資料結構介紹

<https://hackmd.io/@Aquamay/H1nxBOLcO/https%3A%2F%2Fhackmd.io%2F%40Aquamay%2FSyd8UdLqu>

資料結構包括：**線性結構**和**非線性結構**。

**線性結構**

線性結構作為最常用的資料結構，其特點是數據元素之間存在一對一的線性關係。

線性結構有兩種不同的存儲結構，即 (1)順序存儲結構 和 (2)鏈式存儲結構(Chained Storage Structure)。

順序存儲的線性表稱為順序表，順序表中的存儲元素是連續的；

鏈式存儲的線性表稱為連結串列，鏈表中的存儲元素不一定是連續的，元素節點中存放資料元素以及相鄰元素的位址資訊。

線性結構常見的有：陣列(Array)、佇列(Queue)、連結串列(Linked list)和棧(Stack)

**非線性結構**

包括：二維陣列，多維陣列，廣義表(Generalized List)，樹結構，圖結構。

## **陣列 (array)**

在 Python 裏，同樣是陣列，但有 array 和 list 兩種數據類型。兩者的差異在於，前者屬於 Python 模組 numpy 裡的一種數據類型，所包含的所有元素類型都必須相同；而後者則是 Python 內建的數據類型，可以包含不同的元素類型。

numpy.array = [1, 2, 3, 4]

list = [1, 2, 3, 'string', (12, 'hi')]

那為什麼 list 可以包含不同的元素類型？它其實就像 C 語言裡的 指針(Pointer)，保存的資料是數據存放的位置。因此不論 list 裡的元素型態是什麼，只要藉由讀取元素的位置，就可以獲取我們所存在 list 的資料。

在 C 語言中的陣列，如果要插入一個資料，必須將資料的位置一個一個往後挪，把位置空出來，才能把資料放進去。假設要插入 18，勢必得把 11 後面的資料全部往後移一位。

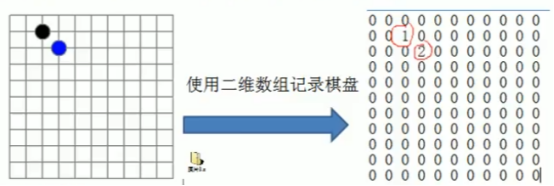
|Memory|0|1|2|3|4|5|

|-|-|-|-|-|-|-|

|Data|11|　|22|33|44|55|

|Data|11|18|22|33|44|55|

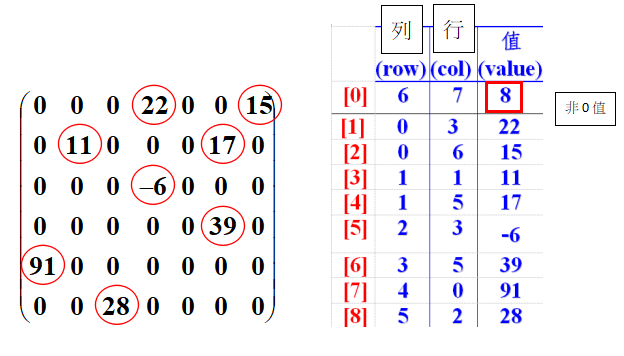
## **稀疏矩陣(Sparse Matrix)**

五子棋程序如何存取盤的下棋情況？A：可以使用二維陣列，將黑棋白棋記錄下來。  


**思考**Q：該二維陣列很多值默認為0，因此記錄了很多沒意義的數據，該如何去解決？  
A：可以使用稀疏矩陣來保存該陣列。

## **稀疏矩陣的處理方法**

1. 記錄陣列一共有幾行幾列，有多少個不同的值。
2. 把具有不同值的元素的行列及值記錄在一個小規模的陣列中，從而縮小程式的規模。

以下圖為例，我們可以將原先 6\*7 = 42 的陣列縮減為 9\*3 = 27 的陣列：  


【小結】

使用稀疏矩陣的情況：

* 當一個陣列有很多沒意義數據(0)或相同數據。

處理方法：

* 第一行記錄陣列共幾列幾行並且有幾個非0的值。
* 第二行開始用於紀錄每個非0值在陣列第幾列第幾行、值為多少。

## **二維陣列 轉 稀疏矩陣**

**思路**

1. 遍歷原始的二維陣列，得到有效數據的個數(sum)。
2. 根據sum 就可以創建稀疏矩陣parseMatrix int[sum+1][3]
3. 將二維陣列的有效數據存入到稀疏矩陣。

## **稀疏矩陣 轉 二維陣列**

**思路**

1. 先讀取稀疏矩陣第一行，根據第一行的數據創建原始的二維陣列，比如上方圖示chessArr = int[11][11]
2. 再讀取稀疏矩陣後幾行的數據，並賦給原始的二維陣列。

## **代碼實現**

現在，我們用JAVA來將下方這個二維陣列轉維稀疏矩陣。



### **創建原始的二維陣列 11\*11**

0: 沒有旗子, 1:黑子, 2:藍子

//創建原始的二維陣列 11\*11

//0: 沒有旗子, 1:黑子, 2:藍子

int chessArr1[][] = new int[11][11];

chessArr1[1][2] = 1;

chessArr1[2][3] = 2;

//輸出原始二維陣列

for(int[] row: chessArr1){

for(int data:row){

System.out.printf("%d ",data);

}

System.out.println();

}



### **獲取非0數據個數**

先遍歷二維陣列,獲取非0數據個數

//1.先遍歷二維陣列,獲取非0數據個數

int sum = 0;

for(int i=0;i<11;i++){

for(int j=0;j<11;j++){

if(chessArr1[i][j] !=0) sum++;

}

}

System.out.println(sum);

### **創建對應的稀疏矩陣**

創建對應的稀疏矩陣並且給稀疏矩陣賦值

//2.創建對應的稀疏矩陣

int sparseArr[][] = new int[sum+1][3]; //列:sum為非0值的個數+1列存放原始二維陣列資訊 行:列,行,非0值

//給稀疏矩陣賦值

sparseArr[0][0] = 11;

sparseArr[0][1] = 11;

sparseArr[0][2] = sum;

//遍歷二維陣列,將非0值存放到稀疏矩陣中

int count=0; //用於紀錄是第幾個非0數據

for(int i=0;i<11;i++){

for(int j=0;j<11;j++){

if(chessArr1[i][j] !=0){

count++;

sparseArr[count][0] = i;

sparseArr[count][1] = j;

sparseArr[count][2] = chessArr1[i][j];

};

}

}

//輸出稀疏矩陣

for(int[] row: sparseArr){

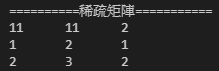
for(int data:row){

System.out.printf("%d\t",data);

}

System.out.println();

}



### **恢復原始二維陣列**

//恢復原始二維陣列

int chessArr2[][] = new int[sparseArr[0][0]][sparseArr[0][1]];

//從稀疏矩陣第二行開始讀取數據,並賦給原始二維陣列

for(int i=1;i<sparseArr.length;i++){

chessArr2[sparseArr[i][0]][sparseArr[i][1]] = sparseArr[i][2];

}

for(int[] row: chessArr2){

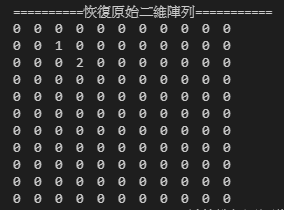
for(int data:row){

System.out.printf("%d ",data);

}

System.out.println();

}



## [**堆疊(Stack)**](https://ithelp.ithome.com.tw/articles/10326088)

堆疊是一種抽象資料型態(Abstract Data Type, ADT)。看到抽象內心是否有一種滿頭問號的感覺，就比如說當你聽到水果，會覺得水果抽象嗎？指的到底是蘋果還是香蕉呢？當描述是指一個大的範圍，不夠仔細、不夠具體的時候，其實就是一種抽象化的名詞。

堆疊 stack 有兩種實作方式：陣列 array 與鏈結串列 linked list，在此選擇以類似陣列的 Vector 實現。

特性：

1. 只能從堆疊的最頂端存取資料
2. 只能從堆疊的最頂端新增或刪除資料
3. 資料的存取必須符合後進先出(Last In First Out, LIFO)

功能:

1. Create : 可建立一個空的堆疊
2. Push : 可在頂端新增資料，並得到一個新的堆疊
3. Pop : 可刪除頂端資料，並得到一個新的堆疊
4. Size
5. Peek : 回傳最後加入堆疊的資料。與 pop 操作類似，但不會對堆疊造成任何影響。如果偷看的是一個空堆疊，會得到 None。

**陣列實作堆疊的壞處**

使用陣列實作堆疊有一個明顯的壞處，就是受限於宣告陣列的大小。如果 stack 中的元素數量超過事先宣告的陣列大小，就會產生問題。這限制了 stack 的彈性和應用範圍。

**實際應用**

1. 資料反序輸出，例如將由小到大的資料改成由大到小
2. 迴文判斷
3. 河內塔問題
4. 括號配對

C++ STL 的 stack

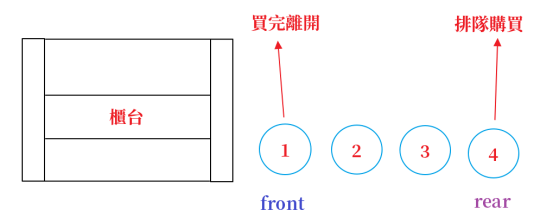
好消息是 C++ 已經提供了內建的 STL 類別，可以動態調整 stack 的大小，從而解決了陣列實作的大小限制。因此使用 STL stack，我們可以更方便地進行堆疊操作。

C++ STL stack常用的操作：

1. empty() 回傳一個布林值，表示 stack 是否為空
2. size() 回傳一個無號整數，表示 stack 的長度
3. top() 回傳 stack 最頂端的元素
4. push() 將元素放入 stack 的頂端



## **佇列(Queue)與環形佇列**

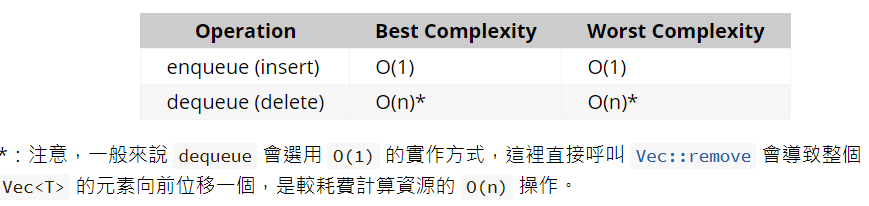
使用場景: 排隊買票  


## **特徵**

1. 先進先出(FIFO)
2. 有序列表，可以用陣列或連結串列來實現。
3. Queue兩個指針：rear 佇列的尾部(含)、front 佇列的頭部(不含)。
4. 新增數據時，front不動rear動；刪除數據時，rear不動front動 => 由尾端加入數據，由頭部取出。

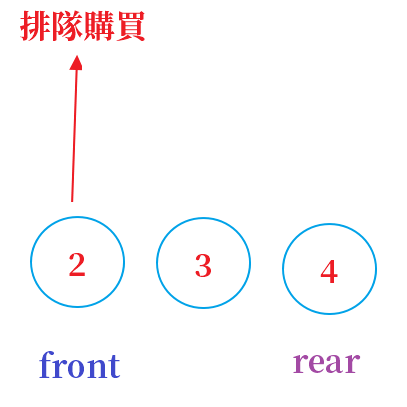
基本操作

1. enqueue：將新資料加入佇列
2. dequeue：將最先放入的資料移出佇列
3. peek：在不將資料移出佇列的情況下取得最先放入的資料
4. size：取得佇列大小

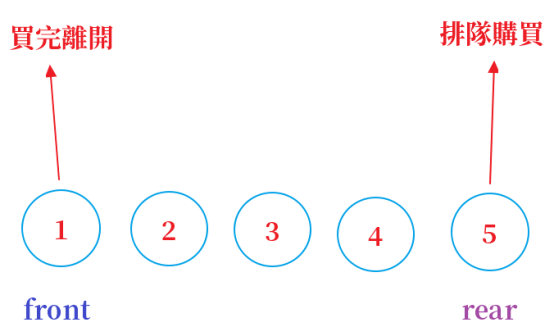


我們可以以排隊買票為例，將排隊的隊列視作Queue，先排隊的人先買票，後到的人排在後面等待買票，並且先買票的人買完票之後就離開隊伍；我們將第一個買票的人稱之為front，隊伍最後一個等待買票的人為rear。

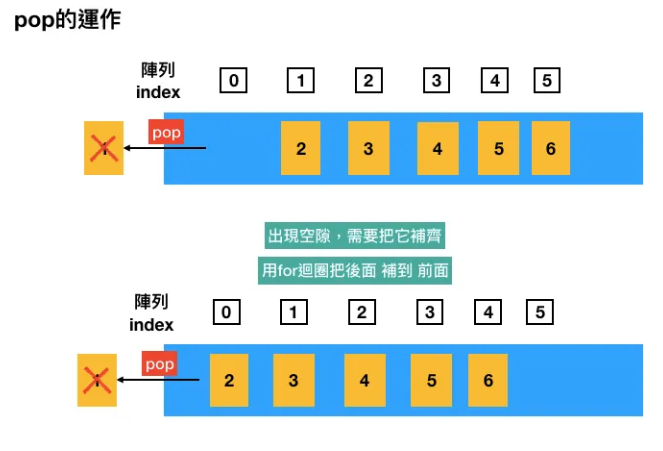
當1號買完票離開換2號買票時，此時的front變為2號。



當又有下一個人(5號)要買票，此時rear從4號變到5號身上。



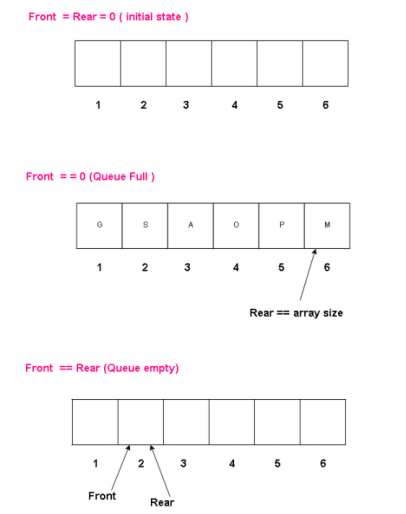
因此我們可以得出，當新增數據時，front不動rear動；刪除數據時，rear不動front動。



## **陣列模擬佇列**

當我們將數據存入佇列時稱為addQueue，addQueue的處理須要有兩個步驟：

1. 將尾指針往後移：rear+1，當 front == rear 為空。
2. 若尾指針 rear < maxSize-1 則將數據存入rear所指的陣列元素中；若 rear == maxSize-1 為佇列已滿。



目前使用會遇到的困難為：**佇列使用一次就不能再用，當我將數據取出，前端明明還有空位但想新增數據卻被告知佇列已滿。**

因此需要優化，將之改為**環形佇列**。

### 陣列模擬環形佇列

**思路分析**

1. front的涵義做一個調整: front就指向佇列的第一個元素，也就是說 arr[front] 就是佇列的第一個元素，front 的初始值 = 0
2. rear的涵義做一個調整: rear就指向佇列的最後一個元素的後一個位置，因為希望空出一個空間用於判斷環形佇列是否為空或滿，rear 的初始值 = 0
3. 當佇列滿時，條件是 (rear+1) % maxSize = front
4. 當佇列為空時，條件是 rear == front
5. 佇列中有效的數據個數為 (rear + maxSize - front) % maxSize

