# CHAPTER 10 資料結構

# 10-1 陣列

- 假設班上只有5名同學，學號分別是1號到5號，且數學成績是整數，我們在C裡面可以如下宣告一個整數陣列叫作score，來儲存這些資料。

```
int score[5];
```

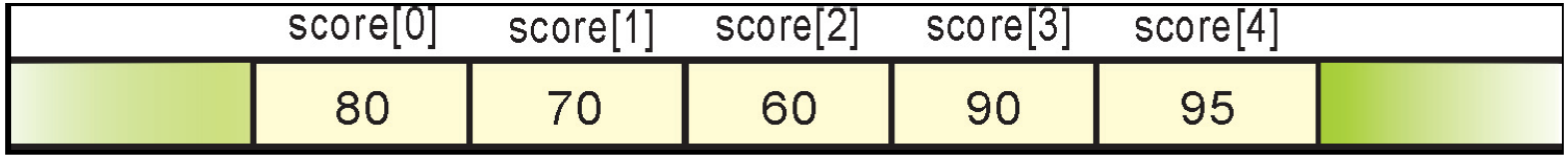
# 10-1 陣列

- 若是這些同學的成績分別是80、70、60、 90、95，則可以用下列的C指令將其指定到陣列裡面，注意到學號1的同學以註標0表示，學號2的同學以註標1表示，依此類推。

```
score[0] = 80;
score[1] = 70;
score[2] = 60;
score[3] = 90;
score[4] = 95;
```

# 10-1 陣列

- 在一般的程式語言裡，陣列的**邏輯順序**(logical order)和**實體順序**(physical order)是一樣的，也就是在記憶體裡，註標小的會排在註標大的之前。
- 這個成績陣列在記憶體裡的示意圖表示如下：



# 10-1 陣列

- 這樣的儲存方式， 是為了可以很快的決定某一個註標在記憶體的位置。
- 假設一個整數的大小是4 bytes ， 而 score[0] 在記憶體的位置是start，則任何一個註標x的位置(position)，都可以用下面這個公式算出來：

```
position(x) = start + x*4
```

# 10-1 陣列

- 舉例來說，score[2]的位置是start+8。在程式執行的時候，使用者要求任一個註標的資料時，都可以利用此公式很快的計算得到。
- 一般程式語言也允許定義更複雜的陣列資料結構。

# 10-1 陣列

- 假設班上這5位同學，我們不僅要記錄其數學成績，還要記錄其英文成績，也就是這些同學的成績資料如下表所示：

| | 學號1 | 學號2 | 學號3 | 學號4 | 學號5 |
|---|---|---|---|---|---|
| 數學成績 | 80 | 70 | 60 | 90 | 95 |
| 英文成績 | 65 | 75 | 85 | 81 | 74 |

同學的數學和英文成績

- 則可以宣告一個二維陣列如下：

```
int scores[2][5];
```

# 10-1 陣列

- 然後，所有同學的數學成績可以記錄在scores二維陣列的第一列，英文成績可以記錄在scores二維陣列的第二列。
- 如此一來，若要取出學號2號同學的數學成績，則表示式為scores[0][1]；若我們要取出學號5號同學的英文成績，則表示式為scores[1][4]。

# 10-1 陣列

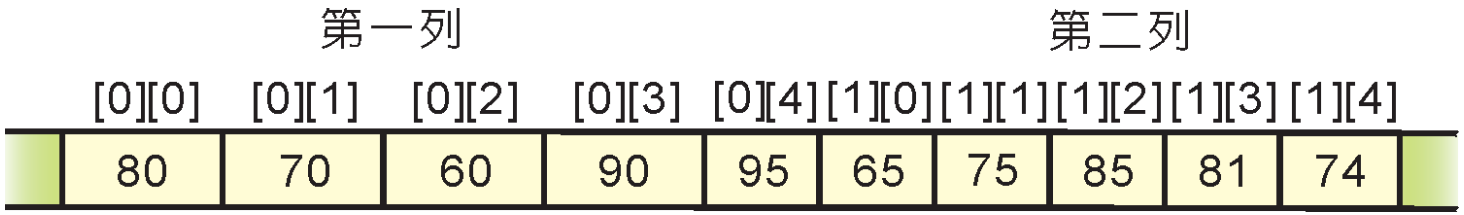
- 綜合而言，每個同學這兩科成績的對應註標如下表所示：

二維陣列的註標對應

| | 學號1 | 學號2 | 學號3 | 學號4 | 學號5 |
|---|---|---|---|---|---|
| 數學成績 | scores[0][0] | scores[0][1] | scores[0][2] | scores[0][3] | scores[0][4] |
| 英文成績 | scores[1][0] | scores[1][1] | scores[1][2] | scores[1][3] | scores[1][4] |

# 10-1 陣列

- 問題是，如此宣告出來的多維陣列，是不是會造成程式執行的時候，存取任一個註標資料的困難？答案是否定的。
- 通常系統在記憶體裡記錄多維陣列的方法，是先從第一列開始，把所有元素連續記錄在記憶體裡，然後接著記錄第二列，其示意圖如下：



