

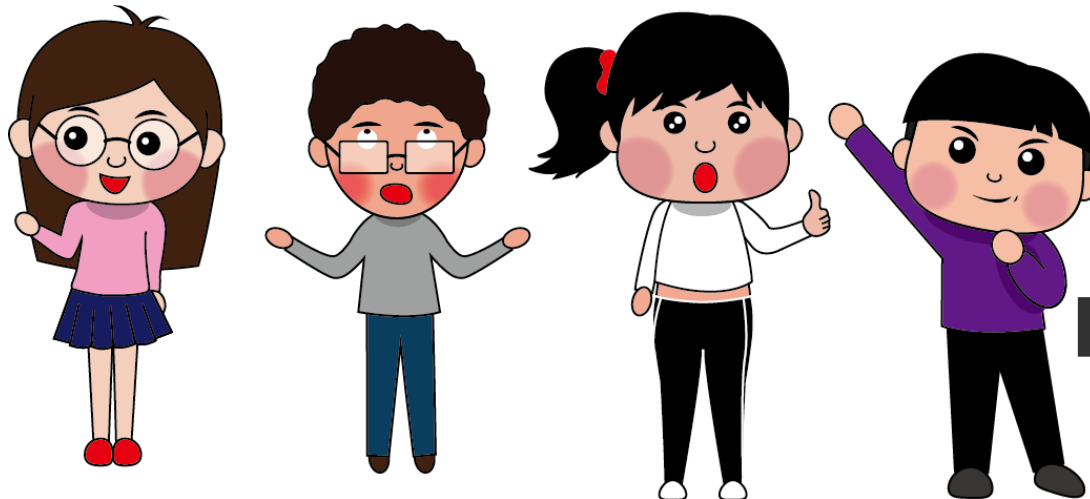
CH03

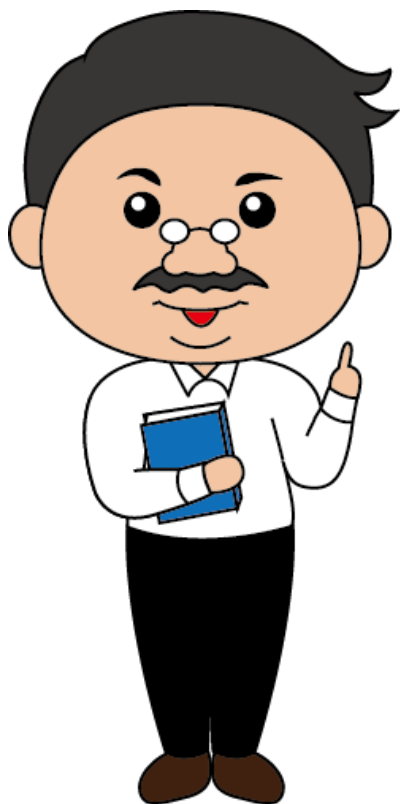
演算法與資料結構

3-1 演算法與演算法的表達方式

3-2 資訊科技常用的演算法

3-3 資料結構





3-1 演算法與演算法的表達 方式

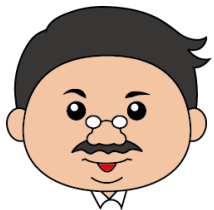
- 3-1-1 文字敘述
- 3-1-2 流程圖
- 3-1-3 虛擬碼



3-1-1 文字敘述

- 利用淺顯易懂的文字來說明解決的步驟。
- 文字敘述除了大概的描述問題之外，通常用於解決問題中每一步驟的說明，可用於流程圖或虛擬碼中的某一部分。





3-1-1 文字敘述

- 例如：以文字敘述來描述百貨公司消費滿額折扣計算的處理流程。

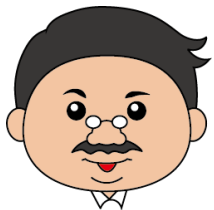
1. 確認消費金額是否等於或大於滿額折扣金額\$1,000。若是，則繼續步驟2。
2. 消費金額打九折。
3. 顯示消費金額。



3-1-2 流程圖

- 流程圖是採用圖形符號來表示解決問題的方法與步驟，流程圖可以清楚的表示程式的執行流程，讓閱讀者了解程式的邏輯架構，對演算法的推演很有幫助。
- 每個流程圖符號各代表一種程式執行功能，藉由組合各種符號而組成一張程式流程圖。





3-1-2 流程圖

- 目前流程圖符號普遍採用美國國家標準協會(ANSI)所公佈的各種圖示符號。
- 流程圖適用於小型且簡單的演算法展示，對於大型系統另外可以採系統架構圖的方式，來呈現整個系統各部分的功能。



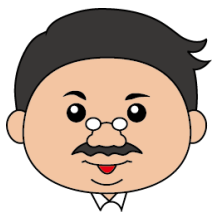


3-1-2 流程圖

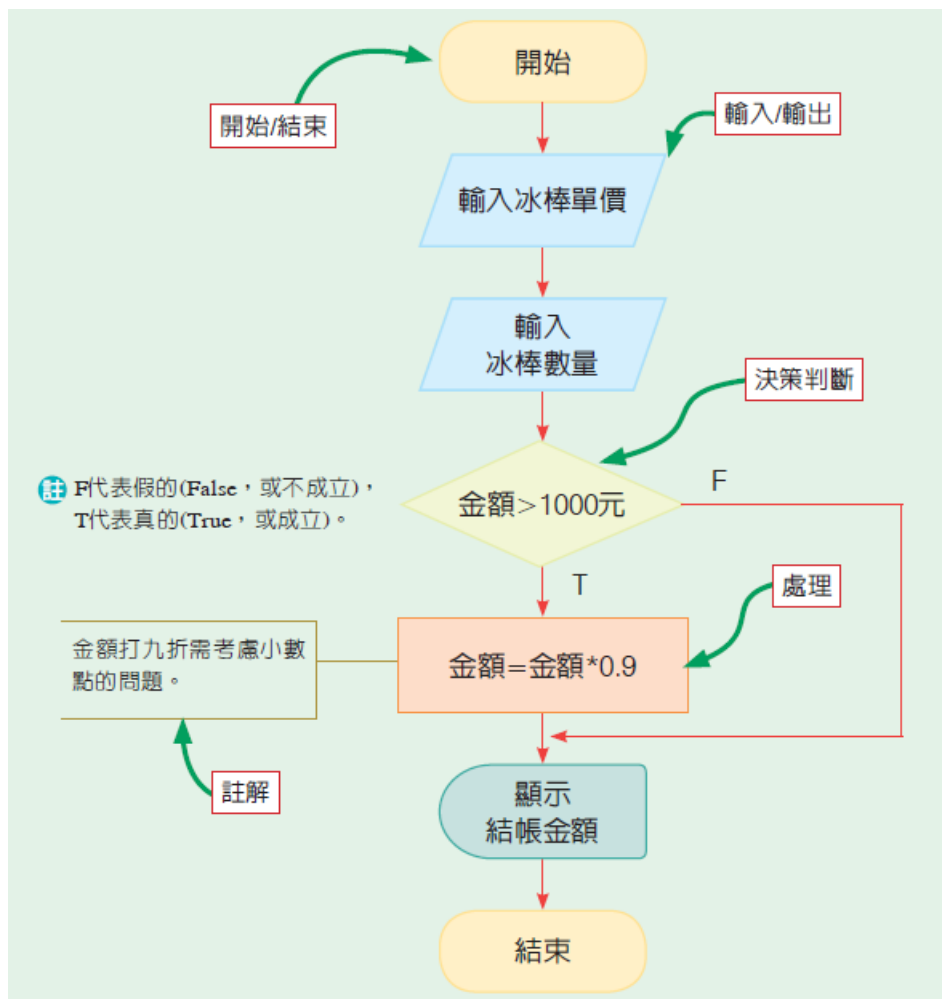
流程圖符號

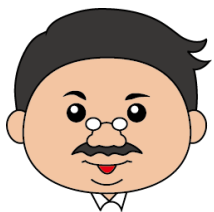
- 流程圖利用各種圖形及箭頭等符號，呈現解決問題的過程與流向。
- 例如：開始及結束用橢圓形；單一執行步驟用長方形；判斷用菱形等等。





3-1-2 流程圖

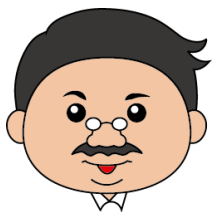





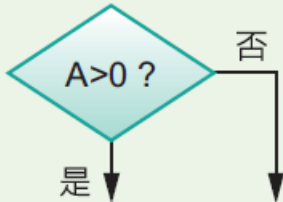
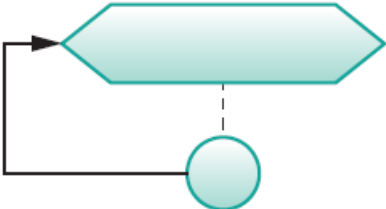
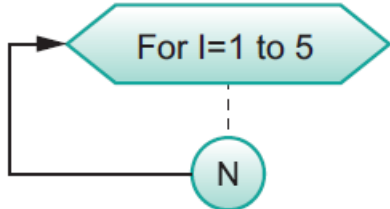