使用環形佇列後，數據可以一直刪減、新增直到沒有空餘的位置，形成一個環狀的結構，因此不會浪費多餘的空間。

**舉個簡單的例子來加深印象好了：**

當我今天宣告一個環形佇列 MaxSize 為4，那就代表我這個佇列有效數據最大是3，因為有一個儲存單元要預留下來判斷是否為空還是滿。

添加數據 10,20,30  
***arr[0] = 10 , arr[1] = 20 , arr[2] = 30***

取出數據，則只剩下  
***arr[1] = 20 , arr[2] = 30***

新增數據 40  
***arr[1] = 20 , arr[2] = 30 , arr[3] = 40***

繼續取出 20 和 30  
***arr[3]=40***

再去新增數據 50,60  
***arr[3] = 40 , arr[0] = 50 , arr[1] = 60***

再把 40 取出，加入70 會變為  
***arr[0] = 50 , arr[1] = 60 , arr[2] = 70***

### **佇列條件**

* front的初始值 = -1 , rear的初始值 = -1
* 判斷佇列為空 rear == front
* 判斷佇列為滿 rear = maxSize -1

### **環形佇列條件**

* front的初始值 = 0 , rear的初始值 = 0
* 需預留一個儲存單元來判斷佇列是否滿 (容量僅剩一個儲存單元時，表示緩衝區已滿。)
* 判斷佇列為空 rear == front
* 判斷佇列為滿 (rear+1) % maxSize == front
* 判斷佇列的有效數據個數 (rear + maxSzie - front) % maxSize
* front = 0  
  rear = 5  
  maxSize = 7  
  (5+7-0) % 7 = 5 ，共有五個有效數據。  
  **為什麼不是6？**因為rear會指向最後一個元素的後一個位置。

## **雙端佇列 Deque**

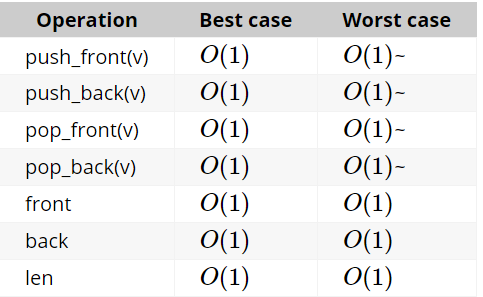
雙端佇列（double-ended queue，通常縮寫為 deque）是一般化（generalize）的佇列或堆疊。比起佇列只能「先進先出 FIFO」，以及堆疊只有「後進先出 LIFO」，雙端佇列可以從最前端或最末端任意方向，在常數時間複雜度內增刪元素，更為方便。

雙端佇列常見用動態陣列或是鏈結串列實作，動態陣列的實作會因空間不夠，需要重新配置記憶體，但通常會支援隨機存取（random access）；鏈結串列的實作版本雖無法隨機存取，相對地增刪元素不需記憶體重配置。

雙端佇列顯著的缺點是：無論以動態陣列或連結串列實作，基本款實作通常無法在 O(k)時間複雜度的情況下，選定範圍內 k 個元素並取得單一切片（slice）。這肇因於動態陣列的實作通常內部儲存空間經過多次增刪，空間利用會不連續；而用鏈結串列實作者，則因不支援隨機存取而無法達到相應的複雜度。

雙端佇列任何操作都是直接對 head 或 tail 的索引讀寫記憶體，複雜度皆為 O(1)，不過因為增減元素需要動態調整儲存空間大小，所以這些方法的時間複雜度需要平攤。

空間複雜度則是只用了一個環形緩衝區儲存元素，和幾個欄位儲存 tail、head 還有容量，因此額外空間複雜度只有 O(1)



## 

## 

## 

## **鏈結串列 (linked list)**

### <https://alrightchiu.github.io/SecondRound/linked-list-xin-zeng-zi-liao-shan-chu-zi-liao-fan-zhuan.html>

### https://pisces1026.wordpress.com/2017/09/21/cc-linked-list/

### **特性：**

在各節點儲存額外的指標指向下一個節點，能夠不使用連續的記憶體空間的情況下，能夠建立一份連續的資料。透過存放位置找到鏈結串列的每一個元素。

相對來看，陣列則需要使用連續的記憶體空間，實作出其他的資料結構，例如堆疊(Stack)和佇列(Queue)等。

**優點：**

1. 不使用連續記憶體空間(不需事先指定資料型別大小)，利用動態記憶體建立。
2. 容易地修改指標，以常數時間插入或移除節點，不需重新配置記憶體（reallocation）。
3. 插入的資料多，讀取的資料少的資料適合使用

**缺點** (因動態配置記憶體等因素)：

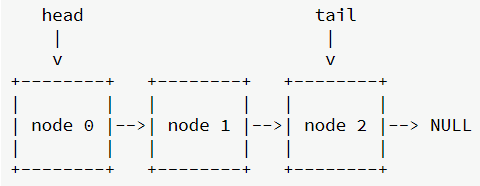
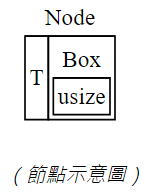
1. 空間開銷大：使用額外的記憶體空間紀錄節點指標
2. 較差的 CPU 快取：不連續存取的特性，不利於 CPU 快取。
3. 不允許隨機存取（random access）(無法快速索引到某個節點)：搜尋特定節點仍需線性時間循序存取，當沒有需要依序讀取整個元素的話，使用鏈結串列就比較浪費時間了(鏈結串列為依序讀取)

**應用場景：**

1. 需要頻繁地插入與刪除資料。
2. 需要頻繁分離與合併（split and merge）資料。
3. 不需要隨機存取的資料。
4. 遞迴友好，因此成為大多函數式語言中基本資料型別之一。
5. 教學上，常用於實作抽象資料型別，如堆疊與佇列等等。
6. 鏈結串列可能用在某些網站為了賺瀏覽量，一頁只顯示一個項目，例如10大電影，使用者必須一頁一頁的點擊『下一頁』去瀏覽整個資訊
7. 服務生將新訂單寫入，廚師只要取出最前端的訂單
8. 紀錄每天花費記帳程式，月底才需要檢視

結構 :

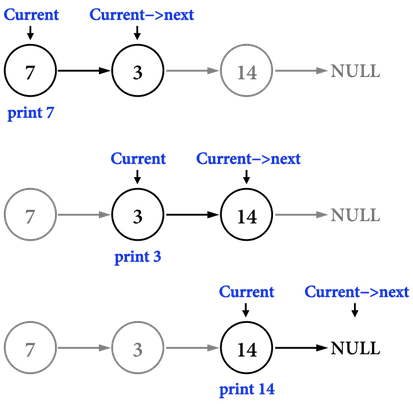
1. [Node](https://rust-algo.club/collections/linked_list/index.html#node) : 又稱「節點」，為組成鏈結串列的基本元素。節點包含(1)資料儲存區與(2)指標儲存區。指標儲存區：指標儲存當前節點指向下一個節點地址。
2. Head 為指向整個串列第一個節點的指標。
3. Tail 則為指向最後一個節點的指標。
4. 最後一個節點連接的地址寫「NULL」(不指向其他節點)，表示鏈結串列結束。



鏈結串列種類 :

1. 單向鏈結串列 : 每個節點只有一個指標，指向下一個節點。
2. 雙向鏈結串列 : 每個節點有兩個指標，分別指向前後一個節點。
3. 循環鏈結串列 : 倘若該鏈結串列末端節點的指標指向第一個的節點，形成一個循環。

### PrintList() : 功能就是把Linked list中的所有資料依序印出。要印出所有的資料，就必須「逐一訪問(**Visiting**)」Linked list中的每一個node，這樣的操作又稱為**Traversal(尋訪)**。能夠完成這樣的操作，要歸功於node中記錄了「下一個node的記憶體位置」，如此，才能在訪問完當前的node之後，知道要繼續往哪一個記憶體位置上的node前進。



* 建立ListNode \*current來表示「目前走到哪一個node」。
* 若要對Linked list存取資料，必定是從第一個node開始，所以把current指向first所代表的記憶體位置，current=first。
  + 目前first即為node(7)。
  + 同時，還能夠知道「下一個node」是指向node(3)。
* 在印出current->data，也就是7後，便把current移動到「下一個node」。
  + 透過current=current->next，即可把current指向node(3)所在的記憶體位置。
* 重複上述步驟，直到current指向Linked list的終點NULL為止，便能印出所有資料。

由此可見，所有需要在Linked list中尋找特定資料的操作，都會用上**Traversal**。

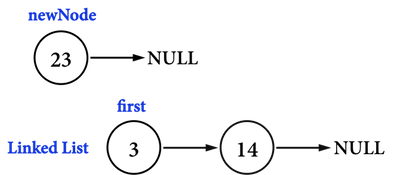
Push\_front()的功能是在Linked list的開頭新增資料。

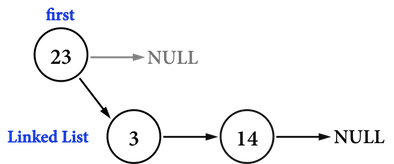
若考慮在Linked list(3->14)的開頭加入23，方法如下：

* 先建立一個新的節點ListNode \*newNode，帶有欲新增的資料(23)，如下圖一
* 將newNode中的**pointer**：ListNode \*next，指向Linked list的第一個nodefirst，如下圖二。
* 接著，把first更新成newNode。

經過以上步驟(時間複雜度為O(1))便得到新的Linked list：

23->3->14。



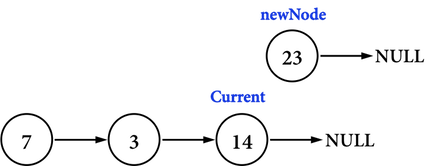


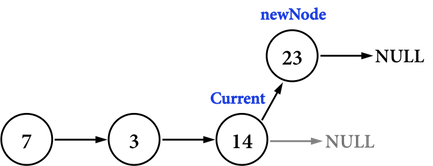
Push\_back()的功能是在Linked list的尾巴新增資料。

若考慮在Linked list(7->3->14)的尾巴加入23，方法如下：

* 先建立一個新的節點ListNode \*newNode，帶有欲新增的資料(23)。
* 先利用如同PrintList()中提過的**Traversal**，把新建立的ListNode \*current移動到Linked list的尾端，node(14)，如圖三(a)。
  + 有些資料結構會在class LinkedList中新增一項ListNode \*last，記錄Linked list的最後一個node，那麼，Push\_back()就不需要**Traversal**，可以在O(1)時間內完成。
  + 若沒有ListNode \*last，就需要O(n)的**Traversal**。
* 接著把current的next pointer指向newNode，如圖三(b)。

即可得到新的Linked list：7->3->14->23。



****

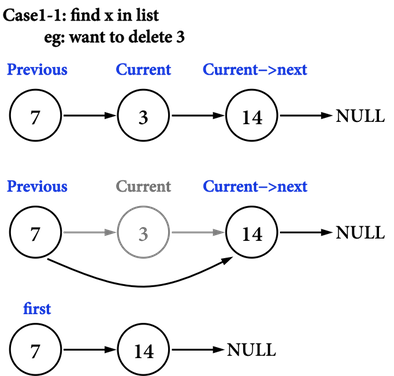
Delete(int x)的功能是要刪除Linked list中，資料為int x的node。

一共會有兩種情形，第一種是Linked list中確實有int x，第二種是沒有。  
在第一種情況中，需要再特別考慮「int x位於first」的情況。

**case1-1**：要在Linked list(7->3->14)中刪除具有3的node，見圖四(a)：

* 建立兩個在Linked list中移動的指標：\*current以及\*previous。
* 利用**Traversal**的概念，以ListNode \*current指向node(3)，以ListNode \*previous指向node(3)的「前一個node」，node(7)。
* 接著，把previous的**pointer**指向current的**pointer**。
  + 此處，即為以node(7)記住node(14)的記憶體位置。
* 再把current的記憶體釋放(若是使用new進行動態配置，就使用delete釋放)，還給**heap**。

關鍵就是，在整個Delete()的過程，只有node(3)知道node(14)的記憶體位置，所以在把node(3)刪除之前，必須先透過node(3)的**pointer**找到node(14)，把node(14)接到node(7)上之後，才可以釋放node(3)的記憶體位置。



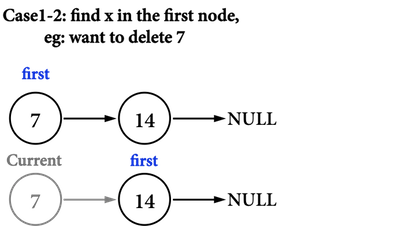
**圖四(a)。**

**case1-2**：若要刪除具有7的node，而且node(7)位於Linked list的第一個位置(也就是\*first)，見圖四(b)：

* 需要把這個情況獨立出來的原因是，這個情況不會進行**Traversal**，所以ListNode \*previous始終指向NULL，便不能呼叫其private data，若進行previous->next將會因為意圖對「無效的」記憶體位置進行存取，而產生像是「EXC\_BAD\_ACCESS」的錯誤(error)。

移除的方法：

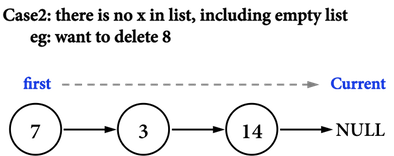
* 只要將first向後移動至first->next。
* 再釋放current的記憶體位置即可。
  + 若Linked list只有一個node，那麼first=first->next將會把first指向NULL。



**圖四(b)。**

**case2**：若Linked list中沒有要刪除的node，見圖四(c)：

* 若想要刪除8，但是Linked list(7->3->14)沒有8，那麼在**Traversal**後，ListNode \*current會一路走到Linked list的結尾，也就是NULL。
* 若Linked list本來就是空的，那麼建立的ListNode \*current = first，current也會指向NULL。
* 以上這兩種情況，直接結束Delete()函式。



**圖四(c)。**

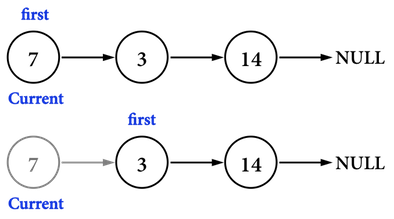
**Clear()的功能是清除整個Linked list。方法如下：**

* **從Linked list的「第一個node」first開始，進行Traversal。**
  + **利用first=first->next即可不斷移動first。**
* **建立一個ListNode \*current記錄「要刪除的node」之記憶體位置。**
* **重複上述步驟，直到first指向Linked list的尾巴NULL為止。**

**見圖五(a)：**

* **原先first記錄的是node(7)。**
* **建立ListNode \*current記錄first，也就是node(7)。**
* **將first移動到node(3)。**
* **刪除current指向的node(7)。**

**如此，便把node(7)從Linked list移除。**

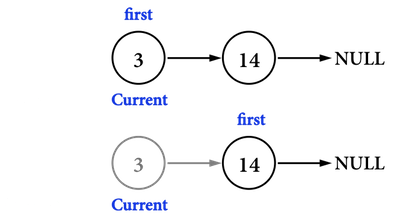
****

**圖五(a)。**

**見圖五(b)：**

* **目前first記錄的是node(3)。**
* **建立ListNode \*current記錄first，也就是node(3)。**
* **將first移動到node(14)。**
* **刪除current指向的node(3)。**