# 10-1 陣列

- 在C程式語言裡，一列或一行可表示幾個元素，必須在宣告陣列時事先宣告好，這裡所謂的元素(element)，是指每一筆儲存在陣列裡的資料。
- 所以根據這些訊息，可以利用下列公式，事先推算出每一個註標在記憶體裡的位置。

```
position(x, y) = start + x*列大小 + y*元素大小
```

# 10-1 陣列

- 以此二維陣列來說，一列表示5個元素，一個元素是一個整數的大小，也就是4 bytes，所以公式可以進一步化簡為：

```
position(x, y) = start + (5x + y)*元素大小
               = start + (5x + y)*4
```

# 10-1 陣列

- scores[0][1] 在記憶體的位置，可算出為start+4；而scores[1][3]在記憶體的位置，則為start+32；也就是所有註標的位置都可以透過此公式很快的決定。
- 另外，在C語言裡，是先存放好第一「列」的元素，接著再存放第二「列」，依此類推，這樣的方式叫作「以列為主」(row major)。

# 10-1 陣列

- 至於有的程式語言，如FORTRAN，則採用「以欄為主」(column major)，也就是先存放好第一「欄」的元素，接著再存放第二「欄」，依此類推。
- 我們可以觀察到，「以欄為主」的記憶體存放位置的公式，會和「以列為主」的記憶體存放位置的公式不同，其公式如下所列：

```
position(x, y) = start + x*元素大小 + y*欄大小
```

# 10-1 陣列

- 以二維陣列scores為例，公式會如下所示：

```
position(x, y)        = start + (x + 2y)*元素大小
                      = start + (x + 2y)*4
```

- 所以，scores[0][1]在記憶體的位置，會是start+8，而非之前的start+4；至於scores[1][3]在記憶體的位置，則會變成start+28。

# 10-2 鏈結串列

- 可以利用指標建立鏈結串列，來表示不確定大小或會動態增減的資料。
- 鏈結串列是由一個個節點所組成的，繼續使用在第6-2節的範例，其節點的資料型態宣告如下：

```
struct node
{
        int data;
        struct node *next;
};
```

# 10-2 鏈結串列

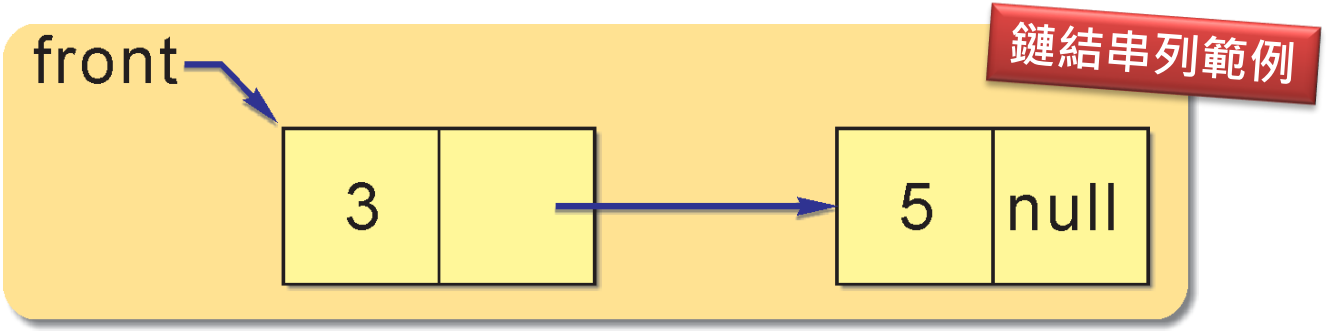
- 我們可以根據此資料型態宣告一個指標變數front，用來指到一個鏈結串列的起始節點。

```
struct node *front;
```

- 根據C語言的語法，若在宣告一個變數時前面加上符號「*」，則該變數就是指標變數，換句話說，變數front記錄的值會是起始節點在記憶體裡的位置。

# 10-2 鏈結串列

- 利用運算式「*front」指到該節點，「*front.data」則會傳回該節點在data欄位的值。
- 以下圖的鏈結串列為例，「*front.data」的值為3。



# 10-2 鏈結串列

- 另一種寫法是利用箭頭->，也就是「front->data」。
- 另外注意的是，null 在C語言具有特殊意義，代表了「空指標」，通常用來表示一個串列的結束。

# 10-2 鏈結串列

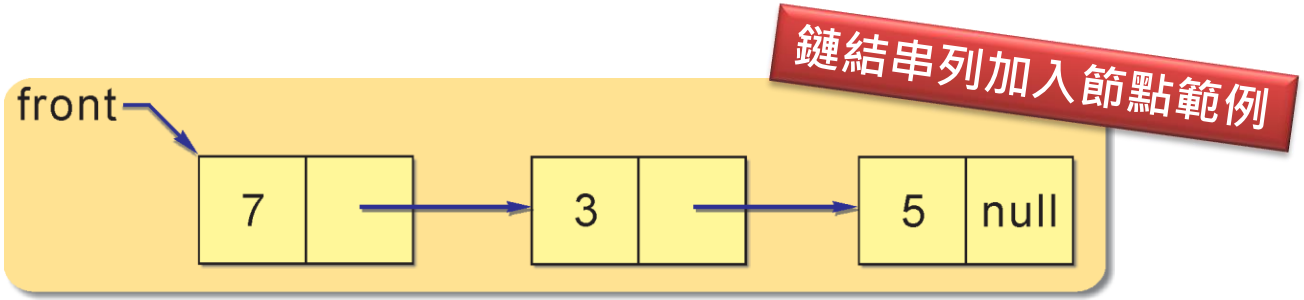
- 假設現在要把一個新的節點加入到鏈結串列的起點，可以定義一個程序叫作insert如下：

```
void insert(struct node *p, int new_item)
{
        struct node *temp = malloc(sizeof (struct
node));

        temp->data = new_item;
        temp->next = p;
        p = temp;
};
```