3-1-2 流程圖

- 流程圖可以明確的指出每一步驟的接續步驟或判斷後的流向，對程式開發很有幫助，對程式完成後的維護與修改也很重要。

名稱	符號	意義	範例
開始/結束		表示流程的開始與結束	 
輸入/輸出		表示資料的輸入或輸出	
處理		表示要進行的處理工作	


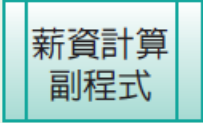


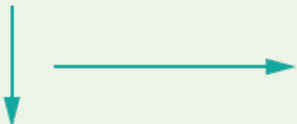

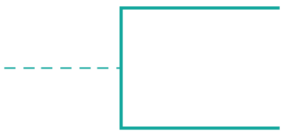
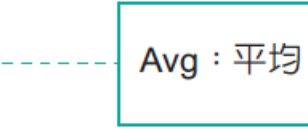


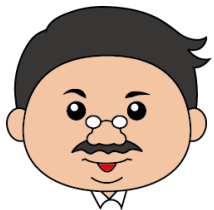
3-1-2 流程圖

決策判斷		根據條件進行判斷選擇	
迴圈		設定迴圈變數的初始值與終端值	
顯示		表示將結果顯示在螢幕上	
報表		表示結果為報表文件或用印表機輸出	



3-1-2 流程圖

副程式		表示一個預先定義好的副程式或函數	
連接		當流程很龐大時，用來連接不同的流程，並且可以避免流程線交叉	
流程線		用來指示流程行進的方向	
註解		可以幫流程加上說明，幫助理解	



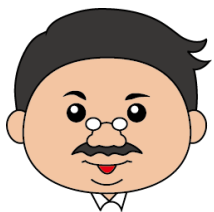
3-1-2 流程圖

■ 實作練習1：「奇偶數判別」的演算法

文字敘述

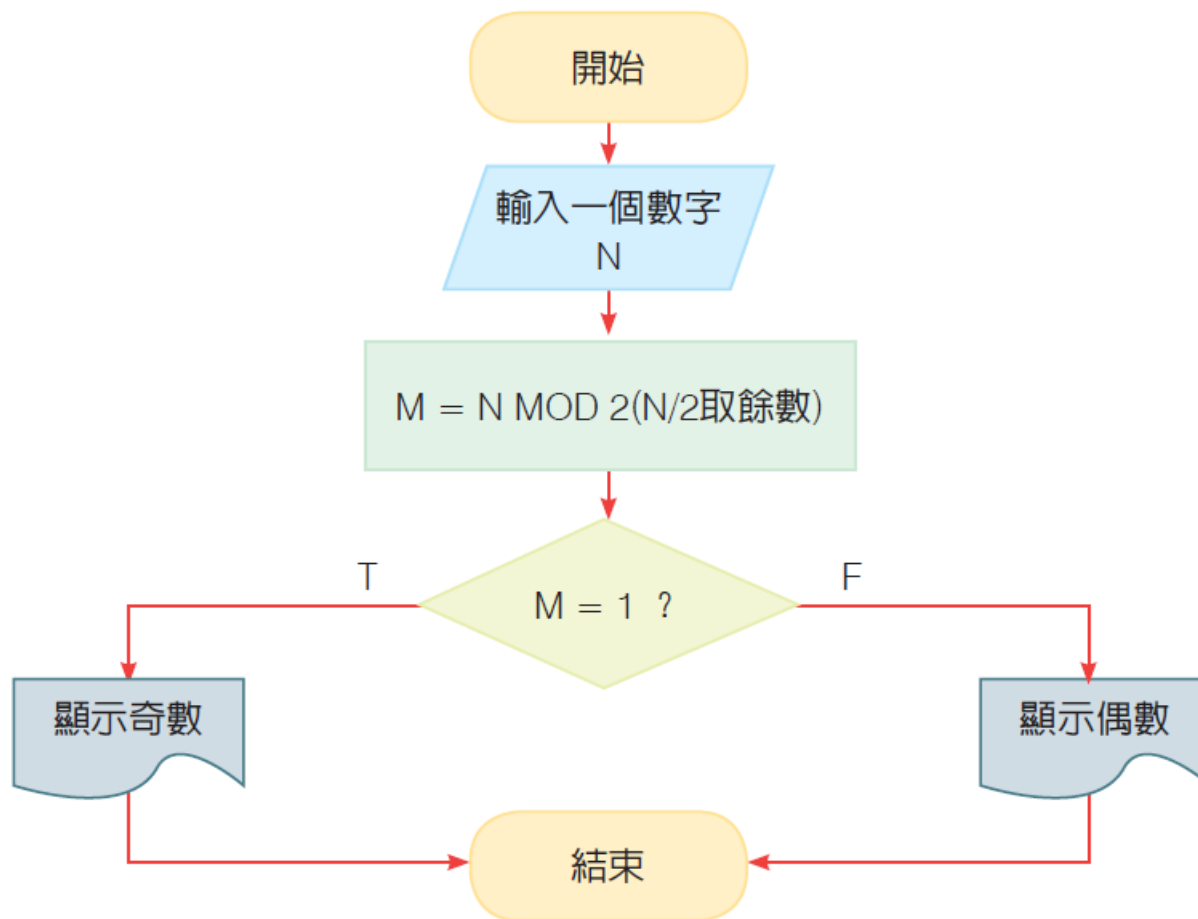
- 步驟一：輸入一個值 N 。
- 步驟二： N 除以2，求餘數 M 。
- 步驟三：假如 $M=1$ ，則顯示奇數，否則為偶數。

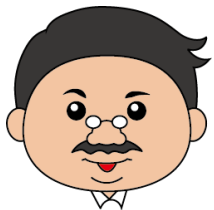




3-1-2 流程圖

■ 流程圖





3-1-2 流程圖

流程圖的使用

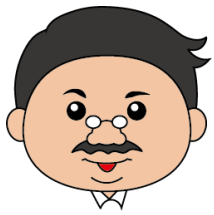
■ 繪製流程圖時最好遵守以下的原則：

- 1. 使用標準的符號，採由上而下，由左至右繪製。
- 2. 符號中標示的文字要清楚明確、簡潔有力。
- 3. 適當採用連接符號，避免產生流程線交叉的情形。
- 4. 如果流程很複雜，可以先分割成幾部分函式，各函式應單獨呈現。



3-1-3 虛擬碼

- 虛擬碼是一種接近程式碼的演算法展示方式，虛擬碼可以採用某一程式語言為主，也可以綜合各程式語言，部分的執行內容，甚至可以採用自然語言來表示。
- 虛擬碼一般是提供給具程式設計經驗者參考用，許多重要資訊科技演算法的發表，都會採用虛擬碼來讓讀者更容易了解演算法的演繹方法。

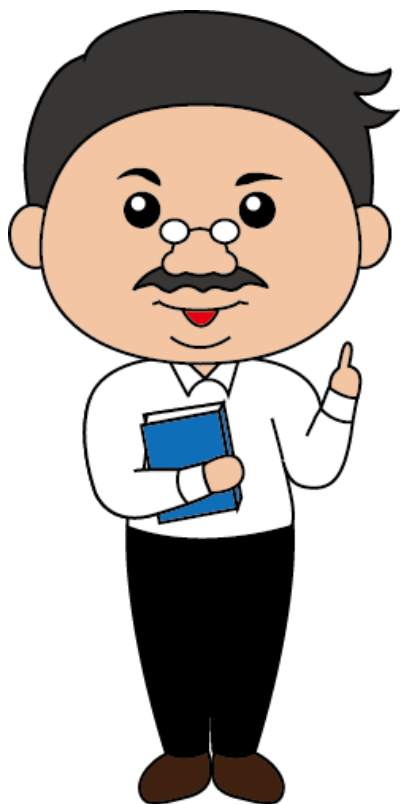


3-1-3 虛擬碼

■ 求k!值的虛擬碼

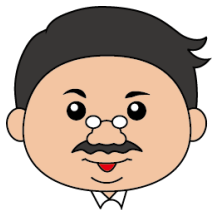
```
Function factorial (integer k)
{
  Set Result as integer = 1
  Set str as string = ""
  Do while (k>1)
  {
    Result = Result * k
    Set k = k-1
    Set str = str & k & "*"
  }
  Set str = str & "1"
  Response (str & "= " & Result)
}
```





3-2 資訊科技常用的演算法

- 3-2-1 排序演算法
- 3-2-2 搜尋演算法
- 3-2-3 演算法的效能表示法



3-2-1 排序演算法

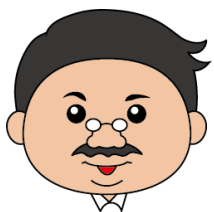
- **排序 (Sorting)** 是常見的功能，例如：成績排名、出場排序，或是透過排序將資料分類等等。
- 排序也是資料搜尋的先前工作，在排序完成的資料中搜尋資料的速度，遠快於在未排序中搜尋的速度。