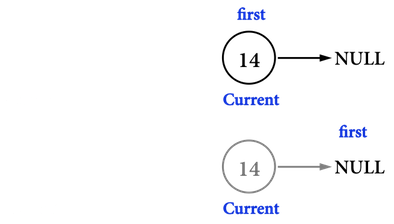
**如此，便把node(3)從Linked list移除。**

****

**圖五(b)。**

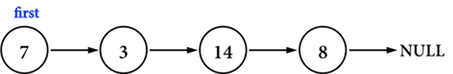
**見圖五(c)：**

* **目前first記錄的是node(14)。**
* **建立ListNode \*current記錄first，也就是node(14)。**
* **將first移動到NULL。**
* **刪除current指向的node(14)。**

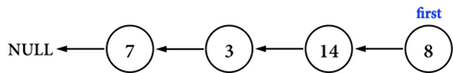
****

**圖五(c)。**

**Reverse()的功能是反轉Linked list，以圖六(a)的Linked list為例，經過Reverse()之後，預期得到圖六(b)。**

****

**圖六(a)。**

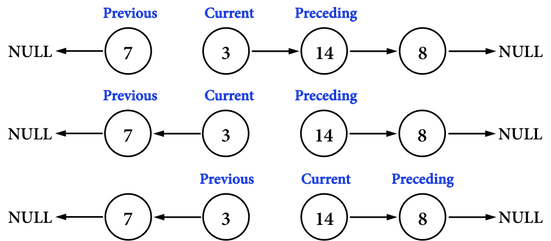
****

**圖六(b)。**

**要倒轉Linked list，其實就是把每個node的pointer的方向前後對調，但是因為每個node都只有被Linked list中的「一個node」記得，例如圖六(a)，只有node(7)記得node(3)的記憶體位置，只有node(14)記得node(8)的記憶體位置，所以，如果把node(14)的ListNode \*next(原本指向node(8))更新成指向node(3)，那麼整個Linked list中，就再也無法存取node(8)。**

**所以在更新任何一個node之pointer之前，除了要知道「新的要指向的node」之記憶體位置，也要記錄「原先記錄的node」之記憶體位置，這裡使用三個指向node的指標，分別為previous、current、preceding，以圖六(c)為例：**

* **目前current為node(3)，其指標current->next指向的是node(14)。**
* **目前previous為node(7)，是current->next最後要指向的記憶體位置。**
* **目前preceding為node(14)，避免current->next更新成node(7)後，再也找不到node(14)。**

****

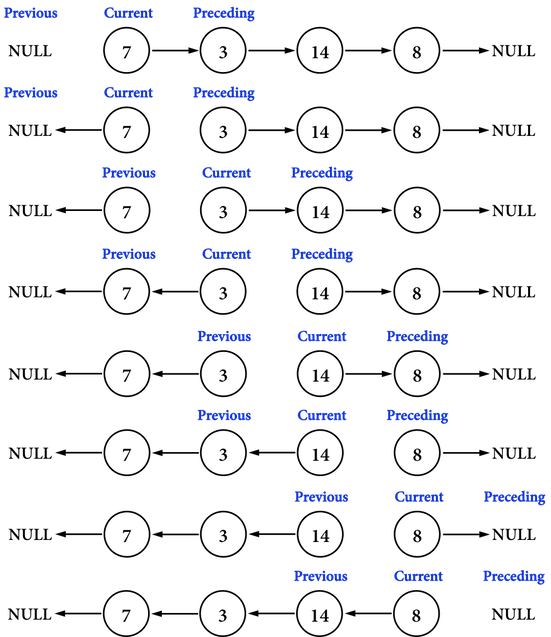
**圖六(c)。**

**有了這三個指標後，要執行的步驟只有兩個：**

1. **將current->next從原本指向preceding更新成指向previous，如圖六(c)中圖。**
   * **執行current->next=previous，就把node(3)的指向node(7)。**
2. **把三個指標「按照順序」往前移動，然後進行下一個node之pointer調整，如圖六(c)下圖。**
   * **previous=current，將previous移動到node(3)。**
   * **current=preceding，將current移動到node(14)。**
   * **preceding=preceding->next，將preceding移動到node(8)。**

**重複上述步驟，直到preceding更新成NULL，調整Linked list的first所指向的記憶體位置，即完成Linked list之反轉。**

**完整圖示見圖六(d)：**

****

**圖六(d)。**

* [**Wikipedia：Doubly linked list**](https://en.wikipedia.org/wiki/Doubly_linked_list)
* [**Tutorials Point：Data Structure - Circular Linked List**](http://www.tutorialspoint.com/data_structures_algorithms/circular_linked_list_algorithm.htm)

## 

|  | **陣列** | **鏈結串列** |
| --- | --- | --- |
| 元素的存放 | 一個接一個連續排列 | 分散在各處 |
| 存取方式 | 隨機存取 | 循序存取 |
| 讀取 | O(1) | O(n) |
| 插入 | 需要搬移插入點之後的所有元素，沒有空間會發生錯誤 O(n) | 只要更改元素的指向位置 O(1) |
| 刪除 | 刪除完元素後，後續元素往前移動 O(n) | 只要更改元素的指向位置 O(1) |

備註：鏈結串列 雖然新增刪除是O(1)但是查訪需要O(n)

### 

### **單向鏈結串列**

1. 特色是每兩個節點間只有一個單向的鏈結。
2. 單向鏈結串列查找的方向只能是一個方向，而雙向鏈結串列可以向前或向後查找。
3. 單向鏈結串列不能自我刪除，需要靠輔助節點，而雙向鏈結串列則可以自我刪除。

單向鏈結串列也支援 tail-sharing，也就是共享 sublist。藉由共享 sublist，單向鏈結串列很容易實作 persistent data structure，再配合 immutable 特性，使得單向鏈結串列幾乎成為函數式程式語言最常見的集合型別之一。

基本操作如下：

1. New：初始化一個空串列。
2. Push\_front：新增節點到開頭的位置。(在 O(1) 時間內做完)

新增節點 push\_front 的概念很簡單，1）建立新的節點，並把新節點 next 指標指向串列第一個節點。2）把串列的 head 指向新建立的節點。

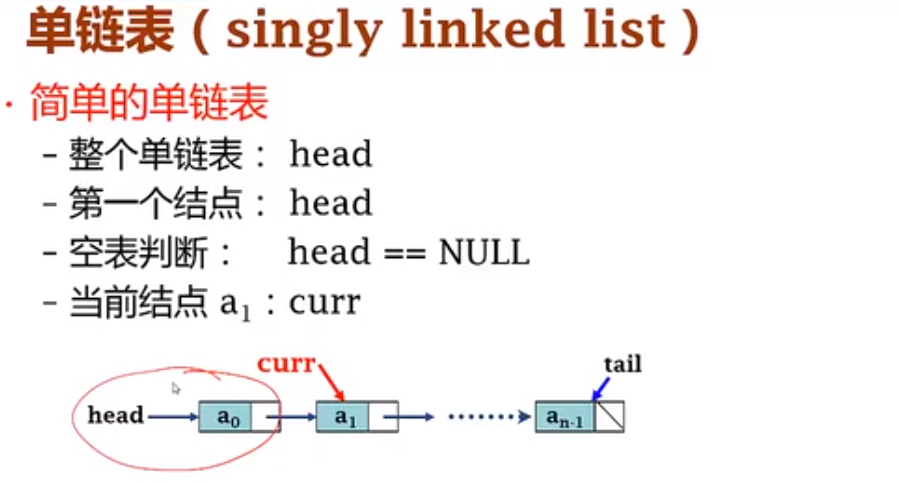
1. Pop\_front：將開頭第一個節點移除。(在 O(1) 時間內做完)

刪除第一個節點 pop\_front 的實作步驟如下：首先取得第一個節點的所有權，再將 head 指向第一個節點 Node.next 下一個節點，再返回第一個節點的資料給呼叫端。

1. Insert\_after：在指定索引位置後插入一個新節點。
2. Remove：移除任意索引下的節點。

插入/刪除任意節點沒有辦法在 O(1) 時間內做完，因為鏈結串列不支援隨機存取（random access），就是無法透過索引在常數時間內取得資料，每次的搜尋都只能從 head 開始。因此，當我們需要在某個索引的節點後新增/刪除一筆資料，需要最差 O(n) 的複雜度。

1. Clear：清除所有節點。
2. Is\_empty：檢查串列是否沒有任何節點。
3. reverse：反轉整個串列（head 變成 tail）。

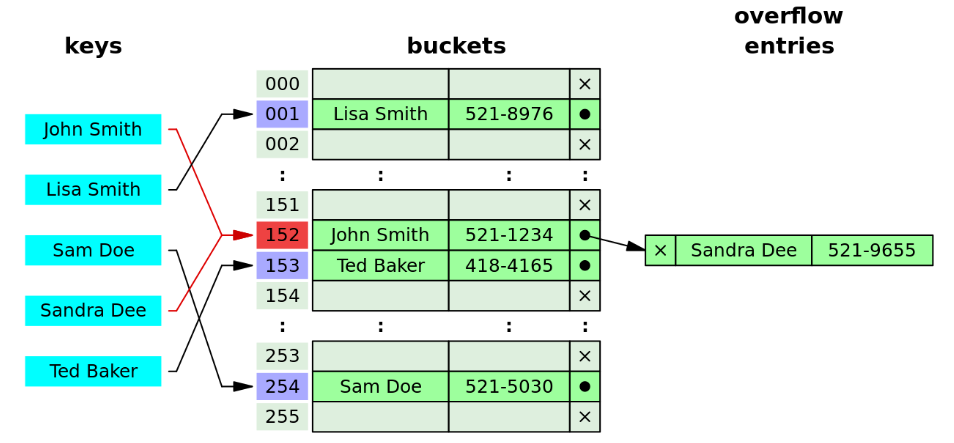




## **關聯容器 (Associative Container)**

關聯容器是一種抽象資料型別，儲存鍵與值配對關係（key-value pair）的集合，並透過鍵存取元素。關聯容器有許多別名，例如字典（dictionary）、關聯陣列（associative array）、映射（map）、表（table）等。在大多數程式語言函式庫中，關聯容器通常是最基本的容器型別之一，如 Python 的 dict，JavaScript 的 Map，以及 Rust 的 HashMap。

方便起見，本文以「映射表」統稱這類集合型別。



映射表特性：

1. 鍵值對為單向關係：可透過鍵取得其唯一值；但無法確保一值僅對應唯一的鍵。
2. 鍵值唯一性：同個映射表內，同個鍵不重複，只會出現一次。
3. 元素組合性：映射表內每個元素都是「鍵值對」，鍵或值無法單獨存在。
4. 操作開銷小：合理實作下，基本操作開銷相對較小，不高於線性時間。

註：多重映射表為一對多的例外。

映射表會有以下幾種基本操作：

1. 新增：配對鍵值關聯，又稱為綁定 binding。
2. 修改：修改任意鍵之下的值。
3. 移除：透過任意鍵移除該鍵值對，又稱 unbinding。
4. 查找：透過任意鍵搜尋該鍵值對。

不難看出，基本操作都是透過鍵取得值。事實上，合理實作的映射表，只要透過鍵來操作，就能有良好效能，甚至上述操作能達到 O(1)複雜度。

適用場景

雖然映射表依實作不同，效能有所權衡。但其最大優勢仍是可「高效地透過鍵尋找值」，只要有映射關係的資料，都非常適合使用映射表。例如，快取暫存機制需透過特定鍵快速查找暫存值。此外，現代常用的 JSON、TOML 等資料交換格式，都是「鍵—值對」的形式，非常適合使用映射表處理。而應用映射表最有名的實際案例莫過於資料庫的索引，透過索引，我們可以大大降低搜尋的成本，從線性時間直落到對數甚至常數時間，不過相對就需要付出額外時空間建立索引。

1. 有映射關係，處理「鍵—值」配對的資料結構。
2. 處理 JSON、TOML 等資料交換，資料序列化。
3. 實作快取（cache）機制。
4. 資料庫索引的實作方法之一。
5. 查找操作頻率遠高於其他操作時。

總的來說，只要資料有對應綁定關係，就可以考慮使用映射表處理。

### **雜湊表 Hash Map**

雜湊表是以雜湊函數實作的映射表。透過雜湊函數將資料轉換為固定長度的雜湊值(=鍵)(對應到容器內部的索引位置)，並將此雜湊值映射到對應資料的值（value）(內部資料結構的某位置)。理論上，只要雜湊函數品質過得去，雜湊表的基本操作都能在常數時間完成。

雜湊表是以雜湊函數實作的關聯容器。透過雜湊函數，計算鍵（key）對應到容器內部的索引位置，進而找到對應的值（value）。一般來說，雜湊表最常見的實作是以一個簡單陣列儲存資料。

雜湊表的優勢是：

1. 在資料量大時，仍然維持常數時間的高效能。
2. 若資料數量上限已知，就可避免重新配置記憶體，效能更佳。
3. 若資料形態已知，就可針對該資料形態找尋適合的雜湊函數最佳化。

而雜湊表相對有以下短處：

1. 資料量不夠大時，單一操作需要雜湊計算，開銷相對高。
2. 效能與雜湊函數息息相關，較差的函數容易雜湊碰撞，較佳函數計算成本通常較高。
3. 只能以某種偽隨機的順序疊代雜湊表。

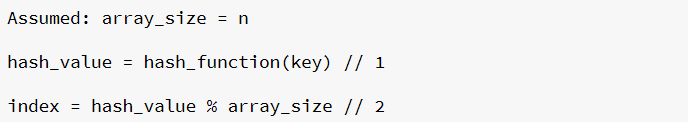
然而，你可能已經開始好奇了。

1. 雜湊是什麼？怎麼知道要映射到哪個索引位置？
2. 雜湊函數是否會計算出相同的索引值？要如何解決？
3. 若預先配置的陣列填滿了，該如何處理？

雜湊

所謂的雜湊函數，就是一種將「較寬的定義域映射到較窄值域」的函數。簡單來說，就是輸入任意值到此函數，則輸出值會落在一已知範圍。再白話一點，雜湊函數就是用來「化繁為簡」，把複雜多變的東西，透過函數生成簡化版本。此外，相同的輸入鍵，必須得到相同的輸出雜湊值，這是雜湊函數很重要的一個特性，以虛擬碼表示：

「映射」這部分只是使用雜湊的一小步。雜湊表根據程式實作的不同，底層儲存資料的形式也不盡相同，為了完全放入陣列中，通常會對雜湊值（雜湊函數的計算結果）取模（modulo）。也就是說：假設有長度為 n 的陣列。1）先對 key 取雜湊值。2）再對雜湊值取模，確認索引值落在陣列內部。



如此一來，所有可能的值都會落在陣列內，這就是最簡單普遍的雜湊兩步驟：計算雜湊值﹢取模。

**選擇雜湊函數**

接下來，你會緊接著向問第二個問題「函數計算出相同索引值該怎麼辦？」不同輸入產生相同雜湊值，多個值映射到同個索引上，這種狀況科學家稱之雜湊碰撞（hash collision）。

首先，要瞭解雜湊函數本身就是時空間的權衡，如果記憶體空間夠多，那讓輸入值與雜湊值呈一對一的完美關係，就不會出現碰撞；大多數情況，尤其是實作泛用的雜湊函式庫，無法預期輸入資料的範圍，實務上會鎖定一個輸出雜湊值的範圍，僧多粥少，難免碰撞。