# 10-2 鏈結串列

- 假設我們呼叫此程序，在下圖的鏈結串列的前端，加入一個新的節點，其值為7，也就是執行「insert(front, 7)」。
- 則程式的執行步驟如下：



# 10-2 鏈結串列

步驟 1
- 利用malloc函數建立一個新的節點，並利用局部變數temp指到該節點

步驟 2
- 把數值7指定給節點temp的欄位data

步驟 3
- 把節點temp的欄位next設定如正式參數p的值，也就是將節點temp的欄位next指到p所指到的節點。注意到，由於正式參數p會對應到真實參數 front，所以新的節點會指到串列的第一個節點

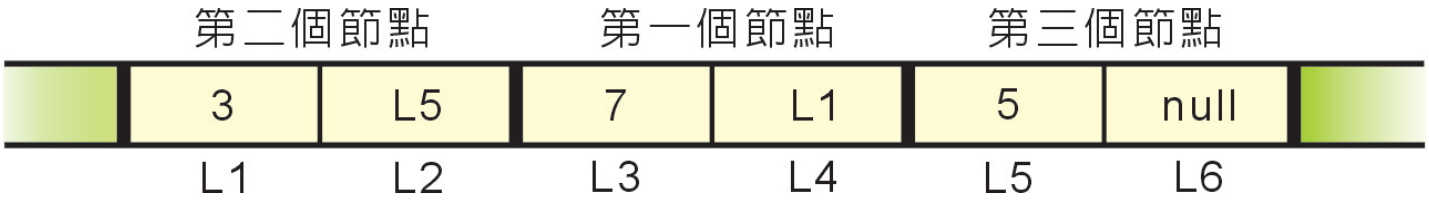
步驟 4
- 最後將參數p(也就是front)，指到新建立的節點

# 10-2 鏈結串列

- 另外值得注意的是，鏈結串列和陣列有一點很大的不同，就是鏈結串列的邏輯順序和實體順序並不一定相同。
- 當我們利用函數malloc向系統要一塊記憶體的空間時，系統會根據當時記憶體哪裡有空位，而把位址回傳給你，也許會在目前節點的前方，或是後方。

# 10-2 鏈結串列

- 下圖顯示上圖鏈結串列的可能實體順序，其中編號L1、L2、L3 等，代表記憶體的實體位置。




- 節點內容值為3的節點，即使在邏輯順序上是排在內容值為7的節點後面，但是在記憶體的實體順序上，則可能是排在其前面。

# 10-2 鏈結串列

- 可以推算出陣列裡元素的位置公式，而很快的知道陣列裡任一註標的位置；但是另一方面，要取出鏈結串列的某一個節點，只能依循事先建立好的指標，一一探訪中間經過的節點。

# 10-2 鏈結串列

- 以下的C程式，把一個鏈結串列內所有節點的內容值依照邏輯順序列出來：

```
void print_linked_list(struct node *p)
{
        printf( “The linked list contains the
following number:” );
        while (p != NULL)
        {
                printf( “%d” ,p->data);
                p = p-> next;
        }
}
```

# 10-2 鏈結串列

- changehead該程式會把第一個參數p指到的鏈結串列的起始節點，變成第二個參數q指到的鏈結串列的起始節點。把該程序再度列舉如下：

```
void changehead (struct node *p, struct node *q)
{
        struct node *temp;

        temp = p;
        p = p ->next;
        temp->next = q;
        q = temp;
}
```