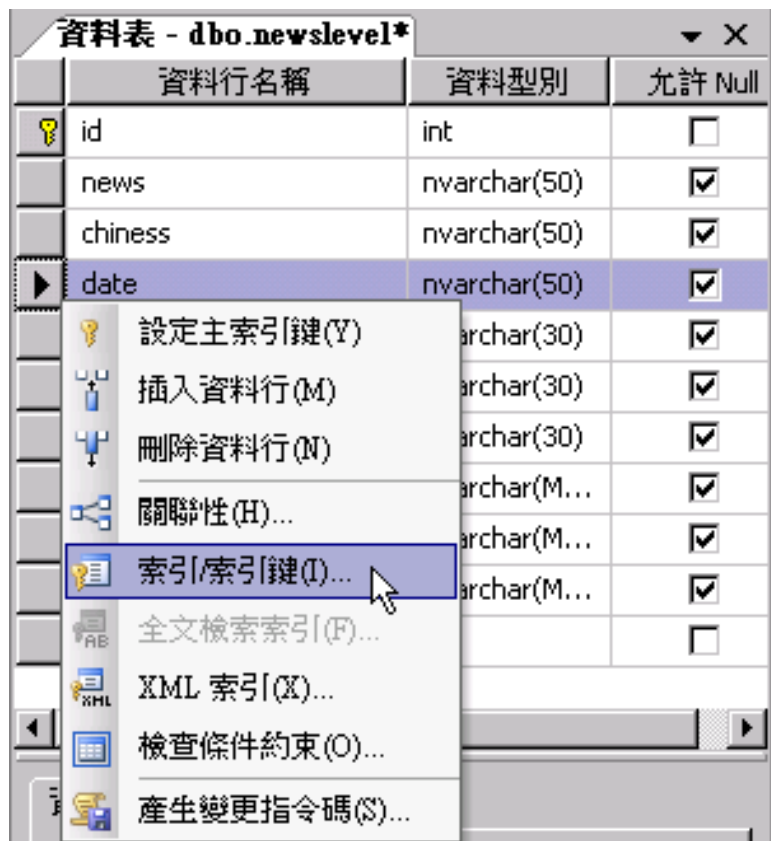


3-2-1 排序演算法

- 譬如說，在資料庫中，資料庫管理者可以設定哪些欄位需要建立索引，資料庫會協助先將索引欄位排序，以加速未來資料庫搜尋該欄位的速度。
- **SQLServer** 資料庫中建立索引的畫面，一個資料表可以建立數個索引欄位，只要有可能被搜尋的欄位，都可以設定為索引欄。



3-2-1 排序演算法



3-1

3-2

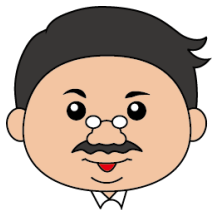
3-3



3-2-1 排序演算法

氣泡排序

- **氣泡排序(Bubble Sort)**的做法是每一回合將一個最大值放到待排序的最上面，就像氣泡一樣的往上升(小至大排序)，如為大至小排則相反。
- **實作練習2：氣泡排序**
- 假設有4筆資料，分別為5、8、1、3。若要將這些資料由小到大排序，則其排序過程如下所示。



3-2-1 排序演算法

影片



3-1

3-2

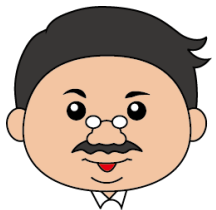
3-3



3-2-1 排序演算法

插入排序

- **插入排序(Insertion Sort)**簡單的說就是將待排數字插入到已排數字中，第一筆視為已排序數字。
- **實作練習3：插入排序**
- 插入排序兩兩數字的比較次數，與氣泡排序相同，都是 $n(n-1)/2$ ，也只要一個暫存空間存放待插入數字。假設有4筆資料，分別為4、8、6、5。
- 若要將這些資料由小到大排序，則其排序過程如下所示。



3-2-1 排序演算法

影片



3-1

3-2

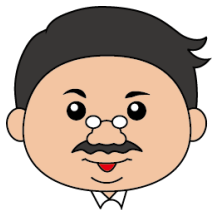
3-3



3-2-1 排序演算法

選擇排序

- **選擇排序 (Selection Sort)** 將資料分成二組，一組是已排序資料，另一組是待排序資料，程式在每一回合會找出待排序中的最小數字，此數字與待排序資料的第一個位置的數字交換位置，最小數字即可排在已排序資料的最後面。
- **實作練習4：選擇排序**
- 假設有6筆資料，若要將這些資料由小到大排序，則其排序過程如下所示。



3-2-1 排序演算法

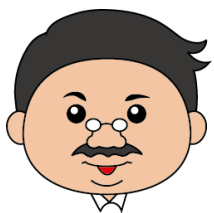
影片



3-1

3-2

3-3



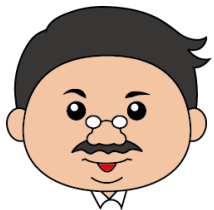
3-2-2 搜尋演算法

循序搜尋

- **循序搜尋**(Sequential Search)又稱為**線性搜尋**(Linear Search)，這個方法很簡單，就是從頭開始一筆一筆的搜尋，直到找到搜尋值(key，鍵值)。
- 搜尋資料一般是放在陣列中，循序搜尋的搜尋資料(Data)無需先行排序。

index	0	1	2	3	4	5	6	7	8	9
data	8	4	1	3	9	5	6	7	2	0





3-2-2 搜尋演算法

- 在循序搜尋時需設定一個搜尋索引(index)，代表目前搜尋的位置，如果鍵值不一定在data中，則演算法每一次迴圈除了必須判斷是否找到資料之外，也必須判斷index是否到達陣列資料的最後一個元素，這會降低搜尋的效率。

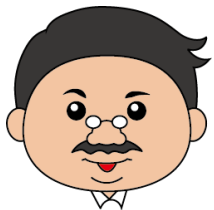




3-2-2 搜尋演算法

- 解決這個問題的方法，是在data最後面加入一個**鍵值 (Key)**，這個值稱為**崗哨 (Sentinel)**，這可以保證在每一次迴圈減少一次程式判斷，但是演算法最後必須判斷找到的是崗哨或者真的是搜尋的key。

index	0	1	2	3	4	5	6	7	8	9	10
data	8	4	1	3	9	5	6(Key)	7	2	0	6(崗哨)



3-2-2

搜尋演算法

影片



3-1

3-2

3-3



3-2-2 搜尋演算法

- 假設尋找的資料有 n 筆，循序搜尋最差的情況是找 n 次才找完(崗哨法為 $n+1$ 次)，最好的情況是1次就找到，平均的搜尋次數是 $(n+1)/2$ 。
- 前面提到的崗哨法找尋次數並不會變，但每回合迴圈中只需1次判斷，無崗哨法則要兩次。



3-2-2 搜尋演算法

- 下圖是無崗哨法與崗哨法程式的比較，網底是迴圈的部分，粗體字是判斷式，無崗哨法比崗哨法多1次判斷(**`index < data.length`**)，判斷索引值是否已超過陣列的長度。



3-1

3-2

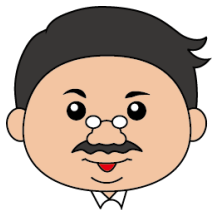
3-3



3-2-2 搜尋演算法

無崗哨法	崗哨法
<pre>Function LinearSearch(data, key){ var index = 0; while(index < data.length) //判斷1 { if(data[index] == key) //判斷2 return index; else index = index + 1; } return -1; //註解 -1代表搜尋不到 }</pre>	<pre>Function sentinel_LinearSearch (data, key){ var index = 0; data.push(key); // 將崗哨加到陣列尾端 while(data[index] != key) //判斷1 { index = index + 1 } data.pop(key); // 移除崗哨 if(index < data.length) return index; else return -1; //註解 -1代表搜尋不到 }</pre>





3-2-2 搜尋演算法

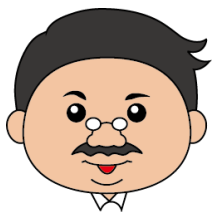
二元搜尋

- **二元搜尋法(Binary Search)**的搜尋速度比循序搜尋快，但前提是必須將搜尋資料先排序。
- 如果搜尋的次數只有一次，採用二元搜尋並沒有比循序搜尋快。



3-2-2 搜尋演算法

- 二元搜尋的原理是根據鍵值，每回合(迴圈)會找出一半符合的資料， n 筆資料最多只要 $\log_2 n$ 次即可找出鍵值的位置，假如 $n=10$ ，則最多4次即可搜尋完。
- 二元搜尋每一回合會定一個搜尋資料的中間索引(middle)位置，及左邊界(left)與右邊界(right)。