好的雜湊函數還必須符合一些條件：

1. 同一筆輸入資料，必須得到相同的雜湊值。
2. 結果必須能夠高效的計算出來（預期為常數時間）。
3. 任意輸入資料所得之雜湊值在值域內需接近均勻分佈（uniform distribution），才能減少碰撞機率。

但總歸一句，欲達成上述條件，就是一種權衡取捨，例如，加密雜湊函數（cryptographic hash function）即是非常優秀的雜湊函數，但相對需付出更高的計算成本。

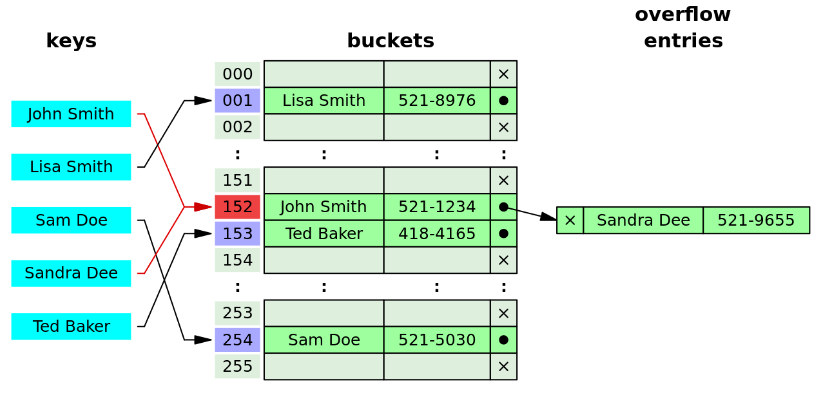
**處理雜湊碰撞**

既然雜湊函數人生在世難免碰撞，科學家也研究幾個處理雜湊碰撞的策略，分別是 separate chaining 與 open addressing。

**Separate chaining (分離鏈接法)** 可以說是最直觀的做法，就是設法讓同一個索引下，可以儲存多個碰撞的值。依據儲存資料的形式，可分為幾種實作：

1. 鏈結串列：以鏈結串列（linked list）儲存元素。發生碰撞時，新的元素串接在既有元素之後。
2. 動態陣列：新增元素時，在該位址配置動態陣列（dynamic array）儲存元素。發生碰撞時，直接將新元素加在陣列尾端。

不同實作方式有各自優缺點，例如串列版本容易實作，但需額外儲存指標資訊；用動態陣列，則會有更好的 CPU caching，但相對地碰撞過多則需要重配置陣列。

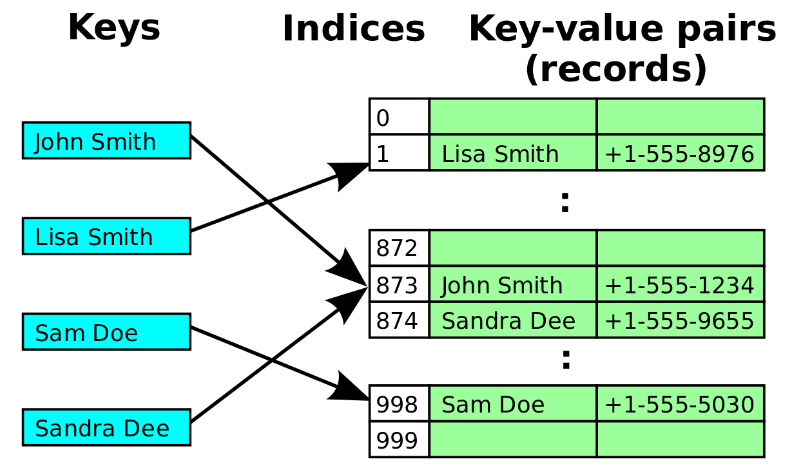


**Open addressing (開放定址)** 則走完全不同的套路，不額外配置儲存空間給碰撞的元素，而是繼續在同個陣列內「探測」其他可用的 slot，再把資料塞進尚未被佔據的 slot 中。而 Open addressing 依據不同探測序列（probe sequence）有不同實作，常見的有：

1. Linear probing：從發生碰撞索引開始，依序往下一個 slot 探測是否可用，因此得名「線性」。
2. Quadratic probing：從碰撞索引開始，間隔以二次式增加往下探測可用 slot，如 i+1^2,i+2^2,i+3^2。
3. Double hashing：以固定間隔大小 k（probe distance），依序探測 i+k,i+k⋅2... 的 slot 是否為空。而這個間隔是以另外一個雜湊函數計算所得，因此得名「雙雜湊」。

這些方法的差異主要在於 CPU caching 的效能，以及 HashMap 資料的群聚效應（clustering）的敏感程度。當然，論 caching 絕對非 linear probing 莫屬，但 linear probing 以線性一個挨一個探勘，效能較容易受雜湊值群聚影響。

以下是 linear probing（間隔 = 1）的示意圖。



**動態調整雜湊表大小**

若資料的筆數已知，那初始配置的陣列大小設定與資料筆數成比例，就不必擔心雜湊表空間不夠，需要重新配置（reallocate）儲存空間的困擾。

倘若資料量未知，而最初配置的 bucket array 滿了，該如何重新配置呢？

動態調整大小對雜湊表來說，不同於一般動態陣列，舊的雜湊表若要對應到新雜湊表，是每個鍵都需要重新計算雜湊值（rehash），成本相對較高。因此，減少動態調整的次數，可說是調教雜湊表的重點之一。說到調教雜湊表，必定要瞭解一個重要指標：load factor。



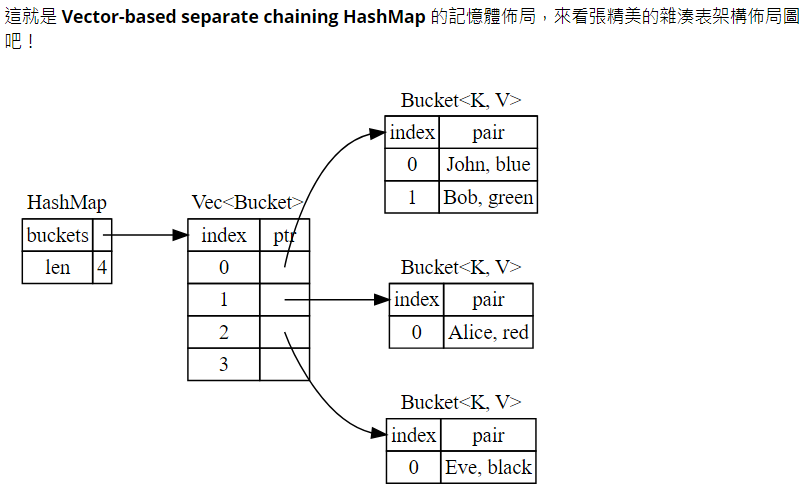
n：已放入雜湊表內的資料總數。

k：雜湊表配置的儲存空間（bucket 總數）。

Load factor 代表目前雜湊表的「使用率」，若三筆資料放在四個 bucket 內，則 load factor 為 3/4=75。

Load factor 太大會更容易碰撞，會有效能上的影響；太小則代表過多冗餘空間沒有使用。如何維持 load factor 在一定範圍內至關重要。一般來說，75% 的 load factor 就可以準備重新配置雜湊表了，當然，這個門檻仍要以實作經驗為主，例如 Rust 的 HashMap 使用了 Robin Hood Hashing，將 load factor 調教到 90%。

重配置雜湊表與動態陣列的動態調整大小雷同，達到某個門檻值，就會將底層陣列大小翻倍。為了避免開銷過高，通常元素減少時，不會主動調整大小，而是提供一個 shrink\_to\_fit 一類的方法，讓呼叫端自行決定釋放多餘空間的時機。



雜湊表有以下幾個基本操作：

1. new：初始化一個空雜湊表。
2. with\_capacity：配置特定數量 bucket 的雜湊表。
3. get：取得指定鍵對應的資料。
4. get\_mut：取得指定鍵對應的資料，並可寫入修改（mutable）。
5. insert：在任意位置插入一組鍵值對。
6. remove：移除任意位置下的鍵值對。
7. clear：清除所有鍵值對。
8. is\_empty：檢查雜湊表是否沒有任何鍵值對。
9. len：檢查目前鍵值對的數目。
10. bucket\_count：檢查目前 bucket 的數目。

以及幾個內部方法：

1. try\_resize：根據給定條件，決定調整 bucket 數目的時機，讓 load factor 維持最適狀態。
2. make\_hash：從輸入資料產生雜湊值，再模除 bucket 數，得到輸入資料對應的索引位置。

**時間複雜度**

在預期情況下，只要雜湊函數品質穩定，大部分操作都可達到在常數時間， 但由於部分操作，尤其是新增或刪除元素的操作，會需要調整 bucket array 的空間，重新配置記憶體空間，所以需要平攤計算複雜度。

最差複雜度出現在每個元素都發生雜湊碰撞。

1. 若使用 open addressing 處理碰撞，則會把雜湊表配置的每個位置都填滿，而所有操作都從同個位置開始，搜尋對應的鍵，複雜度與陣列的線性搜索相同為 O(n)
2. 若使用 separate chaining，碰撞代表所有元素都會在同一個 bucket 裡面，也就是只有一個 bucket 上會有一個長度為 n ，被塞滿的陣列或鏈結串列，結果同樣是線性搜索的 O(n)

我們嘗試使用數學表示搜索的複雜度。另

n：已放入雜湊表內的資料總數。

k：雜湊表配置的儲存空間（bucket 總數）。

load factor=n/k：預期每個 bucket 儲存的資料筆數。

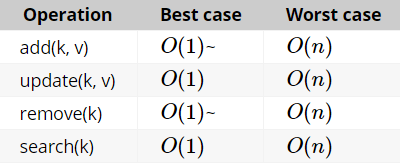
則預期執行時間為



而 1 為計算雜湊與取得索引（random access）的執行時間，nk 則是搜尋陣列的執行時間。只要 load factor 越接近 n，執行時間就相對增加。

**空間複雜度**

雜湊表的空間複雜度取決於實作預先配置的陣列大小，並與維持 load factor 息息相關。一般來說，仍與資料筆數成線性關係，因此空間複雜度只有資料本身 O(n)。而以 separate chaining 會額外配置陣列或鏈結串列儲存碰撞元素，理論上需負擔更多額外的指標儲存空間。



### **有序映射表 Ordered Map**

有序映射表係一種有特定排序方式的映射表。常見兩種排序方式，其一是依照插入映射表的先後順序；其二則是依照鍵的大小。不同排序的底層資料結構各異，操作複雜度也不盡相同，如依鍵大小排序的映射表通常使用搜索樹實作，因此「新增」操作的複雜度為較差的 O(logn)。

### **多重映射表 Multimap**

多重映射表允許鍵值對重複，一個鍵可對應多個值（一對多）。類似於映射表內放入陣列，但能以較方便輕鬆的介面來操作或疊代整張映射表。

### **集合 Set**

集合是一種抽象資料型別（Abstract data type），概念衍生自數學的集合論，和陣列類似，為不同元素所組成的資料結構，不同在於集合有些重要的特性：

* 無序性：集合內各元素無特定排序或排序不重要。
* 互異性：集合內每個元素且只能出現一次。

一般來說，集合的實作會盡量貼近集合論中的有限集合定義，本次實作同樣遵照數學定義。

**架構設計**

1. 以雜湊表為底層容器

集合能以許多不同的資料結構實現，通用的集合實作多會最佳化取得、增加、移除元素等運算，這讓我們想到了「雜湊表」，雜湊表正是能將集合運算最佳化的一種資料結構，我們將借用雜湊表作為底層儲存容器，進一步實作集合。

你可能好奇，集合明明只有元素，並沒有鍵值對，跟雜湊表有啥關係？讓我們回想雜湊表一個重要的特性：每一個鍵（key）只會出現一次，利用雜湊表這個特性，即可達成上述集合兩大特性。

既然選用雜湊表作為底層容器，集合的結構體就非常簡單了，我們可以將集合想像為在 HashMap 薄薄的一層封裝。

1. 不佔空間的 Unit Type

這樣多儲存一個 () 會不會造成額外的儲存空間負擔？事實上，Unit type () 在 Rust 語言中是一種 Zero Sized Types (ZSTs)，在編譯時期 Rust 會將 ZST 當作一種型別來操作，但真正輸出的機器碼中，ZST 並不會佔用任何運算空間。Set = HashMap<T, ()> 完全體現了 Rust 零成本抽象的語言特性。

基本操作

1. new：初始化一個集合。
2. contains：檢查集合內有無特定元素。
3. is\_empty：檢查集合內是否沒有任何元素。
4. insert：新增一個元素。
5. remove：移除特定元素。
6. len：檢查集合內的元素數目。
7. iter：產生一個疊代集合內所有元素的疊代器。

**集合運算**

電腦科學的集合型別衍生自集合論，當然也得符合集合代數（set algebra）的特性，和算術的加減乘除，集合也有自己的二元運算，我們會實作以下幾個基本方法：

1. intersection：交集，A ∩ B 定義為 A 與 B 共有的元素。
2. union：聯集，A ∪ B 定義為 A 與 B 所有的元素。
3. symmetric\_difference：對稱差集，定義為 A △ B = (A ∪ B) - (A ∩ B)，就是只出現在 A 或 B，不會在兩集合內同時出現的元素。
4. difference：差集，A \ B 定義為 A 中所有未在 B 中出現的元素。
5. is\_disjoint：兩集合是否不交集。
6. is\_subset：包含於 ⊆，是否為子集。
7. is\_superset：包含 ⊇，是否為超集。

運算子多載

Rust 提供許多 trait 供使用者多載運算子，可以簡化集合的二元運算：

1. set | other：bitwise or 運算子，效果同 union 聯集。
2. set & other：bitwise and 效果同 intersection 交集。
3. set - other：substraction 運算子，效果同 difference 差集。
4. set ^ other：bitwise xor 運算子，效果同 symmetric\_difference 對稱差集。

比較運算子

除了交集、聯集等運算，我們還可以實作集合間的比較，作為檢查是否為子集或超集的運算子。

1. A <= B：效果同 is\_subset，A 是否為 B 的子集 A ⊆ B。
2. A < B：A 是否為 B 的子集且不等於 B，等同於 A ⊂ B。
3. A >= B：效果同 is\_superset，A 是否為 B 的超集 A ⊇ B。
4. A > B：A 是否為 B 的超集且不等於 B，等同於 A ⊃ B。

但眼尖的 Rustacean 肯定會發現，std::ops 裡面根本沒有 lt、gt 等比較運算子。Rust 的「比較」是透過實作幾個 Trait 後，自動推導生成的方法，這些 trait 放在 std::cmp module 中，分別是Eq、ParitalEq、Ord，以及 ParitalOrd。