# 10-2 鏈結串列

**步驟 1**

- 將局部變數temp指到第一個鏈結串列的起始節點。

**步驟 2**

- 將參數p指到第一個鏈結串列的第二個節點。

**步驟 3**

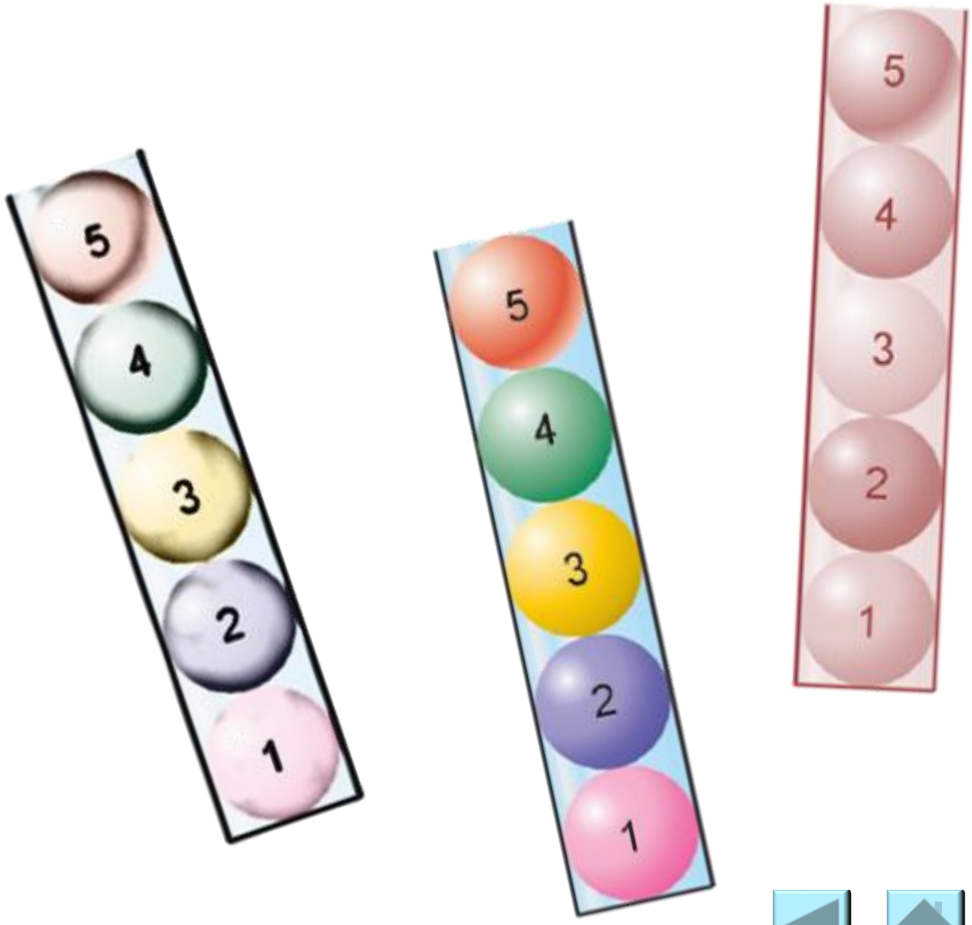
- 將節點temp的欄位next指到q所指到的節點，也就是第二個鏈結串列的起始節點；由於在第一個步驟，temp已經指到第一個鏈結串列的起始節點，所以此一步驟會把兩個串列的鏈結建立起來。

**步驟 4**

- 最後將參數q指到變數temp指到的節點。所以，現在第二個鏈結串列的起始節點，會是原本第一個鏈結串列的起始節點。

# 10-3 堆疊和佇列

- 堆疊
- 佇列
- 環狀佇列



# 堆疊

- 堆疊的概念，是處理一序列資料的時候，採用「後進先出」、「先進後出」的順序。
- 假設現在要在一個狹長的網球桶裡，依序放入編號1號到編號5號的網球，很明顯的，最早放進去的1號球會在球桶的最下方，而最後放進去的5號球會在球桶的最上方。

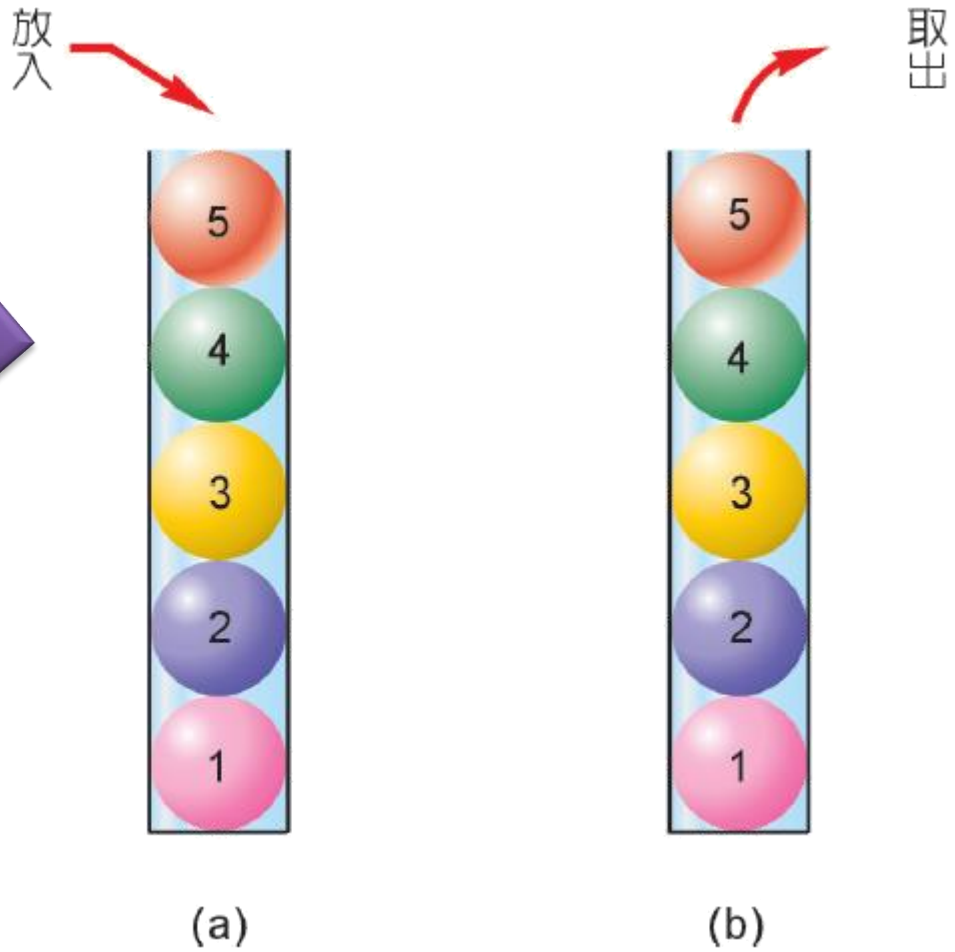
# 堆疊

- 當我們要用球的時候，由於該球桶的開口固定在上面，所以首先拿到的是球桶最上方的5號球，接著是4號球，最後才會拿到1號球。
- 假設我們預先知道只有10個整數要處理，我們可以如下宣告一個一維整數陣列來存放這些元素：

```
int stack[10];
```