3-2-2 搜尋演算法

- 實作練習5：二元搜尋
- 以下圖為例， $left=0$ ， $right=9$ ， $middle = (left+right)/2$ 再取整數 $= \text{int}((0+9)/2) = \text{int}(4.5) = 4$ ， $\text{int}(\text{函數})$ 是指取整數。

	left				middle					right
index	0	1	2	3	4	5	6	7	8	9
data	18	27	39	45	69 middle-value	73	81(key)	86	93	99



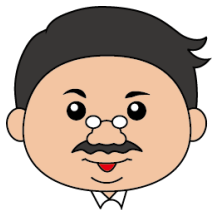
3-2-2 搜尋演算法

二元搜尋的演算法

- 初始值 $left=0$; $right = data.length-1$ (資料長度，從0開始所以減1)， key =指定的搜尋值。

1. 算出 $middle = \text{int}((left + right) / 2)$ 。
2. 取出中間值($middle\text{-}value$) = $data[middle]$ 。
3. 假如 $key = middle\text{-}value$ ， $middle$ 就是 key 在 $data$ 中的位置，搜尋結束。
4. 假如 $key > middle\text{-}value$ ，則 $left = middle+1$ ，程式進入下一回合。
5. 假如 $key < middle\text{-}value$ ，則 $right = middle-1$ 程式進入下一回合。





3-2-2 搜尋演算法

影片



3-1

3-2

3-3



3-2-3 演算法的效能表示法

- 演算法的效能一般以迴圈的最多執行次數來比較，例如：循序搜尋的資料共有 n 筆資料，最差的情況下必須找到最後一筆時才找完，需執行 n 次的迴圈，而二元搜尋最差情況下的執行迴圈數為 $\log_2 n$ 。



3-2-3 演算法的效能表示法

- 在計算機科學中，演算法的效能表示稱為**時間複雜度**(Time Complexity)，一般是以**Big-O**符號表示。
- 然而**Big-O**並非真的算出迴圈的執行次數，以氣泡排序及插入排序為例。

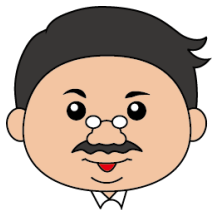




3-2-3 演算法的效能表示法

- 兩者的平均迴圈執行次數都是 $n(n-1)/2 = (n^2-n)/2$ ， n 是待排序資料數，然而時間複雜度僅為 $O(n^2)$ ，省略了減 n 及除 2 的計算，這是因為當次排序的資料量極大時，時間複雜度主要由 n^2 所決定了。





3-2-3 演算法的效能表示法

- 舉例來說，當 $n=1000$ 時， $n^2=1000,000$ (一百萬)，而 $2n=2000$ ，兩者相差極大， $2n$ 省略亦不影響對執行效率的表示。
- 因此Big-O所表達的只是一個概念性的執行次數，而非準確的執行次數。
- 即使是兩個演算法的時間複雜度相等，也不代表兩個演算法的執行時間一定相同。

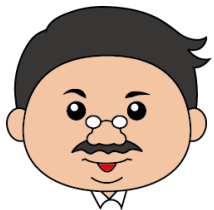




3-2-3 演算法的效能表示法

- 以氣泡排序與插入排序為例，兩者的時間複雜度都是 $O(n^2)$ ，但兩者的執行效能也並非一定相同，這是因為兩者雖然執行迴圈的次數大概相同，但因演算法不同，迴圈內的執行指令也不一定相同，兩者執行的效能就可能不同。





3-2-3 演算法的效能表示法

- **Big-O表示法** $O(1)$ 代表的是常數，是指無論資料大小，透過一次的執行即可獲得答案。



3-1

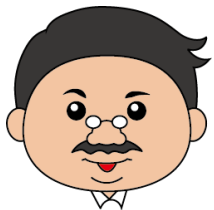
3-2

3-3



3-2-3 演算法的效能表示法

- 演算法的效能一般以迴圈的最多執行次數來比較，例如：循序搜尋的資料共有 n 筆資料，最差的情況下必須找到最後一筆時才找完，需執行 n 次的迴圈，而二元搜尋最差情況下的執行迴圈數為 $\log_2 n$ 。
- 在計算機科學中，演算法的效能表示稱為**時間複雜度**(Time Complexity)，一般是以Big-O符號表示。



3-2-3 演算法的效能表示法

- 然而Big-O並非真的算出迴圈的執行次數，以氣泡排序及插入排序為例，兩者的平均迴圈執行次數都是：

$$n(n-1)/2 = (n^2-n)/2$$

- n 是待排序資料數，然而時間複雜度僅為 $O(n^2)$ ，省略了減 n 及除2的計算，這是因為當次排序的資料量極大時，時間複雜度主要由 n^2 所決定了。



3-2-3 演算法的效能表示法

- 循序搜尋的時間複雜度為 $O(n)$ ，代表搜尋執行時間隨資料大小呈線性增加，二元搜尋的時間複雜度為 $O(\log_2 n)$ ，稱為**次線性時間**(Sub-linear Time)，優於循序搜尋。





3-2-3 演算法的效能表示法

- $O(n^2)$ 代表執行時間是隨資料大小的平方而增加，氣泡排序及插入排序的時間複雜度即是，代表兩者在資料量大時，效率會極差。



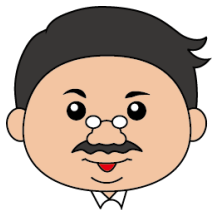


3-2-3 演算法的效能表示法

■ 常見演算法的效能表示（ n 代表資料量）

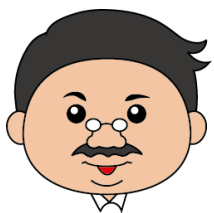
演算法	最差執行次數	Big-O
氣泡排序（Bubble Sort）	$n(n-1)/2$	$O(n^2)$
插入排序（Insertion Sort）	$n(n-1)/2$	$O(n^2)$
選擇排序（Selection Sort）	$n(n-1)/2$	$O(n^2)$
循序搜尋（Sequential Search）	n	$O(n)$
二元搜尋（Binary Search）	$\log_2 n$	$O(\log_2 n)$