在開始介紹如何實作比較前，先讓複習一下離散數學學到的二元關係：

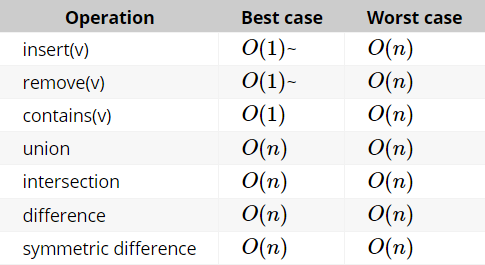
若 ∼ 為一種二元關係，A 為任意集合。

自反性（Reflexive）：對所有 x ∈ A : x ∼ x。

對稱性（Symmetric）：對所有 x, y ∈ A ，若 x ∼ y 則 y ∼ x。

傳遞性（Transitive）：對所有 x, y, z ∈ A ，若 x ∼ y 且 y ∼ z 則 x ∼ z。

反對稱（Antisymmetric）：對所有 x, y ∈ A，若 x ∼ y 且 x ≠ y 則 y ∼ x 不成立。



### **布隆過濾器 Bloom Filter**

布隆過濾器是一種機率資料結構（probabilistic data structure），類似於集合，但只會回報「絕對不存在」或「可能存在」的機率資料結構，實作上節省空間，常用於在海量資料中確認成員是否存在，並能有一定容錯率的場景。有機會出現假陽性（false positive），但絕不會有假陰性（false negative）。

Bloom filter 的優勢是：

1. 類似集合，可在 O(1) 時間複雜度驗證成員是否存在，卻僅需相對少的儲存空間。
2. 承上，在 0.1% 錯誤率下儲存一百萬個元素僅需 1.71 MiB。
3. 非常容易實作的機率資料結構，僅需多次雜湊。

Bloom filter 則有以下短處：

1. 經典款 Bloom filter 容器大小固定（fixed-size），無法動態調整儲存空間。
2. 可能給出假陽性答案：回報存在但實際不存在，且錯誤隨數量變多上升。
3. 自身不儲存成員資料，需要有額外的儲存資料方案。
4. 只能新增成員，但不能移除成員（可透過變形解決）。
5. 若輸入資料集本身離散，接近隨機存取，無法充分利用 CPU cache。
6. 承上，因為隨機存取，不利於延伸到記憶體以外的外部儲存裝置。

Bloom filter 常見應用場景為：

1. 資料庫利用 Bloom filter 中減少實際存取 disk 的 IO 開銷。
2. Chromium 瀏覽器驗證大量惡意連結。
3. Medium 避免推薦已推薦過的文章。

概念

Bloom filter 由下列兩個部分組成：

1. 一個 m 位元的位元陣列（bit array）
2. k 個不同的雜湊函數

經典款的 Bloom filter 作為一個近似集合的容器，提供下列兩個操作

1. 新增： 新增一個值時，透過 k 個雜湊函數產生 k 個雜湊值，分別代表在位元陣列的索引位置，再將 k 個位置的位元翻轉至 1。
2. 查詢： 同樣透過 k 個雜湊函數產生 k 個雜湊值作為位元陣列的索引位置，若所有位元皆為 1，則代表該值存在。

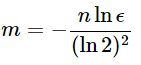
你可能會開始想：

* 欲儲存一百萬個元素，需要多少位元？
* 出現假陽性的機率是多少？可以調整嗎？
* 需要幾個雜湊函數？
* 可重複使用相同的雜湊函數嗎？

回答這些問題需要兩個已知條件：

1. 預期會儲存多少 n 個元素到容器。
2. 可容忍的假陽性機率 ϵ，即容器不包含該元素，檢測卻回報存在（所有雜湊位皆為 1）。

於是可得位元陣列最佳化的長度為 m 個位元，m 為：



而在已知條件下，需要的雜湊函數數量 k 為：



當然，這些公式並非憑空冒出，有興趣可以讀讀維基百科上的數學，和這段詳細的推導，不過也要注意，Bloom filter 的假設是「每個雜湊函數獨立」但位元間是否獨立有待討論，這順便開啟了其他問題，可重複使用相同的雜湊函數嗎？

答案是可以， 這篇「Less Hashing, Same Performance:Building a Better Bloom Filter」提及，在不犧牲漸進假陽性機率（asymptotic false positive probability）的前提下，透過兩個不同的雜湊函數 h1(x) 和 h2(x)，配合以下公式，就可以模擬出多個雜湊函數：



基本操作

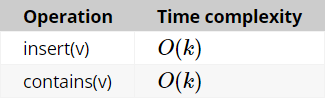
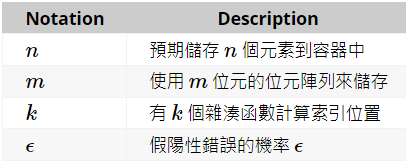
Bloom filter 為類似集合的容器，當然有類似的操作，事實上，這類機率性集合成員檢測的資料結構有個較少聽見但令人印象深刻的名字，稱為「Approximate Membership Query（AMQ）」，提供 add(element) 和 query(element) 兩個基本操作。

本文的 BloomFilter 提供下列幾個公開方法：

1. new：初始化一個容器。
2. insert：新增一個元素。
3. contains：檢查容器內有無特定元素（是否曾新增過）。

以及幾個內部輔助方法：

1. make\_hash：給定輸入元素資料，透過兩個雜湊函數產生兩個雜湊值。
2. get\_index：將 make\_hash 的兩雜湊值帶入 gi(x)=h1(x)+ih2(x) 計算單次 i 的索引位置。
3. optimal\_bits\_count：給定預期儲存元素個數 n 與假陽性機率 ϵ，得位元陣列最適位元數 m。
4. optimal\_hashers\_count：給定預期假陽性機率，得最適雜湊函數個數 k。



而儲存空間複雜度則是 O(m)。

新增和搜尋一個元素個別需要雜湊 k 次，因此時間複雜度為 O(k) 顯而易見，然而，k 通常相對 m n 是非常小的數字，例如 在 0.1% 錯誤率下儲存一百萬個元素僅需 1.71 MiB 和 7 個雜湊函數，實務上直接當作 O(1) 也不算錯。

至於空間複雜度，由於必須事先配置好 m 位元的位元陣列，就算新增的元素 n>m，也不會再新增新位元，因此空間使用為 O(m)個位元。實務上，當 n 成長到接近 m 時，假陽性的機率會大增，不堪使用，進而需要能動態調整儲存空間的 Bloom filter 變形。

經典款 Bloom filter 容易實作，歷久不衰，不過仍有許多可以增進空間：

* Data locality 不夠好：Bloom filter 底層儲存是位元陣列隨機存取，較不符合現代 CPU 架構的 cache line 使用姿勢。Cloudflare 技術部落格文 When Bloom filters don’t bloom 以幽默筆法帶出這個問題，值得一讀。
* 雜湊次數過多：Bloom filter 每一個新增查詢操作都需要雜湊 k 次，就算利用 double hashing 還是要雜湊兩次，比起其他類似資料結構硬生生多雜湊數次。
* 位元陣列大小固定：Bloom filter 容器大小固定，給你預期的元素個數後，無法動態調整儲存空間，
* bits per entry 較高：以類似功能的的資料結構來說，Bloom filter 在空間利用率上，欲維持一定的假陽性機率，每個元素所需位元數相對較高，需要個位元。

變形

1. **可以計數的 Counting Bloom filter**

經典款 Bloom filter 之所以無法刪除元素，是因為沒有記錄哪些元素新增/刪除的資訊，而 Counting Bloom filter 顧名思義，原本用一個位元儲存 0 / 1 資訊，延伸為多位元方便儲存計數（counting），有了個別元素增刪資訊，Bloom filter 因此能實作「刪除元素」。搜尋一個 Counting Bloom filter 是否擁有 n 次以上 x，答案一樣和 Bloom filter 類似是「可能有 n 次以上的 x」或是「x 絕對沒有 n 次以上」。事實上，可將 Counting Bloom filter 視為 Bloom filter 的一般化形式（generalized form），而經典款 Bloom filter 反過來可當作只記一次數的特化。

但 Counting Bloom filter 的缺點是空間需求大，端看決定要用幾個位元計數，例如常見用 4 個位元計數，則是經典款的四倍空間消耗。

1. **動態適應空間大小的 Scalable Bloom Filter**

Scalable Bloom Filter 的特色是：動態適應空間大小，不需事先知道預期儲存的元素個數。

Scalable Bloom Filter 的實作蠻暴力的，本身是由一至多個經典款 Bloom filter 組成，若一個 filter 滿了（超過 fill ratio），則會新增一個 filter，往後所有新增都在這個新 filter 上面，直到它也滿了，可視為一個 recursive data structure。

至於查詢，這就是 Scalable Bloom Filter 比較弱的地方，查詢會從第一個 filter 開始找，若找不到往下一個 filter 找，找到會沒有下一個 filter 為止。若 filter 數量為 l，則查詢的時間複雜度從 O(k)

變成 O(k⋅l)。

除了初始化大小和假陽性機率率，Scalable Bloom Filter 提供設定成長率和假陽性錯誤緊縮率：

* 成長因子 s：每個新增的 filter 空間大小成長率，論文的經驗法則得出預期小成長趨勢選擇 s=2，有較大成長趨勢則 s=4 效果好。
* 錯誤緊縮率 r： 每個新增的 filter 會以等比級數得到更緊縮的假陽性機率上限，由於是等比級數，逼近極限時會小於原始機率，這讓整體假陽性機率得以保持。論文中實證 0.8 到 0.9 在元素預期有大成長率下有最佳平均空間利用率。

1. **Quotient filter**

商數過濾器（Quotient filterF）利用雜湊表為底層儲存容器，來做集合成員檢測的 AMQ，為了節省空間使用量，Quotient filter 的雜湊表只儲存 partial-key，俗稱指紋（fingerprint），指紋的鍵短空間用量低，副作用是更容易碰撞，代表需要更有效處理雜湊碰撞（hash collision）。

一般來說，處理雜湊碰撞有 separate chaining 和 Open addressping 兩大類方法，而 Quotient filter 選擇了另一條詭譎的方法：利用 open addressing 中 linear probing 的方式，對每個 slot 儲存額外資訊，使得我們可辨認碰撞的元素是在相同指紋下的同個 bucket 內。換句話說，額外資訊就是在「透過 linear probing 模擬 separate chaining」。

回到指紋，Quotient filter 實際上並不直接儲存指紋，而是將指紋 f 進一步拆成商 fq 與餘數 fr，商作為索引位置，而餘數則為真實被儲存的值。透過商和餘數，可重組回推原本的指紋。不需存完整的指紋，又再次減少空間使用量，帥！

1. **支援刪除元素的 Cuckoo filter**

📚 論文連結（有趣易讀，誠摯推薦）

Cuckoo hashing 是一種解決雜湊碰撞的方法，透過一次計算兩個雜湊函數產生兩個索引位置，若其中一個位置有空位則插入空位，若都沒有空位，則隨機踢掉一個，被踢掉的再去找下一個替死鬼，直到全部都有位置，或踢掉次數大於一定值則停止。這種行為和杜鵑鳥（cuckoo、布穀鳥）鳩佔鵲巢的生物習性很像，因此得名。

Cuckoo filter 利用雜湊表為底層儲存容器，來做集合成員檢測的 AMQ，會和 cuckoo 扯上關係則是因為使用 Cuckoo hashing 解決雜湊碰撞，以增加空間使用率（達到 95% occupancy）。Cuckoo filter 的雜湊表和 Quotient filter 一樣，為了減少空間使用量而只儲存 partial-key。

## **字串處理**

### **漢明距離（Hamming distance）**

漢明距離（Hamming distance）是指兩個相同長度的序列（sequence）在相同位置上，有多少個數值不同，對二進位序列來說就是「相異位元的數目」。漢明距離同時也是一種編輯距離，即是將一個字串轉換成另一個字串，需要經過多少次置換操作（substitute）。

漢明距離多應用於編碼理論中的錯誤更正（error-correcting），漢明碼（Hammming code）中計算距離的演算法即為漢明距離。

長度為 n 的序列，計算漢明距離的時間複雜度為 O(n) ，空間複雜度為 O(1) 。

### **萊文斯坦距離 Levenshtein distance**

萊文斯坦距離（Levenshtein distance）是一種量化兩字串間差異的演算法，代表從一個字串轉換為另一個字串最少需要多少次編輯操作，這種量化指標的演算法稱為「編輯距離（Edit distance）」，不同的演算法允許的編輯操作不盡相同，萊文斯坦距離允許使用：

1. 插入（insertion）
2. 刪除（deletion）
3. 置換（substitution）

三種編輯操作，通常一般實作上三種操作的權重會相同。

萊文斯坦距離概念易理解，實作簡單，常用在簡易拼字修正與建議，為最具代表性的編輯距離演算法。

萊文斯坦的遞迴函數

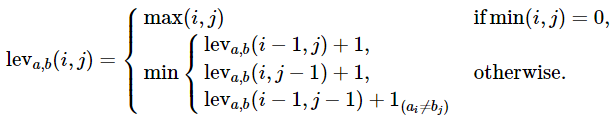
編輯距離愛拿 kitten 和 sitting 當例子，這兩個單字的編輯距離為 3，可以透過以下步驟，算出 kitten 與 sitting 轉換會經歷三個編輯操作：

sitting -> kitting：置換 /k

kitting -> kitteng：置換 i/e

kitteng -> kitten g：刪除最末端 g

等等，怎麼知道 3 就是最少編輯操作呢？我們可以經由下列函數計算萊文斯坦距離：



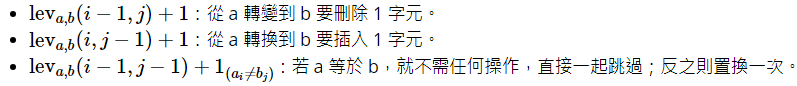
別怕，這個函數是一個分段函數（piecewise function），由兩個子函數組成，接下來將一一拆解。

首先來看第一個子函數：



max(i,j)ifmin(i,j)=0根據函數定義，i j 可以視為 a b 前幾個字元的子字串（substring），所以這個子函數白話翻譯是「若有子字串是空字串，則以較長的子字串長度作為編輯距離」。這其實非常直觀，如果有一空字串和另一個非空字串 abc，那編輯距離一定是插入三次或刪除三次，也就是該字串長度。這帶出萊文斯坦距離一個很重要的上界：「兩字串的編輯距離至多為較長字串的長度」。

第二個子函數稍微複雜，要再從三個函數中取最小值，但剛剛好，這三個函數分別代表了萊文斯坦距離接受的插入、刪除、置換三種操作：



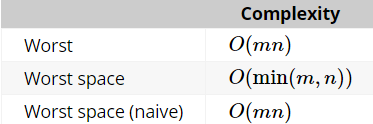
這是子函數是遞迴函數，每次都會分出三個子程序計算萊文斯坦距離，空間複雜度直逼 O(3m+n−1)，驚人！

**動態規劃的距離矩陣**

由於上述遞迴複雜度過搞，因此處理萊文斯坦距離，通常會選擇由下至上（bottom-up） 的動態規劃（Dynamic programming），利用一個距離矩陣，將兩個字串所有子字串間的萊文斯坦距離累積紀錄起來，省去重複計算的成本。

首先，先將最基礎的 a、b 其一為空字串時的編輯距離寫上去，因為其中一個為空字串，編輯距離必定是隨著另一字串增加，逐一插入字元（最左方的直行）或刪除字元（上方橫列）。

顯而易見，萊文斯坦距離最差時間複雜度就是內外兩個迴圈疊代兩個字串的所有字元。而空間複雜度原本是 m⋅n 的矩陣，在最佳化動態規劃後，只需兩字串長度 m 或 n 中最小值長度陣列作為儲存空間。



