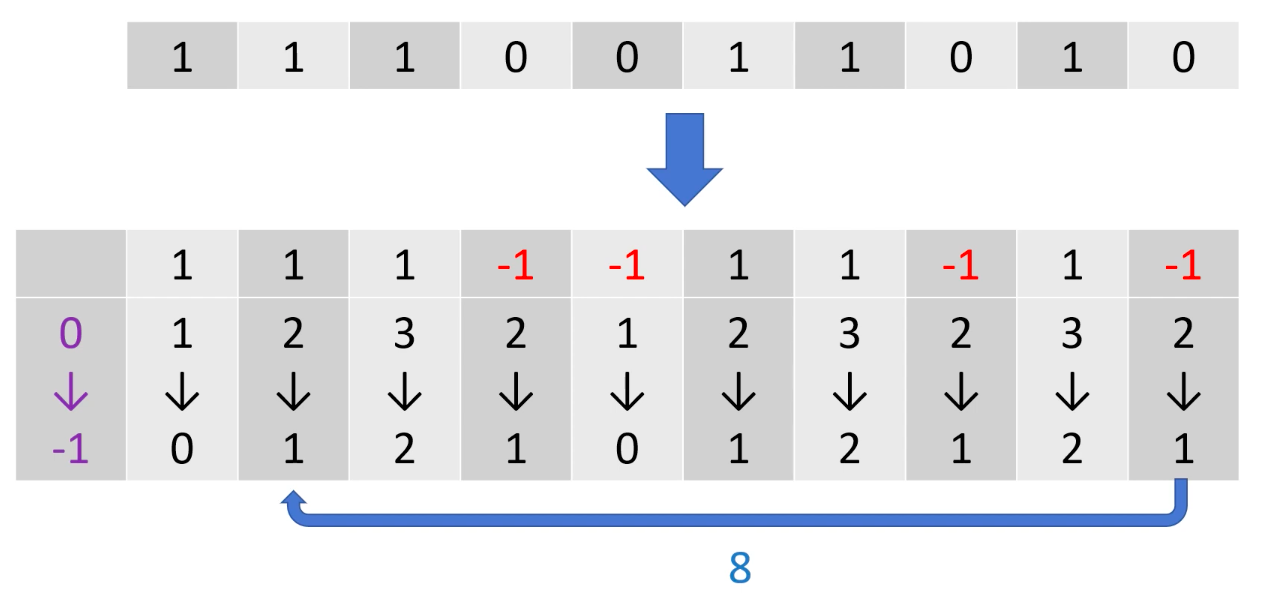
開始處理之前，先把總和 sum 設為 -1，避免開頭就誤認出現符合條件的子陣列的情形，接下來，每遇到一個數字就加上 1 或減去 1。



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  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50 | class Solution{  public:  int findMaxLength(vector<int>& nums){  // 宣告一個 unordered map  // 紀錄某個數字上一次出現在索引值多少  // 第一個 int 是數值，第二個 int 是上次出現的索引值  unordered\_map<int, int> data;  // 將開頭設定為 -1  data[0] = -1;  // 儲存目前的數字加總  int sum = 0;  // 目前符合條件的最長的陣列長度  int longest = 0;  // 依序處理 num 中的數字  for(int i=0;i<num.size();i++){  // nums[i] 是 1  if (nums[i])  sum++;  // nums[i] 是 0，看成 -1  else  sum--;  // 檢查目前為止的 sum 數值是否出現過  auto iter = data.find(sum);  // 目前數值已經出現過  if (iter!=data.end()){  // 比較最長的子陣列長度 longest  // 與這次發現的子陣列 i-data[sum] 長度哪個較大  longest = longest > i-data[sum] ? longest : i-data[sum];  }    // 若目前的 sum 值第一次出現：把 sum 對應目前索引值  else {  data[sum] = i;  }  // 回傳得到的最長子陣列長度  return longest;  } // end of for  } // end of findMaxLength  }; // end of class Solution |

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