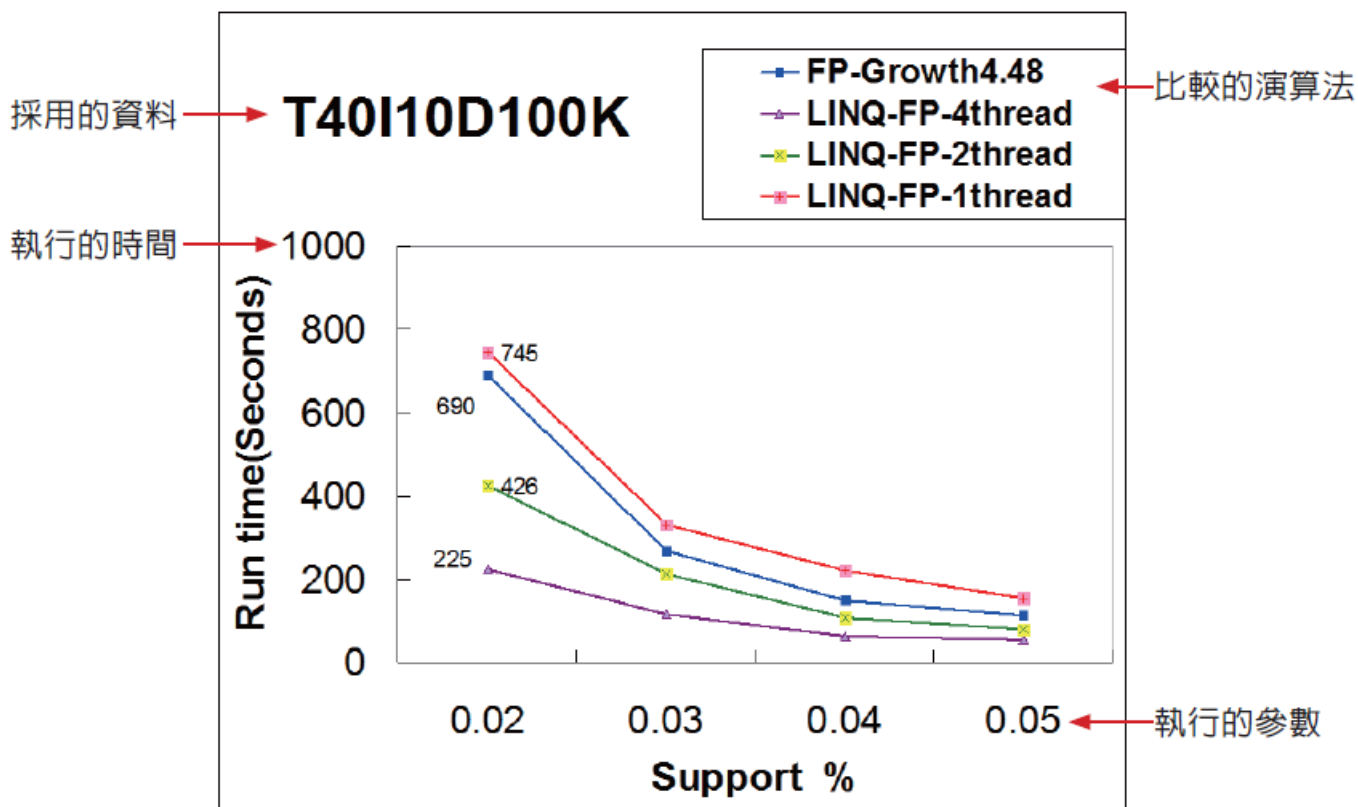
3-2-3 演算法的效能表示法

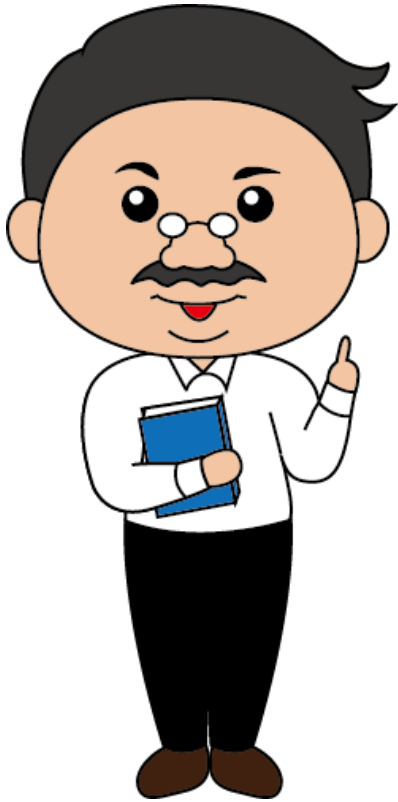
- 實務上還有一種執行效能的比較方法，就是在相同的機器上，採用相同的資料，在條件盡可能相同下，透過不同演算法真實的執行時間來比較。
- 下圖是一種**資料挖掘(Data mining)**演算法的執行效能比較，採用的data是T40I10D100K，在執行參數Support=0.02%下，四種演算法的執行時間秒數各為745、690、426及225。



3-2-3 演算法的效能表示法

- 透過這樣實際的效能比較，可以更容易的呈現各演算法的效能。





3-3 資料結構

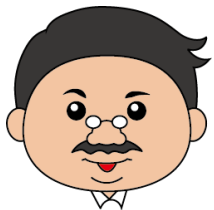
- 3-3-1 陣列
- 3-3-2 鏈結串列
- 3-3-3 堆疊
- 3-3-4 序列



3-3 資料結構

- 電腦的資料關機時是存放於輔助記憶體(一般為硬碟)，執行時為了效率，會將程式及資料從磁碟中複製到主記憶(RAM)中，所謂的**資料結構(Data Structure)**，就是資料存放在主記憶體中的範圍及排列方式，以及資料的存取方法。





3-3

資料結構

結構類型	記憶體排列方式	存取方法
陣列 (array)	連續	循序、隨機 (指定索引即可存取該元素)
堆疊 (stack)	連續或不連續	先進後出 (只有一個頂端入口及出口，先堆入的後取出，後堆入的先取出， First In Last Out, FILO) 。
序列 (queue)	連續或不連續	先進先出 (從一入口進，依序從另一出口出， First In First Out, FIFO) 。
串列 (list)	連續或不連續	循序 (每一儲存空間存有下一儲存空間的位址，只能循序至每一儲存空間)



3-1

3-2

3-3



3-3 資料結構

- 資料結構的範圍(大小)必須視結構類型及定義的資料型態而定。
- 例如：宣告一個長度為5個元素的整數(2bytes)陣列，程式電腦系統在執行時會保留一塊10bytes($5 \times 2 = 10$)的連續記憶體空間，如果是宣告一個長度為5的長整數陣列，則是20bytes(位元組)的連續記憶體空間。



3-3 資料結構

- 程式設計師必須在程式中依據變數的可能範圍，宣告一個適當的資料型態。
- 例如：一個計算薪資的程式，薪資變數可能用長整數(long)比較適合，如果用整數(int)，則當某人的薪資超過32767元，程式就會出現錯誤。





3-3 資料結構

■ 資料型態與數值範圍

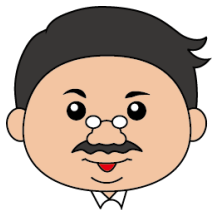
資料型態	容量(位元組byte)	數值範圍
字元char	1	-128 ~ 127
整數int (短整數short)	2	-32768 ~ 32767
長整數long	4	-2147483648 ~ 2147483647
浮點數float	4	3.4E-38 ~ 3.4E+38
雙精度浮點數double	8	1.7E-308 ~ 1.7E+308



3-1

3-2

3-3



3-3-1 陣列

- 在程式運作中，需要使用**變數**(Variable)來儲存資料，例如：學生的學期成績，但是如果存放**10**位學的成績，此時就可以宣告一個陣列變數，同時存放**10**個人的成績，而無需宣告十個變數。

```
int a[10]
```

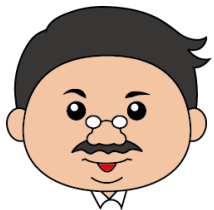
索引	0	1	2	3	4	5	6	7	8	9
Score	83	56	85	78	57	90	98	45	66	74



3-3-1 陣列

- 陣列是一群相同資料型態的連續記憶體(部分程式語言不保證為連續記憶體)，透過索引(Index)可循序或隨機指定存取其中的一筆資料(陣列中的一個元素)，陣列的索引(資料在陣列中的位置)一般都是從0開始。





3-3-1 陣列

- 陣列在宣告時，必須同時宣告資料型態，C語言的陣列宣告語法如下：

```
char Uname[60]; //宣告一個含60個元素的字元陣列
```

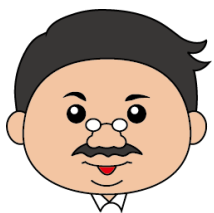




3-3-1 陣列

- 部分程式語言可以不用宣告資料型態，由程式解譯器在執行時依資料內容自動配置所需的記憶體。
- 此外宣告時如果有兩個標號，則稱為二維陣列。





3-3-1 陣列

- 如下圖為包含學生平均成績及學號的二維陣列，每個陣列中的儲存空間 (Element，元素) 必須含行及列二個索引。

索引	0	1	2	3	4	5	6	7	8	9
0	101 [0][0]	102 [0][1]	103 [0][2]	104 [0][3]	105 [0][4]	106 [0][5]	107 [0][6]	108 [0][7]	109 [0][8]	110 [0][9]
1	83.1 [1][0]	56.0 [1][1]	85.7 [1][2]	78.9 [1][3]	57.3 [1][4]	90.1 [1][5]	98.8 [1][6]	45.9 [1][7]	66.3 [1][8]	74.5 [1][9]



3-3-1 陣列

- 陣列是相當常用的資料結構，它的結構簡單，操作方便，也常是其他資料結構的實體結構。
- 例如：陣列可當堆疊的實作結構，也就是說堆疊的結構其實也是採用陣列來完成。





3-3-1 陣列

優點

- 結構簡單且穩定
- 可隨機存取
- 循序存取速度快

缺點

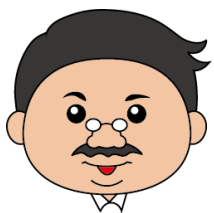
- 空間固定，可能浪費部份空間
- 改變儲存空間的大小不容易，改變時需重新配置空間
- 各空間的資料型態都相同（JavaScript可以不同）





3-3-2 鏈結串列

- 串列是指一個串接的儲存空間，儲存空間連續者稱為**有序串列**(Ordered List)，**陣列**(Array)即是一種有序串列。
- 儲存空間不連續，必須採用指標指引下一個空間者，稱為**無序串列**(Unordered List)，例如：**鏈結串列**(Linked List)。



3-3-2 鏈結串列

單向鏈結

- 鏈結串列中的每個儲存空間稱為節點(Node)，每個節點除了儲存資料值(Value)之外，會有一個指標(Point)，指向下一個節點的位址(Address)，最後一個節點(Tail Node)的指標則設定為Null(空)。



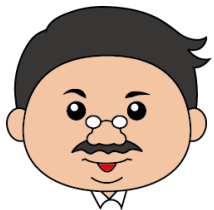


3-3-2 鏈結串列

雙向鏈結

- 雙向鏈結的每個節點有兩個指標，一個指向下一個節點(Next Node Point)的位址，另一個指回上一個節點(Forward Node Point)的位址，首節點(Head Node)的前節點指標必須設定為Null。