## 

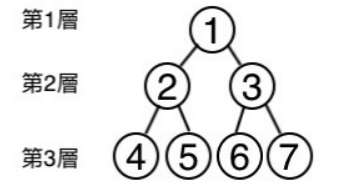
## **樹 (Tree)**

<https://www.youtube.com/watch?v=Fj9qy9bea5s&list=PLAWtuOYMqk9n-s3U6tEfNA-A5btw8tmKW&index=16>

<https://ithelp.ithome.com.tw/articles/10293888>

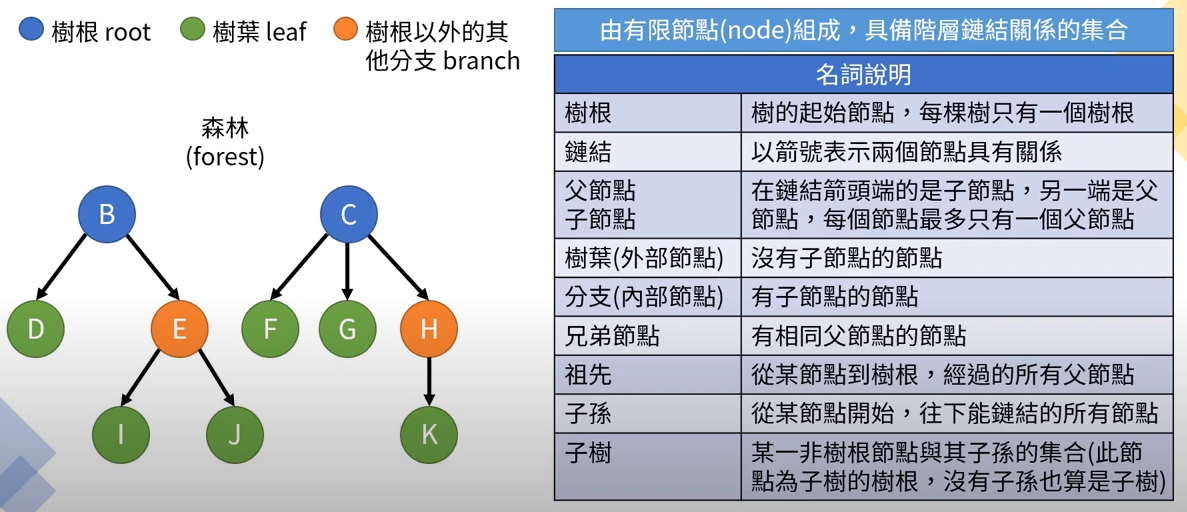
特性：

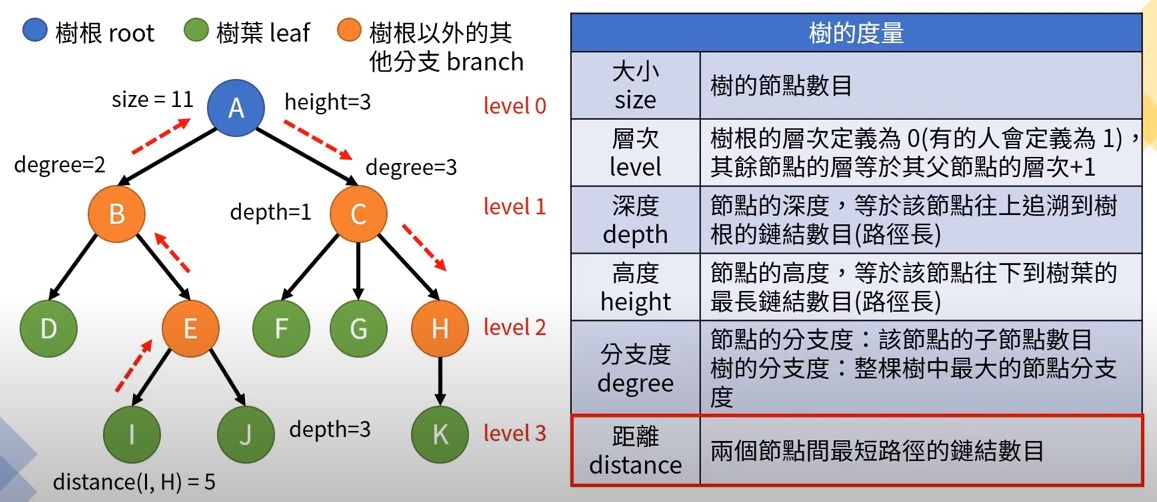
1. 樹是由有限數目的節點(node)組成的集合
2. 不包含迴路(「循環」 (Cycle))。
3. 如果存在「循環」在內，那他就叫「圖」。因此我們可以稱「樹是一種圖」。
4. 一棵樹中的任意兩個節點有且僅唯一的一條路徑連通。
5. 一棵樹如果有 n 個節點，那一定恰好有 n-1 條邊。
6. 如果一棵樹所有節點的 BSTNode \*left / \*right 都為 NULL，那這棵樹其實可以簡化為「鏈結串列」



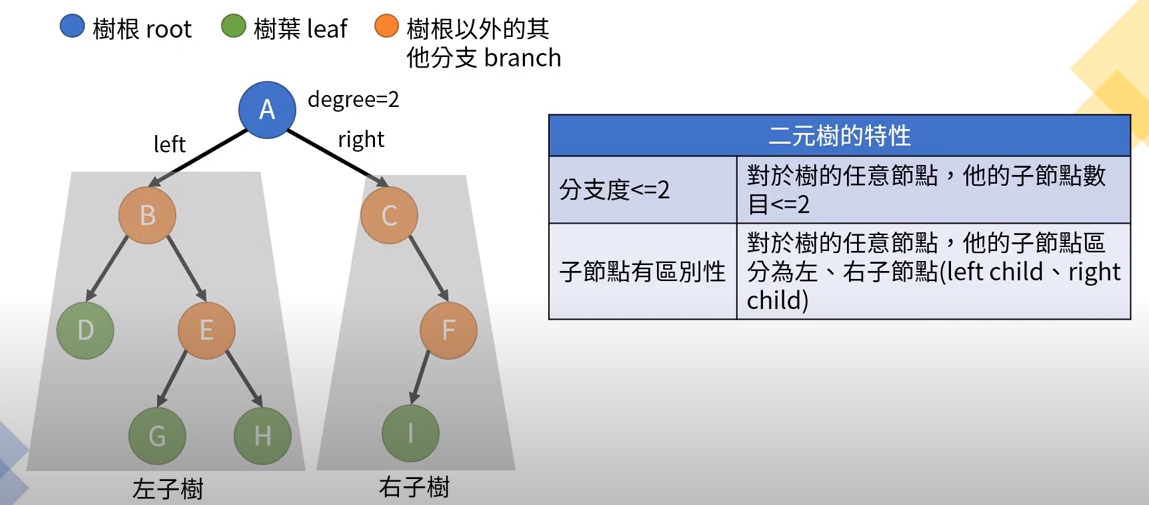
### **樹的結構**

1. 節點：圖上的每個點。
2. 根節點/根/祖先：一棵樹只有一個根節點，沒有父節點的就稱為根節點。
3. 葉節點：如果節點不能往下延伸，便稱為葉節點。例如 4、5、6、7 號。
4. 父節點：如果節點能往下沿伸其他節點，那此節點就稱為父節點。例如 2 是 4 和 5 號的父節點。
5. 子節點：承接父節點的節點，就稱子節點。例如 4 和 5 號是 2 號的子節點。
6. 兄弟節點：
7. 深度：每一個節點都有深度，第一層從根節點開始算起。例如 4 號的深度是 3。

****

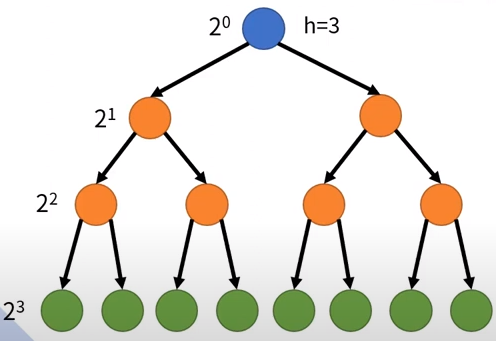
****

## **二元樹**

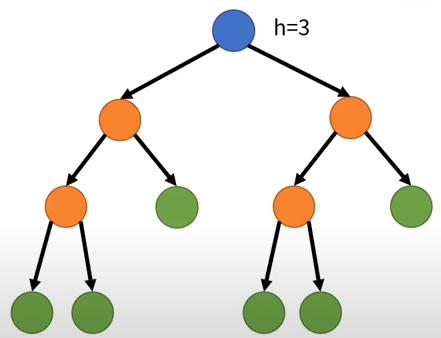
****

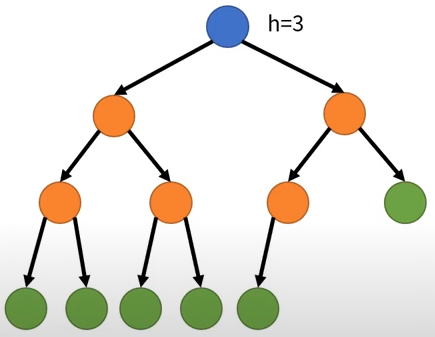
### **二元樹種類**

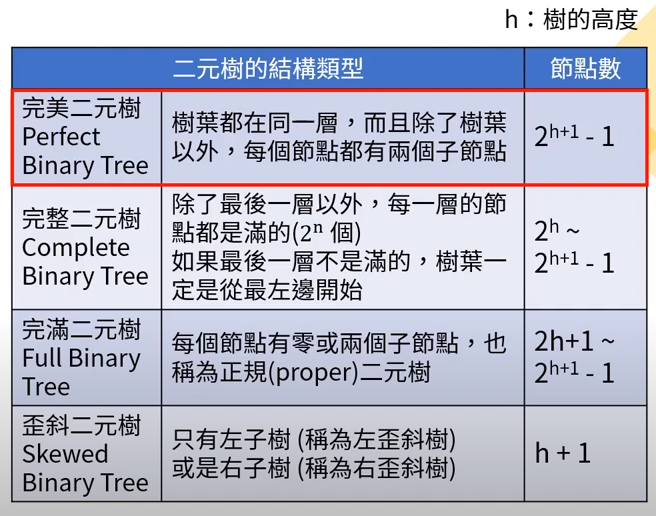
1. perfect binary tree ：各層節點全滿。同時也是 full binary tree 和 complete binary tree 。

****

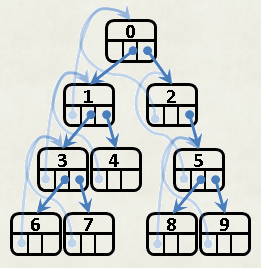
1. full binary tree ：除了樹葉以外，每個節點都有兩個小孩。

****

1. complete binary tree ：各層節點全滿，除了最後一層，最後一層節點全部靠左****

****

第一種方式：建立節點，以指標串接節點。



**struct Node**

**{**

**Node\* parent;**

**Node\* left;**

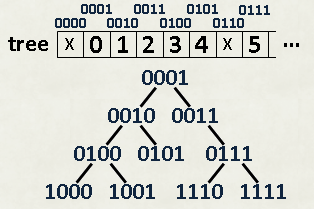
**Node\* right;**

**int data;**

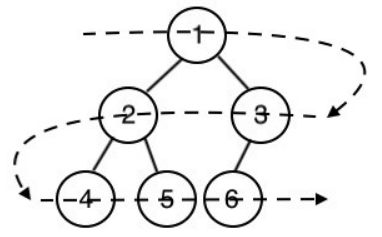
**};**

**Node\* root = 0;**

第二種方式：二進位數字一一對應到二元樹的節點。



1. 第三種 : 一個一維陣列把完全二元樹的資料儲存。
2. 首先將完全二元樹進行編號，從上到下，從左到右。



[1, 2, 3, 4, 5, 6]

發現編號和一維陣列儲存的規律。

1. 父節點編號為 k
2. 左兒子的編號為 2＊k
3. 右兒子的編號為 2＊k+1
4. 如果兒子編號為 x ，父節點的編號就為 x/2 取整數的商。
5. 如果有 N 個節點，則完全二元樹的高度(深度)為

## [**二元樹的走訪 (深度優先搜尋(Depth-First-Search(DFS))、廣度優先搜尋) :**](https://www.shubo.io/iterative-binary-tree-traversal/)

### **深度優先搜尋(Depth-First-Search(DFS)**

深度搜尋 (Depth-first Search)，透過函數的遞迴來一層一層縱向地找出所有可能，但程式碼卻會簡潔的多。

#### **前序走訪(Preorder Traversal)**

先拜訪節點N、再拜訪左子樹、最後拜訪右子樹

#### **中序走訪(inorder Traversal)**

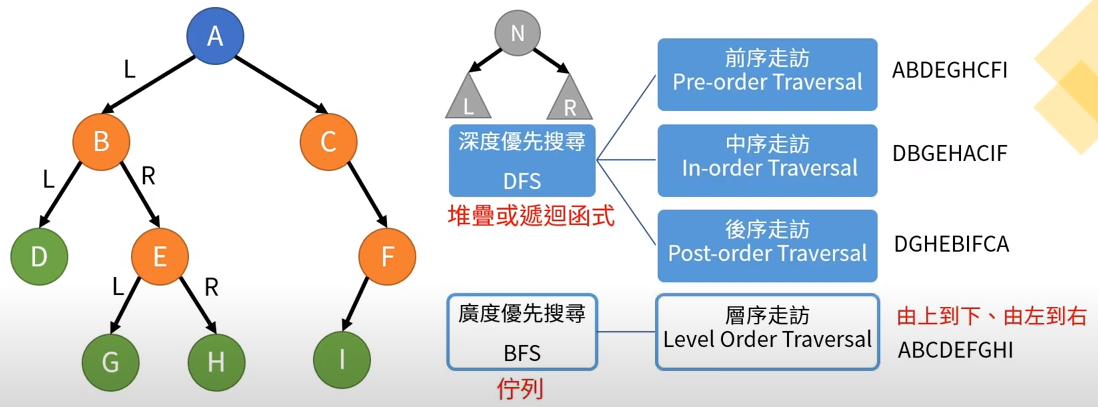
先拜訪左子樹、再拜訪節點N、最後拜訪右子樹

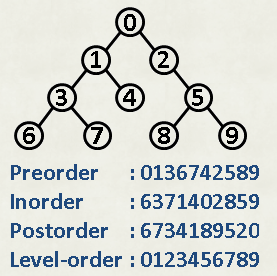
#### **後序走訪(postorder Traversal)**

先拜訪左子樹、再拜訪右子樹、最後拜訪節點N

### **廣度優先搜尋 (Breadth-first search(BFS))**

1. 透過「一層一層」橫向擴展的方式來搜尋。
2. 層序走訪 : 由上到下、由左至右

****

****

### **二元搜尋樹 (Binary Search Tree)**

二元搜尋樹 (Binary Search Tree)，又稱 有序二元樹 或 排序二元樹。如若不是空樹，則有以下幾個特點：

* 若任意節點的左子樹不空，則左子樹上所有節點的值均小於它的根節點的值
* 若任意節點的右子樹不空，則右子樹上所有節點的值均大於它的根節點的值
* 任一個節點的左子樹都比父節點小，右子樹都比父節點大，且每一個節點的值都不重複
* 任意節點的左、右子樹也分別為二元搜尋樹
* 沒有鍵值相等的節點