# 堆疊

堆疊示意圖



# 堆疊

- 重點在於如何針對此陣列撰寫對應的程式，以實作「後進先出」和「先進後出」的想法。
- 也就是，必須適當的定義如何將資料放入堆疊，再如何將資料從堆疊取出，才能造成「後進先出」和「先進後出」的效果。

# 堆疊

- 為了達到此目的，我們還要記錄其他相關資訊。首先，為了知道目前堆疊內元素的個數，我們定義一個整數變數top，對應到最上層元素的註標，一開始設為-1，以表示空堆疊。

```
int top = -1;
```

# 堆疊

- 接著，我們定義將資料放入堆疊的程序push如下。注意到，我們會先增加變數top，也就是後放進去的元素會放在註標比較大的位置，而同時top會代表最後一個元素在陣列的註標：

```
void push (int data){
        top = top + 1;
        stack[top] = data;
}
```

# 堆疊

- 然後，要將資料從堆疊取出的話， 直接回傳陣列在top註標存放的資料即可；同時，我們要更改變數top的值，以表示堆疊內的元素減少。
- 相關的函數pop定義如下：

```
int pop( ){
        top = top -1;
        return stack[top+1];
}
```

# 佇列

- 佇列(queue)這種資料結構的操作方式和堆疊相反。佇列的概念， 是處理一序列資料的時候，採用「先進先出」、「後進後出」的順序。
- 假設現在在一個狹長的巷道裡，編號1號到編號5號的車子依序駛入，然後因為紅燈而停了下來，很明顯的，編號1號的車子會在最前面，最靠近燈號，其次為編號2號的車子，依此類推。

# 佇列

- 等到綠燈的時候，首先開出巷道的會是等在最前面的1號車，接著是2號車，最後才會是5號車。進入佇列和出來佇列的示意圖如下所示。


- 我們同樣利用陣列來實作佇列。假設我們預先知道只有10個整數要處理，我們可以宣告如下：

```
int queue[10];
```

# 佇列

- 以下我們提出相關的定義， 說明如何將資料放入佇列，再將資料從佇列取出，以實作「先進先出」和「後進後出」的想法。
- 為了適當的指出目前佇列內元素的個數，必須定義兩個整數變數front和rear，它們可用來對應到最前面和最後面元素的註標，一開始設為-1。

```
int    front = -1;
       rear = -1;
```

# 佇列

- 接著，我們定義將資料放入佇列的程序put，注意到我們更改的是最後面元素的註標，也就是變數rear會對應到佇列最後面元素的註標：

```
void put (int data){
        rear = rear + 1;
        queue[rear] = data;
}
```

# 佇列

- 然後，要將資料從佇列取出的時候， 根據的變數是front，因為它對應到最前面元素前一個位置的註標，相關的函數get如下：

```
int get( ){
        front = front +1;
        return queue[front];
}
```

# 環狀佇列

- 觀察佇列的相關程序，我們可以看到front和rear對應的註標會一直增加。
- 在前例中，我們宣告的陣列大小為10，所以當我們加入10個數字後，儘管我們已經又拿出5個數字，也就是陣列裡還有5個空間，還是無法再加入任何數字，因為已經超過了陣列合理註標的上限。所以，為了有效的利用空間，「環狀佇列」的資料結構被提了出來。

# 環狀佇列

- 為了在以下便於說明，我們假設陣列裡只能存放6個數字，然後front和rear這兩個變數，分別表示陣列的最前面和最後面註標，兩個的初始值都設定為0。

```
int    queue[6];
       front = 0;
       rear = 0;
```

# 環狀佇列

- 要將資料放入環狀佇列之前，首先必須先決定放入的位置，所根據的是對應陣列最後面的註標變數rear。
- 由於可以再度回到之前曾被使用過，但是現在已經是空的位置，所以我們使用運算子%，然後根據其計算所得的餘數來決定下一個要加入資料的註標位置。

```
rear = (rear + 1)%6;
```

# 環狀佇列

- 至於要將資料取出時，所根據的是對應陣列最前面的註標變數front，我們同樣需要利用運算子%取得其註標的位置，如下列公式所決定：

```
front = (front + 1)%6;
```