3-3-2 鏈結串列

刪除節點

- 以刪除節點為例，當我們要刪除下圖中92這個值，只需要更改前一節點(值=88)的指標位址為下下一個節點的位址(3300)即可。

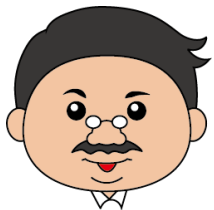




3-3-2 鏈結串列

插入節點

- 插入節點的程式需先替此待插入節點建立一個實體的節點，才能產生記憶體空間並取得此記憶體位址，然後寫入值至節點中，並將指標指向下一個節點，最後更改前節點的指標位置(指向待插入節點)即可。



3-3-2 鏈結串列

鏈結串列的優缺點

- 我們假設有一種情況，想想該如何設計資料結構。情況是一個過關比賽，在比賽前才知道比賽人數，每一關每個人會有一個得分，超過60分者過關，程式必須記錄每人的分數，過關者才能再進入下一關比賽，直到無人過關，最後一關比賽分數最高者為冠軍。



3-3-2 鏈結串列

優點	缺點
<ul style="list-style-type: none">1. 可以動態刪除或插入儲存空間（節點，Node）。2. 可準確配置記憶體數量。3. 各節點資料型態可以不同。	<ul style="list-style-type: none">1. 僅能循序存取。2. 存取速度慢（需要先讀取指標）。3. 指標配置操作不易。



3-3-3 堆疊

- **堆疊(Stack)**的資料結構一般採用**有序串列(Ordered List)**來實作，也可以採用**鏈結串列(Linked List)**來實作。

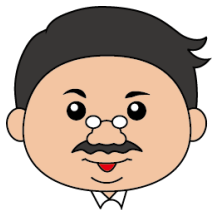




3-3-3 堆疊

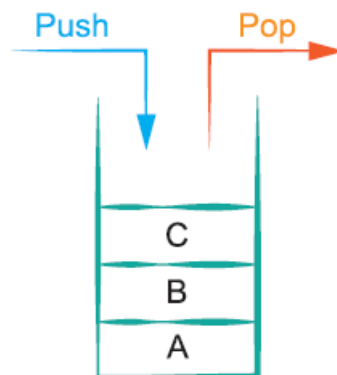
- 然而它最大的特點在於資料的存取只能**先進後出**(First In Last Out, FILO)，或稱為後進先出，資料只能從堆疊頂端(top)進出，先堆入的後取出；後堆入的先取出，這動作可以想像成堆碟子，碟子一個接一個往上疊在頂端，取出時也從頂端一個接一個取出，先堆入的會被壓在下方，取出時必須等頂端的資料一個接一個取出，而不可從整疊碟子的中間抽出某碟子，以免整疊的碟子都打翻了。





3-3-3 堆疊

- 堆疊往上疊的動作稱為**Push**，取出資料的動作稱為**Pop**，如下圖所示，堆疊是一種邏輯資料結構。

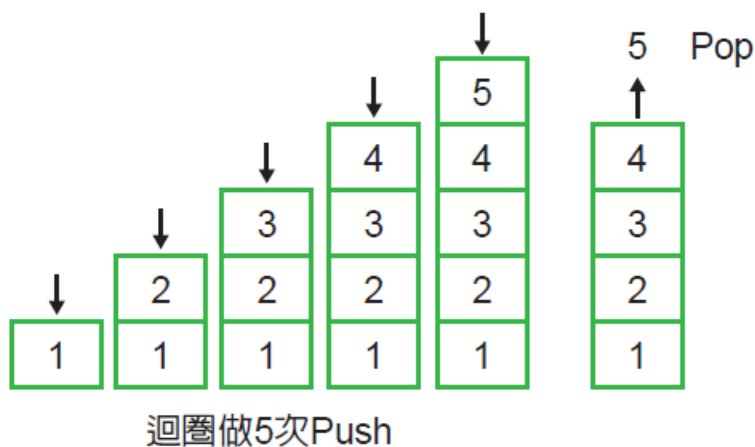


- 並非全部的程式語言皆有堆疊的資料型態，在許多程式語言中，堆疊實作採用陣列來進行，因此在宣告時也是採用陣列的形式來宣告。

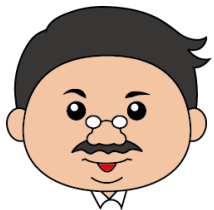


3-3-3 堆疊

- 下圖是一段利用迴圈將1~5五個數字push到Astack這個堆疊中，迴圈結束後pop(彈出)Astack堆疊最頂端的值，此例是數字5。



```
var Astack = new Array();  
for(i=1;i<=5;i++)  
{  
    Astack.push(i);  
}  
var A = Astack.pop();
```



3-3-3 堆疊

堆疊的應用

- 在循序搜尋程式中，我們將崗哨值推入陣列的最頂端，搜尋結束後再將陣列最頂端的崗哨值彈出陣列，這就是堆疊的應用。
- 此外，堆疊在系統程式及程式的編譯或解譯中，也有許多應用。



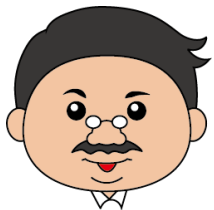


3-3-3 堆疊

1. 副程式的呼叫與返回

- 在程式執行中，如果有呼叫函式(**Function**，或稱副程式**Subroutine**)，而函式又可能呼叫其他函式，則每一層副程式或函式的位址，會被一層一層的存入堆疊中，待程式返回時，即可繼續回到先前程式的位址繼續執行。





3-3-3 堆疊

2. 運算式的轉換

- 電腦在進行運算時，例如： $3*100+50/6$ 這個運算式，電腦一次只能進行兩個數字的計算，這時會先將 $3*100$ 的計算結果(300)及加號先推入堆疊中，待後續的 $50/6$ 運算完成時，再取回加號及300，最後完成運算。





3-3-3 堆疊

3. CPU中斷的處理

- **CPU**遇到特殊的情況時，會中斷目前的程式執行，先執行緊急的事項，此時**CPU**暫存器中的資料會先推入堆疊中，待中斷完成後再彈出，返回的程式即可取回資料繼續執行。





3-3-4 序列

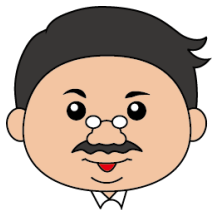
- **序列**(Queue)也稱為佇列，可應用陣列及鏈結串列實現的一種邏輯資料結構。
- 序列的最主要特點在於**先進先出**(First In First Out, FIFO)的資料讀寫方式。
- 印表工作的排程，CPU執行程序的排程，或是鍵盤輸入緩衝處理等等，這些工作的處理特徵都是先進入序列者先處理，也就是排隊等待的概念。



3-3-4 序列

- 序列有頭端(Head或Front)及尾端(Tail或Rear)，資料從尾端寫入(一般術語為：enqueue，排隊)，從頭端讀出(一般術語為：dequeue，出隊)。





3-3-4 序列

線性序列

- **線性序列**(Linear Queue)是指利用陣列來實現序列先進先出的資料結構，因此需先宣告一個固定大小的陣列，並需要宣告頭端(Head)及尾端(Tail)兩個變數，此兩變數初始值皆為-1，兩變數相等時，代表queue為空的。





3-3-4 序列

```
var queue = Array(5); //宣告陣列  
var head = -1;  
var tail = -1;
```

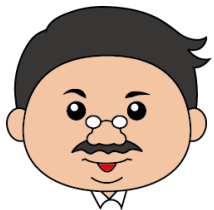
Index	0	1	2	3	4
head					
tail					



3-1

3-2

3-3



3-3-4 序列

- 從尾端寫入三筆資料，執行完成後，**tail = -1+3 = 2**

```
queue.enqueue("A") //tail = -1+1 = 0  
queue.enqueue("B") //tail = 0+1 = 1  
queue.enqueue("C") //tail = 1+1 = 2
```