節點的搜尋

* 要查找資料的時候，就可以從根節點開始，比根節點小的就從左子樹開始找，比較大的就從右子樹開始找

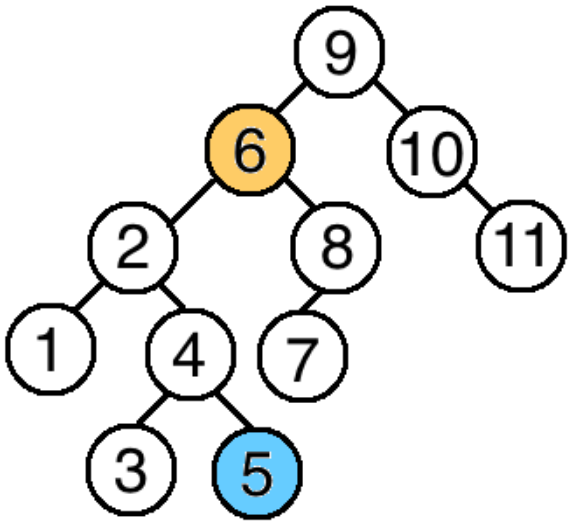
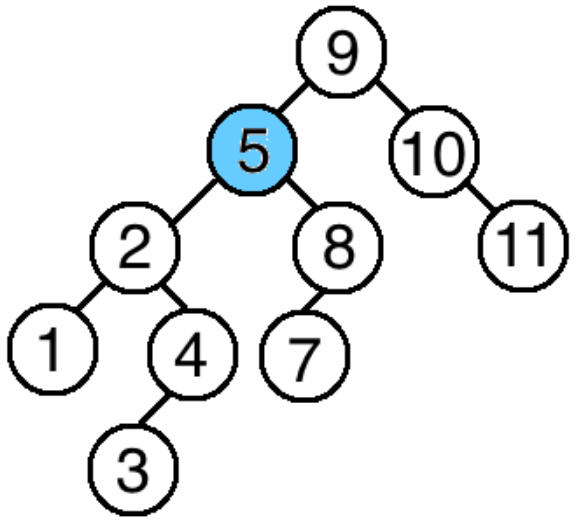
節點的插入

* 和搜尋很像，從根節點開始比大小，比較小的往左子樹走，比較大的往右子樹走，一樣的則返回，直到插入為止。

刪除節點

* 比較複雜。刪除的節點要從右子樹中找到對大的值來接替。

假設我們要刪除節點 6，就要找節點 5 來替換。如果找 2 的話，那 3~5 都比二大，卻在 2 的左子樹中，於定義相左。因此我們要找 6 的左子樹中，最大的節點來接替。

 ⇒ 

尋找、插入的時間複雜度較低，為Ｏ(logN)。

### **堆積 (Heap)**

堆積 (Heap)，是一種特殊的完全二元樹。

有分兩種，一種是最小堆積，另一種是最大堆積。

#### **最小堆積**

完全二元樹所有的父節點都比子節點要小，就屬於最小堆積。

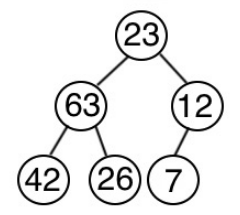
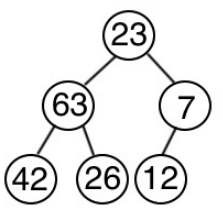
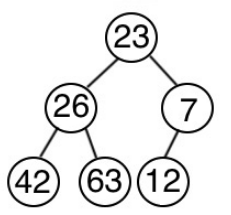
#### **最大堆積**

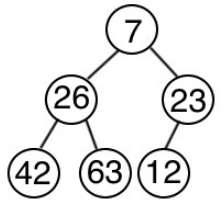
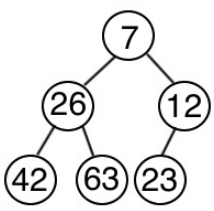
若完全二元樹所有的父節點都比子節點要大，則為最大堆積。

**建立堆積 (**最小堆積**)**

將資料存入到完全二元樹當中，再來一一調整以符合堆積的特性。

從最後一個節點開始，依次判斷以這個節點與父節點是否符合最小堆積的特性，否則調整(互換位置)。一層一層往上判斷調整，直到整棵樹都符合為止。

=>=>=>

=>

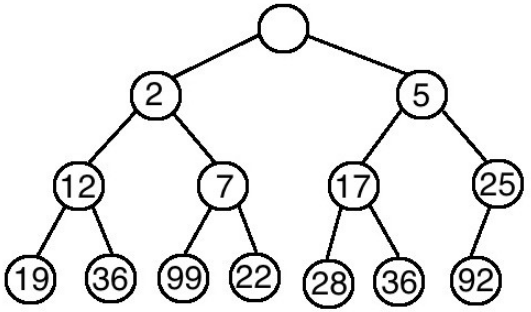
**搜尋指定節點**

只要用迴圈從頭掃一次就可以解決，時間複雜度是 Ｏ(N)

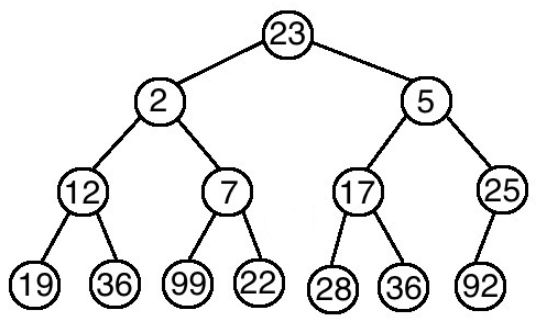
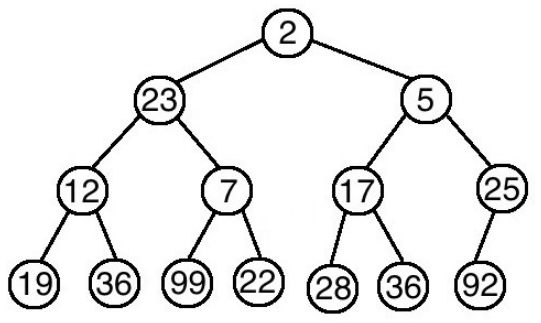
**插入和刪除節點**

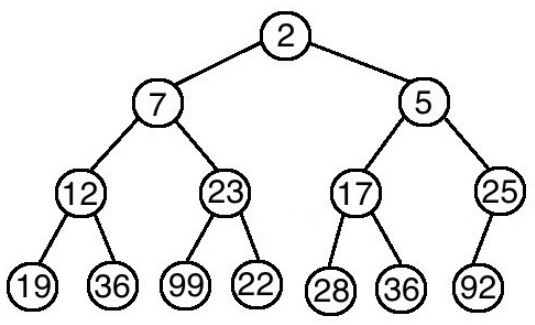
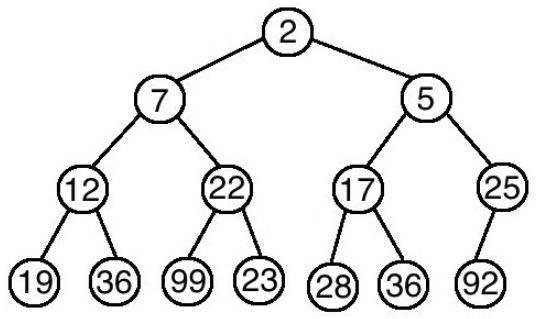
刪除也是只要用迴圈從頭掃一次就可以解決，時間複雜度是 Ｏ(N)。但執行上就不太一樣了。

在最小堆積中，如果刪除的是最小的數，理論上來說就會是在樹的頂端，若堆積的陣列叫做 h 的話，最小樹就是 h[1] 了。然後從右子樹網上遞補。但若父節點大於子節點的話，則要往下持續調整，直到樹符合最小堆積。

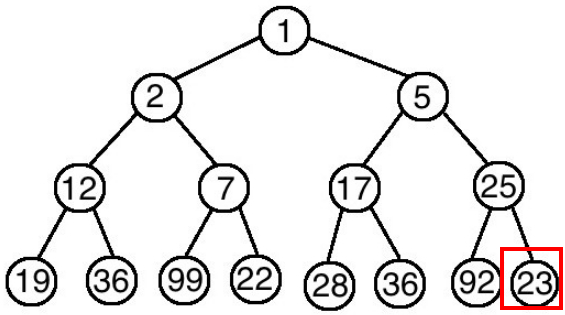
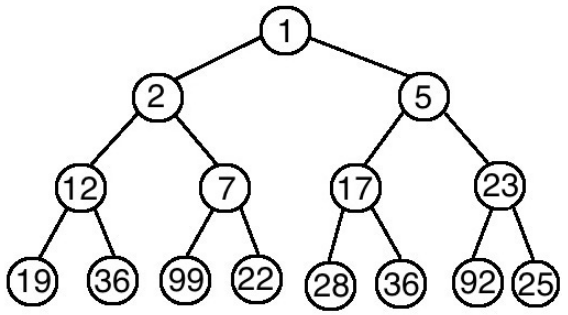


若刪除了最小數，接著要插入一個數(23)，我們可以從頂端開始，向下比較，取左右子樹較小的值與之交換，再一一往下重複施作，直到符合最小堆積為止。

=>=>

=>

直接插入新元素(23)呢？我們可以將它插在樹的末尾，再根據情況判斷新元素是否需要上移，直到滿足堆的特性為止。如果堆的大小為Ｎ，那插入一個新元素所需的時間為 Ｏ(logＮ)。

 => 

**堆積的排序**

堆積也可以用來排序。假設一個陣列 h，有 n 個元素，我們要由小排到大，可以先建好最小堆，然後每次刪除頂端元素並放到一個新的陣列，直到整棵樹圍空，最終輸出新的陣列就是排序好的陣列。

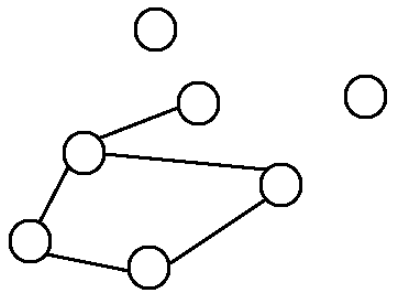
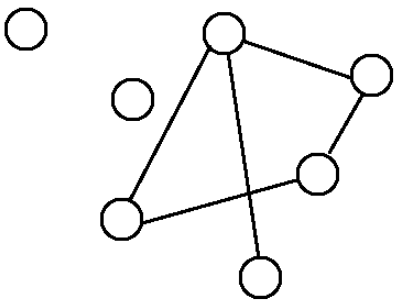
## **圖 (Graph)**

<https://ithelp.ithome.com.tw/articles/10208277>

圖 (Graph)，在資料結構上指的是點和點之間的關聯的東西，並不是數學定義上的兩點成一線，三點成一面的那種圖。一張圖由數條邊(Edge)和數個點(Vertex)所構成，點和點之間可用邊相連，表示這兩個點之間有所關聯。當然，兩個點之間可以有很多個邊，代表很多關係，又或者是一個點可以與很多個點相連接。

### **1. 同構 (Isomorphism / Isomorphic)**

指的是兩張圖的連接方式一模一樣的，而圖上的點可以任意移動，邊不論直的彎的皆可。

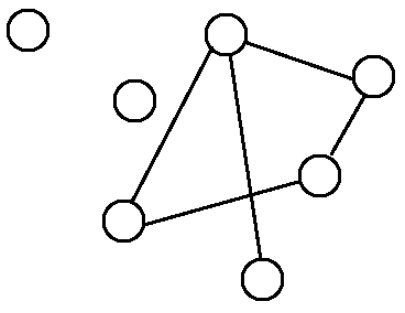
https://ithelp.ithome.com.tw/upload/images/20181106/20111557ehSfTOvFWI.pnghttps://ithelp.ith

### **2. 有向圖 & 無向圖**

有向圖，邊有方向性，表示兩點之間為單向關係。



無向圖，邊無方向性，表示兩點之間為雙向關係。

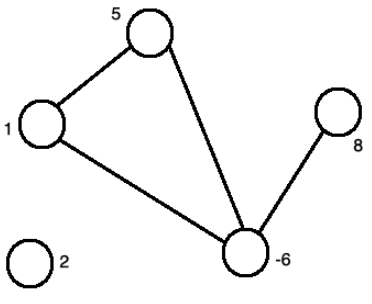


### **3. 權重**

邊加上權重，代表兩點之間的關係

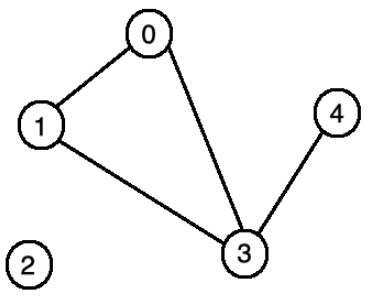
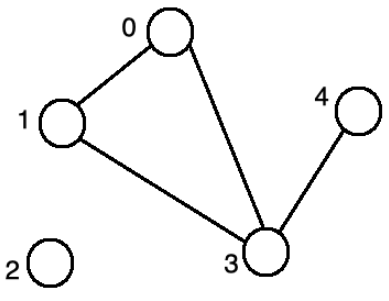


點加上權狀，代表狀態



### **4. 名稱 or 代號**

點可以有名稱或者代號

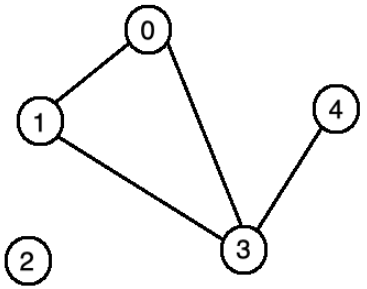
 

### **5. 圖的資料結構**

圖是由點和邊所組成，為了方便人了解，所以視覺化。但實際上在電腦中的儲存方式卻有很多不同的方法。可以有陣列或矩陣的方式儲存。

#### **Eagle List**

#### 將圖用陣列的方式，記錄點與點之間的邊。(0-1相連、0-3相連、1-3相連、3-4相連)



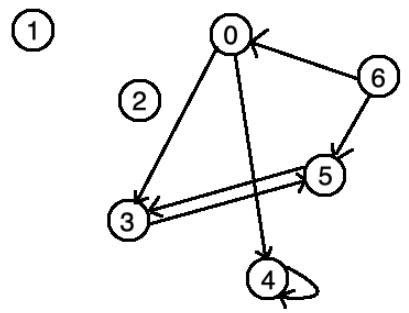
|0|0|1|3|

|-|-|-|-|

|1|3|3|4|

#### **Adjacency Matrix**

把一張圖上的點依序標示編號，然後建立一個矩陣，來記錄連接資訊。方陣中的每一個元素都代表著某兩點的連接資訊。



|　|0|1|2|3|4|5|6|

|-|-|-|-|-|-|-|-

|0|0|0|0|1|1|0|0| // 0 分別有一條連向 3 和 4 的邊，所以第 0 列的 3、4 行為 1。

|1|0|0|0|0|0|0|0|

|2|0|0|0|0|0|0|0|

|3|0|0|0|0|0|1|0| // 0 有一條連向 5 的邊，所以第 3 列的 5 行為 1。

|4|0|0|0|0|1|0|0|

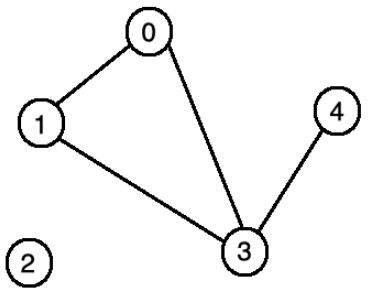
|5|0|0|0|1|0|0|0|

|6|1|0|0|0|0|1|0|

#### **Adjacency Lists**

在每一個點後方，串連所有相鄰有連接的點

**無相圖**



0　[1, 3]