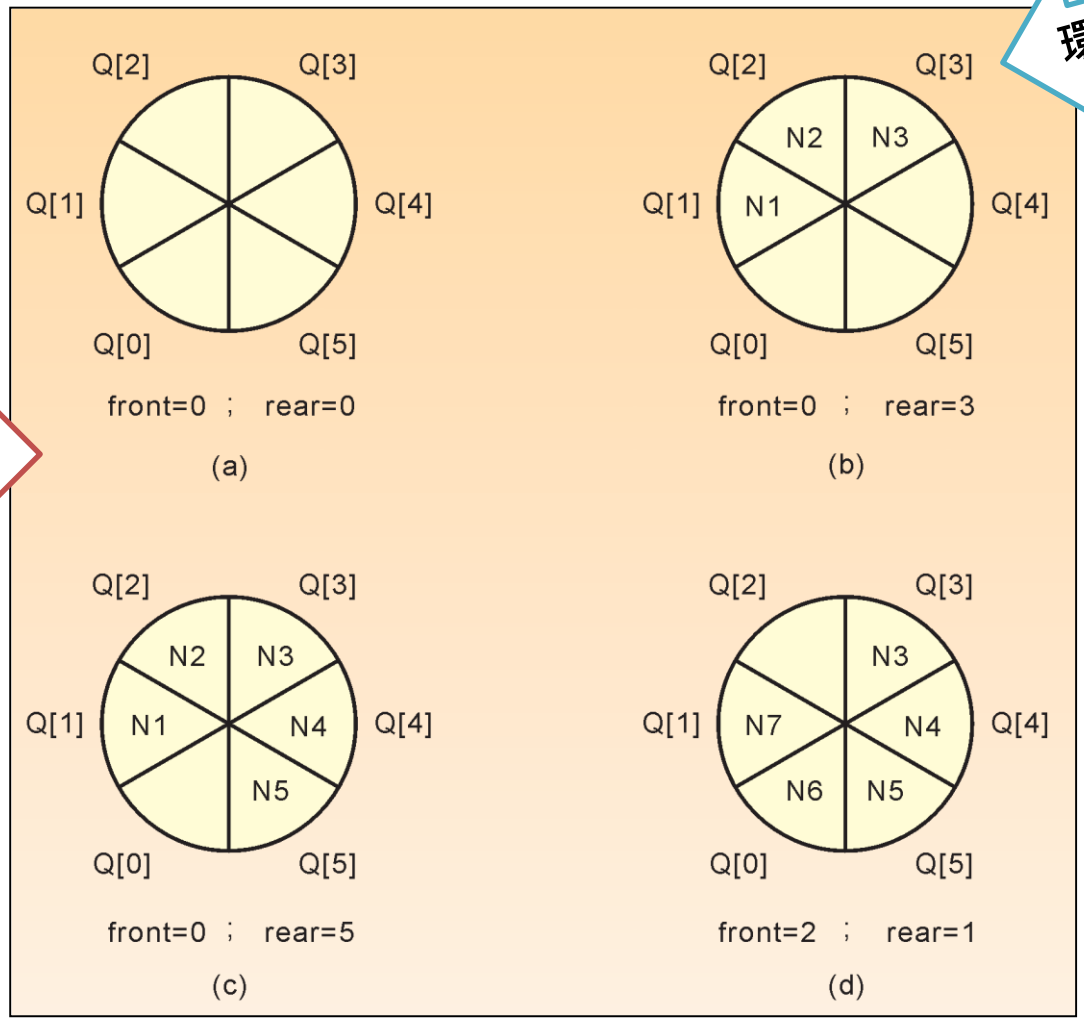
- 接下來我們說明如何利用front和rear這兩個變數的值，來判斷環狀佇列裡現在是滿的(full)還是空的(empty)。

# 環狀佇列

- 由於一開始當環狀佇列還沒存放任何東西的時候，front和rear這兩個變數都設為0，所以很直覺地可以推論出當front和rear這兩個變數的值相同時，佇列是空的，如下式所列：

```
front == rear
```

# 環狀佇列


環狀佇列示意圖
佇列的名稱簡寫為Q
Q[2]
Q[3]
Q[1]
Q[4]
Q[0]
Q[5]
front=0 ; rear=0
(a)
N2
N3
N1
front=0 ; rear=3
(b)
N4
N5
front=0 ; rear=5
(c)
N7
N6
front=2 ; rear=1
(d)

# 環狀佇列

- 一開始在上圖(a)中，佇列是空的，然後front和rear這兩個變數的值都為0。
- 之後如同一般的佇列，元素會加入佇列的尾端，所以當我們加入了數字N1、N2和N3之後，rear的值會變為3，如圖(b)所示。

# 環狀佇列

- 照理來說，佇列內還剩下3個位置，似乎還可以再加入3個數字，但是當我們加入N4、N5，此時的rear變數值為5，如圖(c)所示。
- 若我們要繼續加入N6，則根據之前的公式去計算陣列的註標，我們會得到下面的結果。

```
rear = (rear + 1)%6 = (5 + 1)%6 =0;
```

- 也就是rear的值計算為0。

# 環狀佇列

- 若是我們真的將N6加入Q[0]的話，等到之後我們要去此環狀佇列取資料時，由於front和rear的值此時皆為0，根據之前的判斷式，會判斷此佇列為空佇列，也就是儘管佇列是滿的，卻會被誤判成空的。
- 所以，環狀佇列一個很重要的性質是，當我們宣告環狀佇列裡有6個空間時，我們最多只能表示5個元素。