Index	0	1	2	3	4
head	A	B	C		
tail					

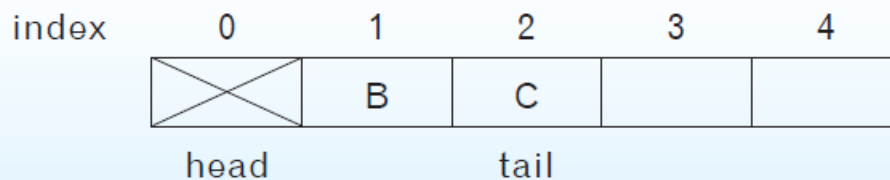




3-3-4 序列

- 從頭端移出一筆資料(A)，執行完成後，
 $\text{head} = -1 + 1 = 0$

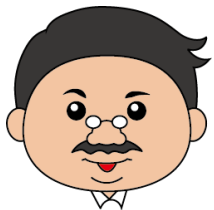
`queue.dequeue()`



3-1

3-2

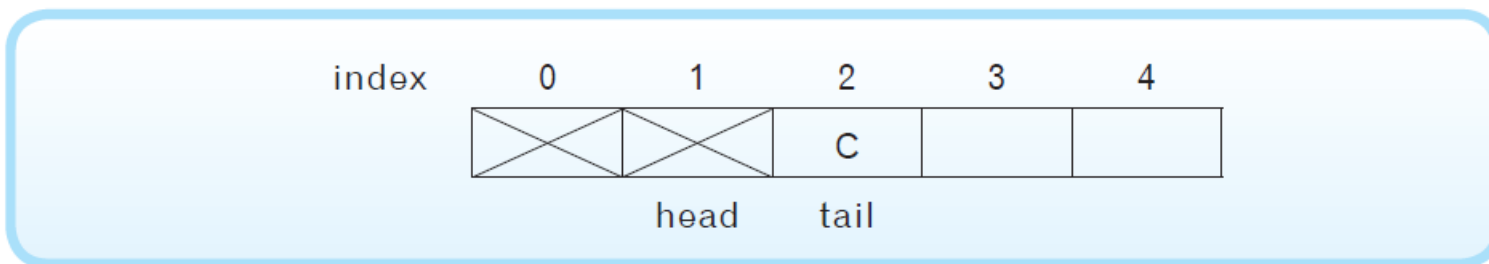
3-3



3-3-4 序列

- 從頭端移出一筆資料(B)，執行完成後，
 $\text{head} = 0 + 1 = 1$

`queue.dequeue()`

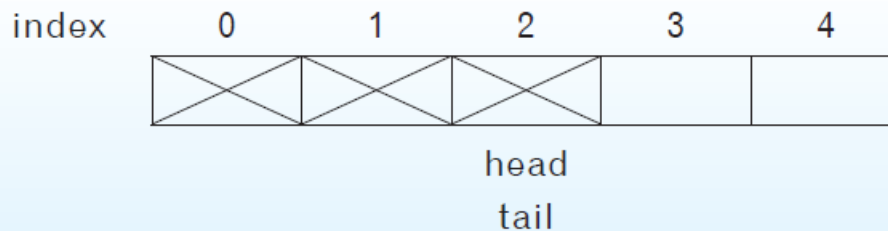




3-3-4 序列

- 從頭端移出一筆資料(C)，執行完成後，
 $\text{head}=1+1=2$

`queue.dequeue()`



3-1

3-2

3-3



3-3-4 序列

環狀序列

- 線性序列有一個很大的缺點，即移除後的儲存空間因head索引的移動而無法再使用，而形成記憶體浪費。
- 解決這個問題的方法，是tail到達陣列長度(索引最大值)時，tail可以從陣列頭端(index=0)再開始，而形成環狀序列(Circular Queue)。

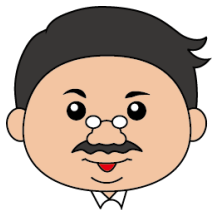




3-3-4 序列

head及tail的移動

- 在環狀序列中，tail及head的值在新增及移出資料時，必須除以 n 取餘數， n 是宣告的陣列儲存空間數，如此當這兩個變數達到陣列長度後，可以回到陣列頭端。



3-3-4 序列

- 在C語言中以%代表取餘數的運算式，在VB則是mod，以下圖為例， $n=5$ ，tail的變化如下：

tail = 0時新增資料：

$$\text{tail} = (\text{tail} + 1) \% n = (0 + 1) \% 5 = 1$$

...

tail = 3時新增資料：

$$\text{tail} = (3 + 1) \% n = (3 + 1) \% 5 = 4$$

tail = 4時新增資料：

$$\text{tail} = (\text{tail} + 1) \% n = (-1 + 1) \% 5 = 0 \text{ (回到陣列最前端)}$$



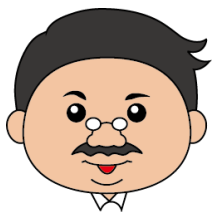
3-3-4 序列

空序列的條件

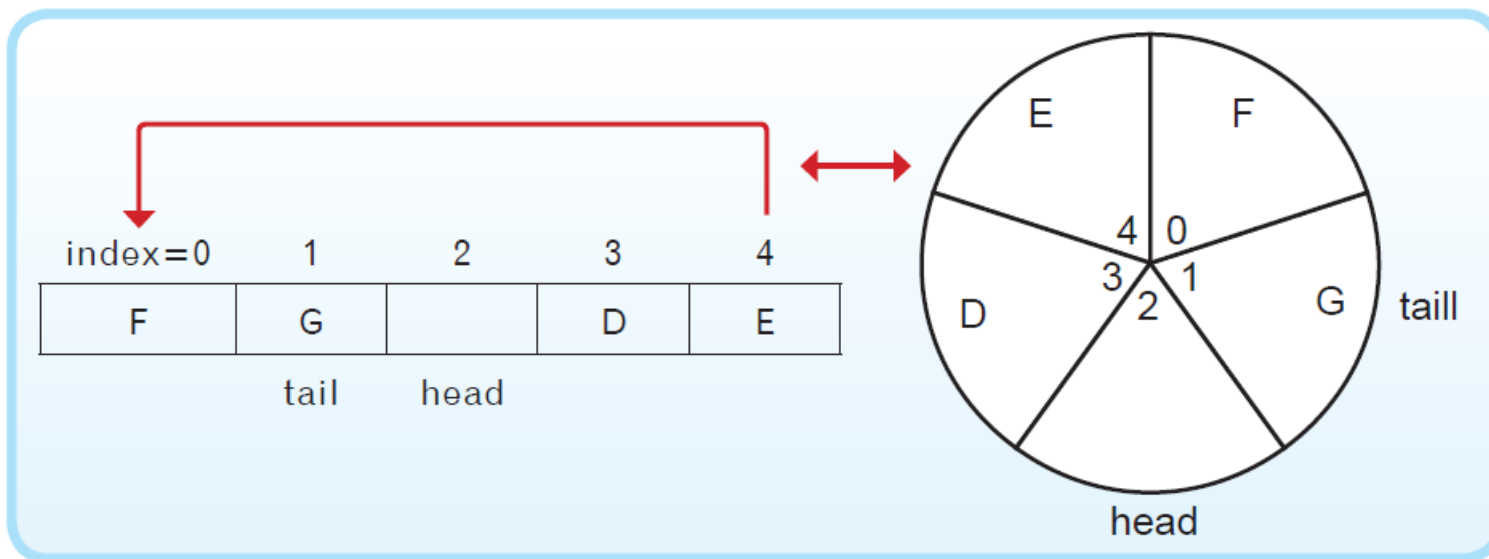
- 空環狀序列的條件與線性序列相同，都是 $\text{head} = \text{tail}$ ，此時不能移出資料。

滿序列的條件

- 環狀序列 $(\text{tail} + 1) \% n = \text{head}$ 時，即為滿序列。如下圖， $\text{tail} = 2$ ， $(2+1)\% 5 = 3 = \text{head}$ ，即表示為滿序列，不能再進行資料的新增了。



3-3-4 序列

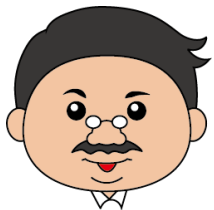


- 從上圖可以發現，環狀序列有一個儲存空間是不能夠用的，它的陣列使用率為 $n-1$ 。



3-3-4 序列

- 為什麼上圖的情況下，**index=2**那一個空間無法使用呢？
- 如果要使用**index=2**這一空間，則滿環狀序列的條件變成**head = tail**，這與空序列的條件相同，則程式將無法判斷序列是空或滿序列。



3-3-4 序列

環狀序列head及tail的初始值

- 線性序列的頭(head)及尾(tail)兩變數的初始值都是-1，進行資料新增及移除時會先+1，進入陣列開始的索引值=0。



3-3-4 序列

- 但是環狀序列中，序列完全未曾移出資料時，若head維持= -1，當滿序列($\text{tail} = n-1$)而程式又要新增資料時，滿序列的判斷條件 $(\text{tail} + 1) \% n = 0 \neq \text{head}(-1)$ ，並不會滿足滿序列的條件，此時tail則回到0這一儲存空間，這個空間的值將會被覆蓋掉，資料也就產生錯誤。



3-3-4 序列

- 環狀序列head及tail的初始值不能為-1，兩者可以都指定為0，當新增資料時tail會先加1再進行資料寫入，也就是tail從索引值1開始寫入。

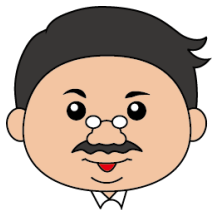




3-3-4 序列

- 環狀序列head及tail的初始值也可以指定為 $n-1$ ， n 是陣列的長度， $n-1$ 就是陣列的最後一個空間，這樣當tail新增資料時 $\text{tail} = (\text{tail} + 1) \% n = 0$ ，環狀序列就可以從陣列的索引0開始新增資料，而且符合 $\text{head} = \text{tail}$ 為空序列的條件。