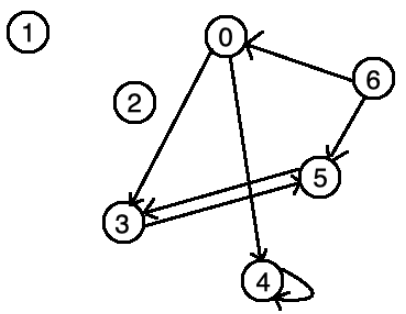
1　[0, 3]

2

3　[0, 1, 4]

4　[3]

**有向圖**



0　[3, 4]

1

2

3　[5]

4　[4]

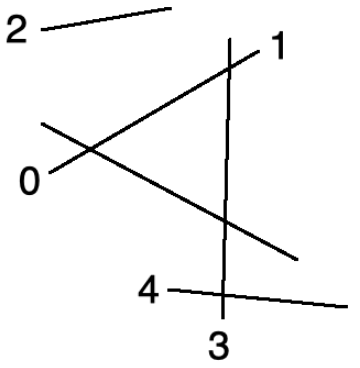
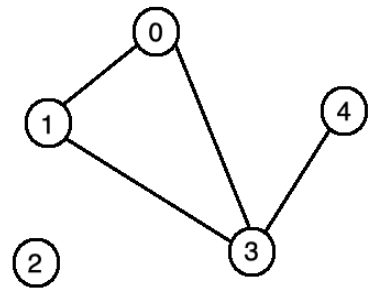
5　[3]

6　[0, 5]

**特殊圖**

* **Intersection Graph**

若兩個物件有交集關係(非空集合)，就可以表示成圖。

 =等於= 

又或者是：

0 = {a, b, c}

1 = {a, d}

2 = {e}

3 = {b, d, f}

4 = {f}

* **Dependency Graph**

是將各個物件所仰賴或依賴的對象關係表示成圖。

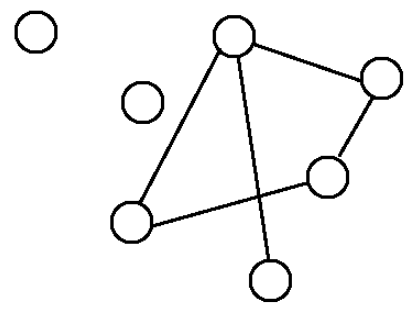
A > C ; E < C

B > A ; C < B

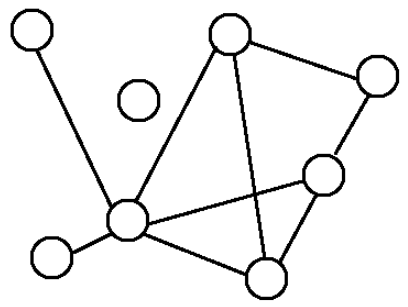
D

* **子圖 Subgraph / 父圖 Supergraph**

一張圖，刪除一些點、一些邊，得到的圖稱作「子圖」。原圖（沒有刪除）、空圖（完全刪除），也算是「子圖」。

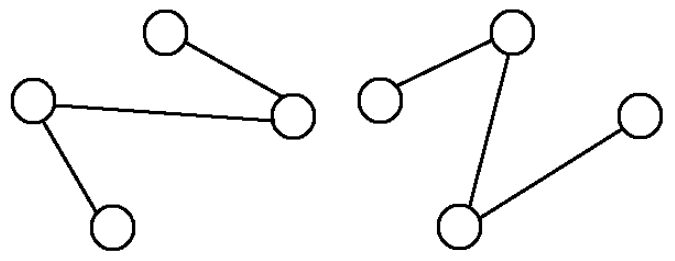


一張圖，增加一些點、一些邊，得到的圖稱作「父圖」。原圖（沒有增加）也算是「父圖」。



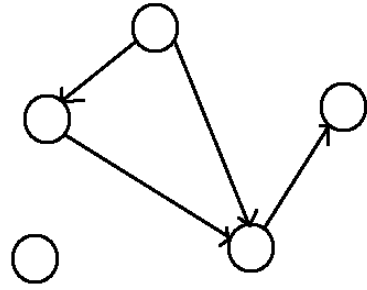
* **補圖 Complement Graph （ Complement ）**

兩點之間，有邊變沒邊，沒邊變有邊。



* **反向圖 Reverse Graph （ Transpose ）**

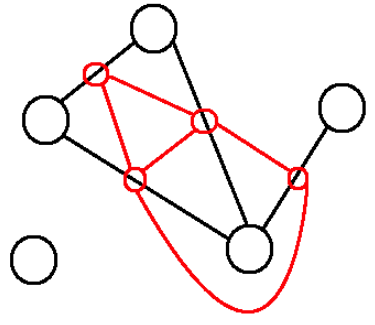
有向圖的邊顛倒過來，方向相反。

 => 

* **Line Graph**

一張圖當中，觀察邊與邊關係，相鄰的邊表示成一張圖，稱作「線圖」。

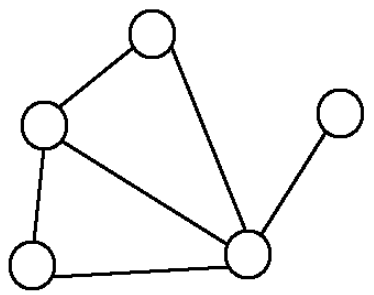
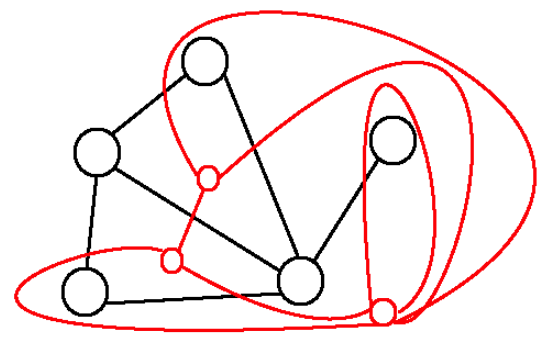
(黑色為原圖，紅色為線圖)



* **Dual Graph**

一張平面圖當中，觀察面與面關係，共邊的面表示成一張圖，稱作「對偶圖」。

(黑色為原圖，紅色為對偶圖)

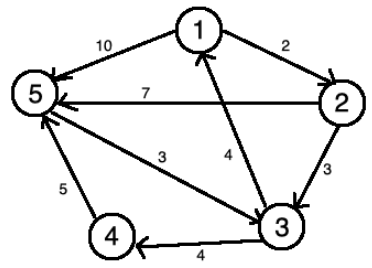
### **圖的遍歷 (Graph Traversal)**

* 圖的深度優先搜尋 Depth-first Search
* 圖的廣度優先搜尋 Breadth-first Search

(之後會另外討論~)

#### **圖的深度優先搜尋 Depth-first Search**

假設我們有 1 到 5 號五個城市，八條公路，且每條公路都是單行的。我們要找出從 1 號城市到 5 號城市的最短路徑。



首先，要先把圖的資料轉為二維陣列 (存在矩陣 e)。

|　|1|2|3|4|5|

|-|-|-|-|-|-|

|1|0|2|∞|∞|10|

|2|∞|0|3|∞|7|

|3|4|∞|0|4|∞|

|4|∞|∞|∞|0|5|

|5|∞|∞|3|∞|0|

每條路的權重代表路的距離，0 代表原地，∞ 代表無法到達。

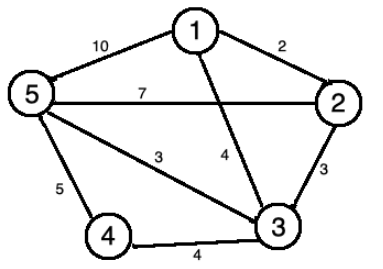
我們從 1 開始走，可以是：

1 → 2 → 3 → 4 → 5　路徑長度為 10

1 → 2 → 5　　　　　 路徑長度為 9

1 → 5　　　　　　　路徑長度為 10

上面的範例是有方向性的，那如果改成無向呢？



|　|1|2|3|4|5|

|-|-|-|-|-|-|

|1|0|2|4|∞|10|

|2|2|0|3|∞|7|

|3|4|3|0|4|3|

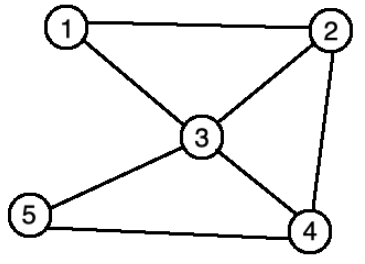
|4|∞|∞|4|0|5|

|5|∞|7|3|5|0|

但其實程式碼只要再多加一行就可以了。



#### [**圖的廣度優先搜尋 Breadth-first Search**](https://ithelp.ithome.com.tw/articles/10208679)



這是一個無向圖，比較接近現實中的地圖。今天我們要從 1 號城市搭飛機到 5 號城市，請問要怎麼搭轉機次數最少呢？

當然，我們用看的就可以知道先從 1 號飛到 3 號，再飛到 5 號，但電腦要怎麼判別？

首先，先把圖的資訊存入二為矩陣(e)。

|　|1|2|3|4|5|

|-|-|-|-|-|-|

|1|0|1|1|∞|∞|

|2|1|0|1|1|∞|

|3|1|1|0|1|1|

|4|∞|1|1|0|1|

|5|∞|∞|1|1|0|

再用兩個佇列來一一擴展

首先可以從 1 號到達 2 號和 3 號城市。

|x|1|2|3|　|　|　|　|

|-|-|-|-|-|-|-|-|

|step|0|1|1|　|　|　|　|

2 號城市又可以到達 3 號和 4 號城市，但 3 號已經在佇列中，所以只需將 4 號加入到佇列。

|x|1|2|3|4|　|　|　|

|-|-|-|-|-|-|-|-|

|step|0|1|1|2|　|　|　|

接著 3 號城市又可以到達 4 號城市和 5 號城市，因為 4 號也已經在佇列中，所以只需將 5 號加入。

|x|1|2|3|4|5|　|　|

|-|-|-|-|-|-|-|-|

|step|0|1|1|2|2|　|　|

找到 5 號城市，演算法就結束啦。可以想一想，為什麼一擴展到 5 號城市就可以結束，而之前的深度優先搜尋卻不行呢？ (答案我不告訴你，哈哈)

## **雜湊 (Hash)**

<https://ithelp.ithome.com.tw/articles/10208884>

很多人以為雜湊就是加密，但雜湊不是加密！ 雜湊是因為他的特性很適合來做加密的運算，但真的不等同於加密！

雜湊 (Hash)

雜湊（英語：Hashing）是電腦科學中一種對資料的處理方法，通過某種特定的函式/演算法（稱為雜湊函式/演算法）將要檢索的項與用來檢索的索引（稱為雜湊，或者雜湊值）關聯起來，生成一種便於搜尋的資料結構（稱為雜湊表）。舊譯哈希（誤以為是人名而採用了音譯）。它也常用作一種資訊安全的實作方法，由一串資料中經過雜湊演算法（Hashing algorithms）計算出來的資料指紋（data fingerprint），經常用來識別檔案與資料是否有被竄改，以保證檔案與資料確實是由原創者所提供。

如今，雜湊演算法也被用來加密存在資料庫中的密碼（password）字串，由於雜湊演算法所計算出來的雜湊值（Hash Value）具有不可逆（無法逆向演算回原本的數值）的性質，因此可有效的保護密碼。

---引自〈維基百科〉

這邊來統整一下。

**雜湊函數 (Hash function)**

主要是將不定長度的輸入，演算成固定長度雜湊值的輸出。且所計算出來的雜湊值必須符合兩個主要條件：

1. 由雜湊值是無法反推出原來的訊息
2. 雜湊值必須隨明文改變而改變。

舉例來說，雜湊函數就像一台果汁機，我們把蘋果香蕉你個芭樂 (資料) 都丟進去打一打、攪一攪，全部變得爛爛的很噁心對吧？！這時候出來的產物 (經過雜湊函數後的值)，是獨一無二的，沒有辦法反向組合成原來的水果 (資料)。倘若我們把蘋果改成紅龍果，出來的產物 (經過雜湊函數後的值) 就會跟著改變，變成桃紅色的，不再是原來的淡黃色。

承上述的例子，用紅龍果香蕉你個芭樂經過雜湊函數出來的顏色是桃紅色 (雜湊值)，那有沒有可能我用其他的水果也可以打出相同的顏色呢？但因為雜湊值的特性是無法反推的，所以如果真的打出相同的顏色的話，我們稱為碰撞 (Collision)。這就代表說這個雜湊值已經不安全，不再是獨一無二的了，需要更改雜湊函數。

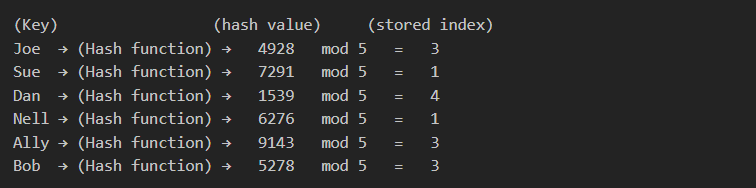
**雜湊表 (Hash table)**

在用雜湊函數運算出來的雜湊值，根據 鍵 (key) 來儲存在數據結構中。而存放這些記錄的數組就稱為 雜湊表。

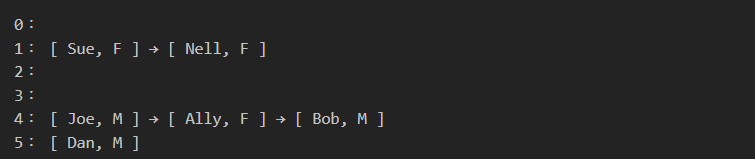
舉例來說，我們有一筆資料用字典的形式表示，每個名字都搭配性別：



而我們將每個名字經過雜湊函數的運算。



hash value 是獨一無二的，用 mod 5 來得到餘數並儲存才記憶體中。



當我們要找資料的時候，例如 Ally 的性別，我們就把 Ally 丟到名為 Hash 的果汁機來得到 hash value，再用 mod 5 找到儲存在記憶體中的位置，但記憶體中的第一個位置並不是 Ally 是 Joe，我們根據 Joe 的鏈結找到下一個元素，直到找到答案。

另一個會讓人疑惑的點是為什麼用 mod 5？其實有很多種方式來選擇怎麼儲存在記憶體中，這邊只是範例。而會用 mod 5 是因為記憶體儲存空間假設為 5 。但儲存空間的大小也不是隨意設的，如果設得太大，資料沒那麼多，會造成空間浪費；若設得太小，會造成每一個空間的資料重疊，查找不易。所以設定空間大小也是一門學問。

**優點 :**

1. 雜湊表其實很適合來存放不確定數量大小的資料
2. 查找的時候也很快速、彈性。