# 環狀佇列

- 我們繼續對此環狀佇列處理：取出N1和N2，使得front的變數值變成2；再加入N6和N7，使得rear的變數值變成1。
- 此時佇列仍然是滿的，如圖(d)所示。所以我們可以推論出，當rear在front順時針方向的後一位時，佇列是滿的，也就是如下式所列：

```
(rear+1)%6 == front
```

# 環狀佇列

- 根據以上的討論， 我們將資料加入環狀佇列的程序 put定義如下：

```
void put (int data){
        rear = (rear + 1)%6;
        if (front == rear)
        {
                printf( “Queue is full” );
                return;
        }
        queue[rear] = data;
}
```

# 環狀佇列

- 然後，將資料從環狀佇列取出的函數get，如下所列：

```
int get( ){
        if (front == rear)
        {
                printf( "Queue is empty" );
                return;
        }
        front = (front +1)%6;
        return queue[front];
}
```

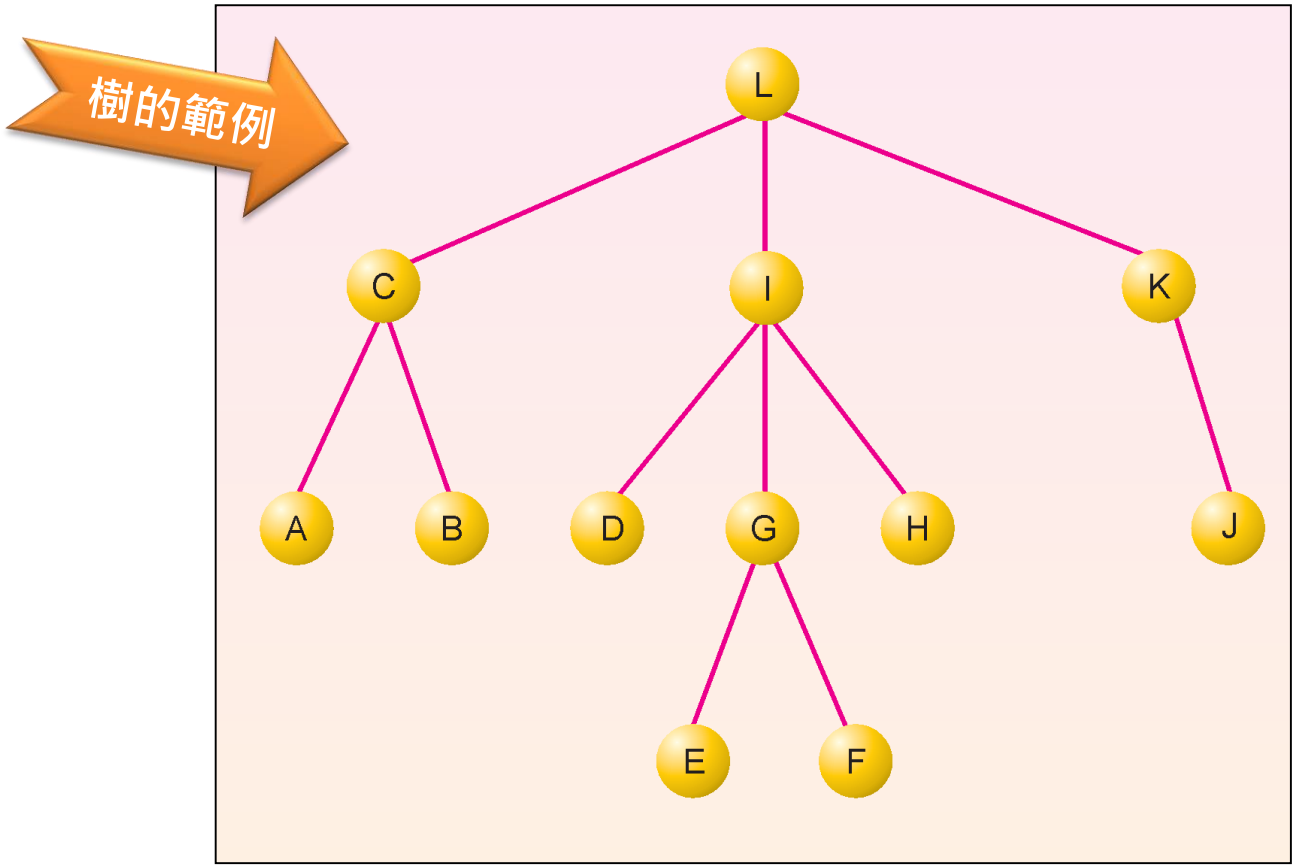
# 10-4 樹狀結構

- 樹(tree)在資訊科學裡是一種很重要的技巧，有很多專門的課程在教導樹的定義和應用。
- 在此節中，我們會介紹樹的基本定義和對應的程式。在大自然裡的樹木，是由底下的樹根(root)往上長出茂密的枝葉(leaf)；在資訊科學裡的樹有類似的定義，只是是反過來由樹根往下長出葉子，下圖就是一個樹的例子。

# 10-4 樹狀結構

- 樹是由節點(node)和邊(edge)所構成，而樹中的節點又可細分為三種：
  - 外部節點(external node)：又稱作葉節點，位於樹的最下層，如編號E、F、H等的節點。
  - 內部節點(internal node)：不是外部的節點，如編號 C、I、G等的節點。
  - 根節點(root node)：位於最上層的節點，如編號L的節點。