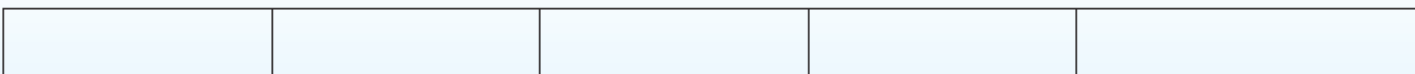




3-3-4 序列

```
var queue = Array(n); //宣告n個空間的字元陣列  
var circular_head = n-1;  
var circular_tail = n-1;
```

index = 0



tail = head = n-1



3-1

3-2

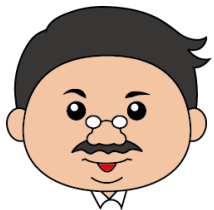
3-3



3-3-4 序列

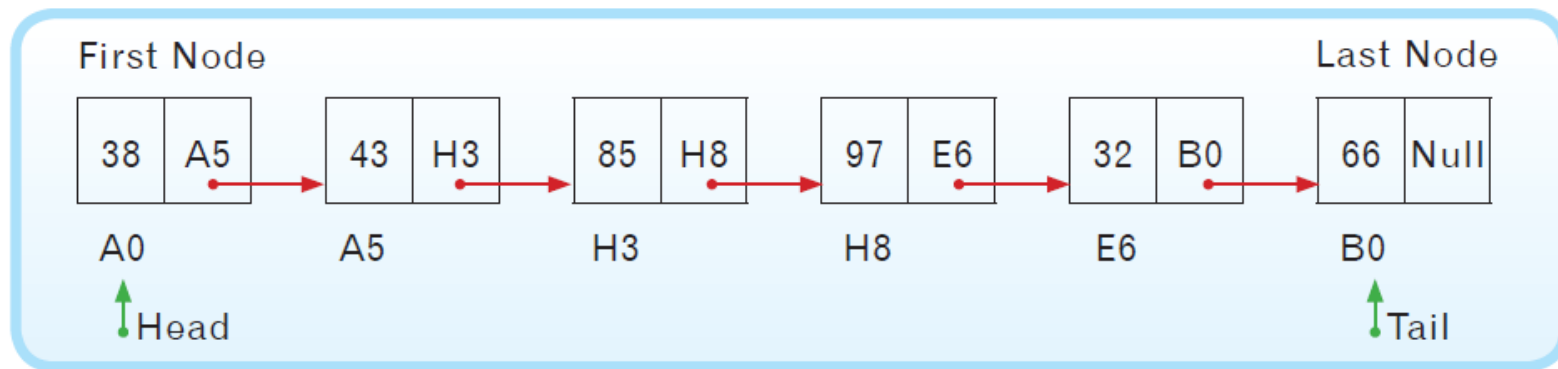
鏈結序列

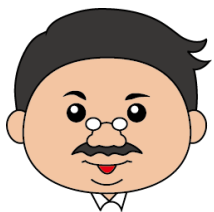
- 鏈結序列與鏈結串列的不同在於鏈結序列只能於最後一個(**Last Node**)節點增加資料，資料移除只能從第一個節(**First Node**)點，而且必須設下**head**及**tail**兩個節點(這兩節點不含值，只是個指標)，各指向第一節點及最後一個節點，**head**及**tail**兩個節點不能刪除。



3-3-4 序列

- 以下圖為例，head指向First Node的位址 (A0)，tail 指向 Last Node 的位址 (B0)。





3-3-4 序列

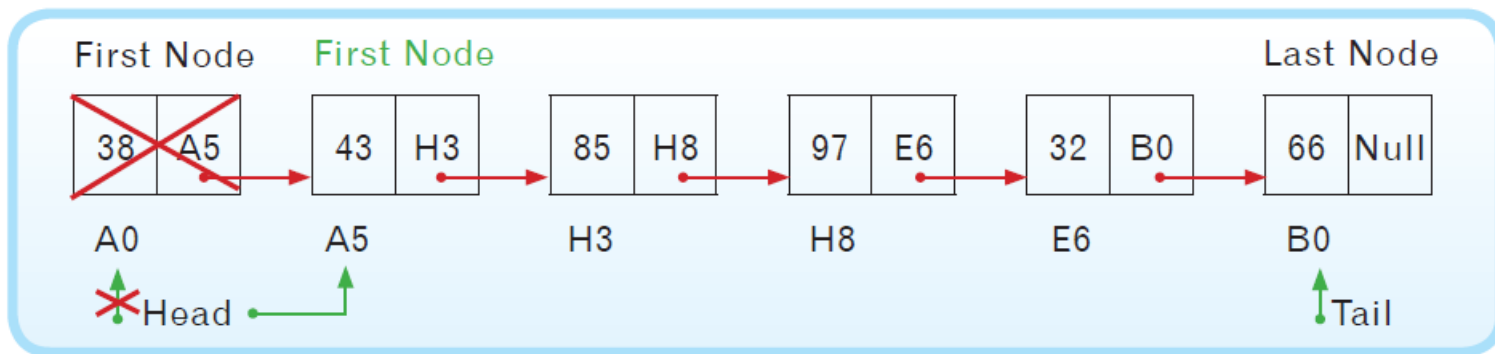
移出資料

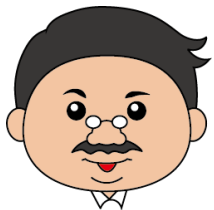
■ 在鏈結佇列中移出資料的步驟有3個：

1. 從head取得first Node的位址，取得first中的值(38)。

2. 取得first的指標值(A5)，改寫Head Node中的指標值，first node移至A5。

3. 釋放原first node的記憶體。



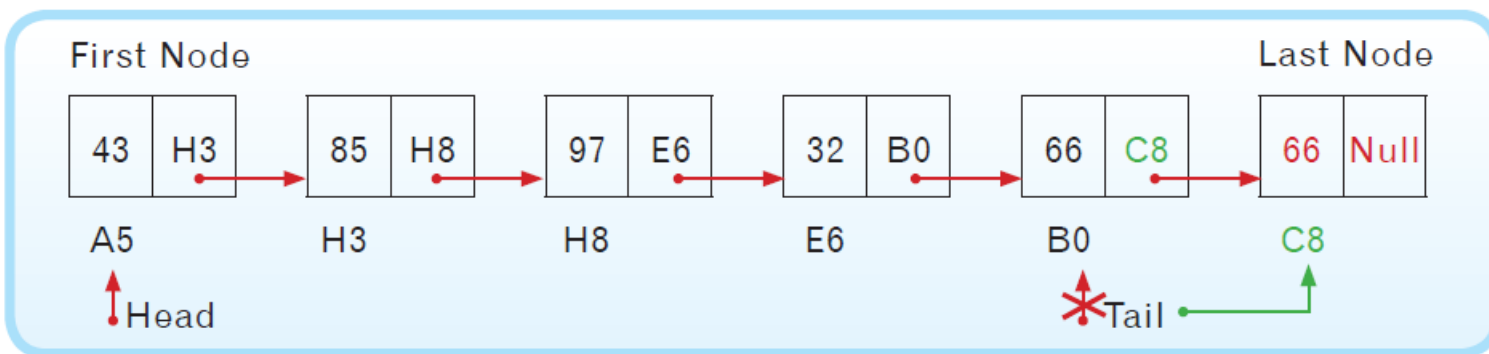


3-3-4 序列

新增資料

■ 在鏈結佇列中新增資料的步驟有3個：

1. 新增一個Node(位址假設為C8)，寫入值(假設為66)，並改寫此Node的指標為Null。
2. 從tail取得原Last Node的位址，改寫原Last Node指標(Null)為新節點的位址(C8)。
3. 改寫tail中的指標值，指向新Last node節點的位址(C8)。





3-3-4 序列

JavaScript序列的實現

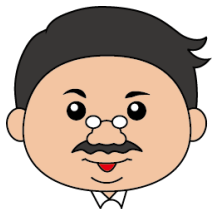
- 鏈結序列必須用到**指標(Pointer)**，在實作上比較不容易。
- 在JavaScript中，提供了以陣列結構實作出序列的基礎，而且實作方法相對的簡單許多。



3-3-4 序列

- 陣列push可以將資料加入陣列的尾端，另外提供了shift的方法，可以將陣列最前端(陣列索引值0)的資料刪除(移出)陣列中，即可達成序列先進先出的要求，但是移出並不會取得資料，因此必須以下表程式array_item0 = queue[0]這一行取得移出的資料。





3-3-4 序列

程式	說明
<code>var queue[];</code>	宣告一個陣列
<code>queue.push(1);</code>	進行enqueue的動作，加入三筆資料1 2 3
<code>queue.push(2);</code>	
<code>queue.push(3);</code>	
<code>var array_item0 = queue[0];</code> <code>queue.shift();</code>	進行dequeue的動作，取得序列最前端的值並移出序列，此例array_item0=1