# 10-4 樹狀結構



# 10-4 樹狀結構

- 樹具有下列特殊性質：
  - 只有唯一一個根節點。
  - 樹中沒有迴圈(loop)，也就是任一節點循著邊往下走的話，不可能走回自己。
  - 任兩點只有唯一路徑。譬如說，節點E要走到節點I的話，一定會經過節點G，而沒有其他方法；另一個例子，從節點J要走到節點C的話，也一定會經過節點K和節點L。

# 10-4 樹狀結構

- 樹的高度
- 樹的高度(height)此為從根節點到樹中所有葉節點的最長可能路徑。
- 以課本圖10-6為例，圖中共有7個葉節點，而我們可以看到從根節點L到葉節點E或F的路徑長度為4(也就是途中經過4個節點)，比起根節點到其他葉節點的長度都還長，所以這棵樹的高度為4。

# 10-4 樹狀結構

- 樹的階層
- 樹的階層(level)代表任何一個節點，距離根節點的距離。我們可以看到，根節點L的階層為0，內部節點I的階層為1，至於在第2階層的節點，包含A、B、D、G、H、J等節點。

# 10-4 樹狀結構

- 祖先節點和父節點
- 若是考慮某1個節點，和該節點往上走到根節點的那一條唯一路徑，則在該路徑上的所有節點(不包含自己)，都是該節點的祖先節點(ancestor node)。
- 以圖10-6的節點F為例，它的祖先節點有G、I、L三個節點。我們可以觀察到，祖先節點包含它的父節點(parent node)G，也就是最靠近該節點的祖先節點。

# 10-4 樹狀結構

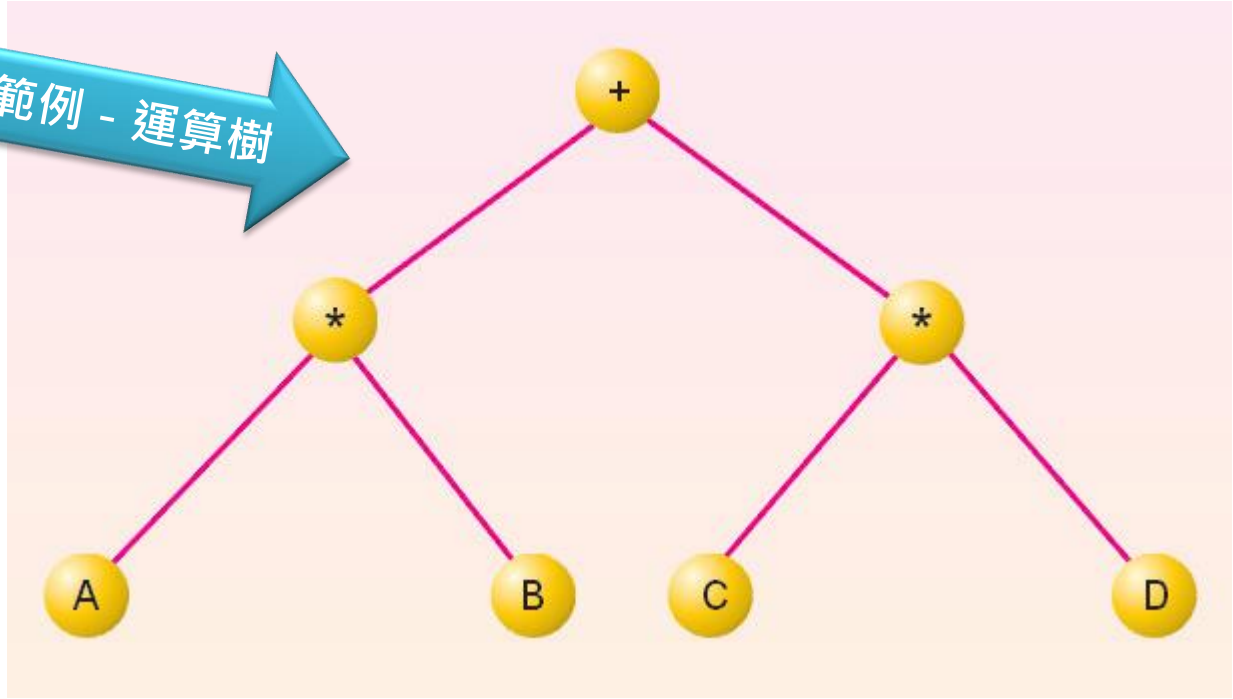
- 子孫節點和子節點
- 考慮某1個節點，和該節點往下走到葉節點的所有可能路徑。那麼，在這些路徑上的所有節點(不包含自己)，都是該節點的子孫節點(descendent node)。
- 以圖10-6的節點I為例，它的子孫節點有D、G、E、F、H等節點。我們也可以觀察到，子孫節點(child node)包含它的三個子節點D、G、H，也就是最靠近該節點的子孫節點。

# 10-4 樹狀結構

- 所謂的二元樹，就是每一個節點最多只有2個子節點(可能沒有子節點，或是只有1個) 。
- 也稱作運算樹(expression tree)，是將一個算數運算式以樹狀結構表示，其中運算子(operator)為父節點，運算元(operand)為子節點。至於圖10-6的樹不是二元樹，因為我們可以看見節點I有三個子節點。

# 10-4 樹狀結構

二元樹範例 - 運算樹

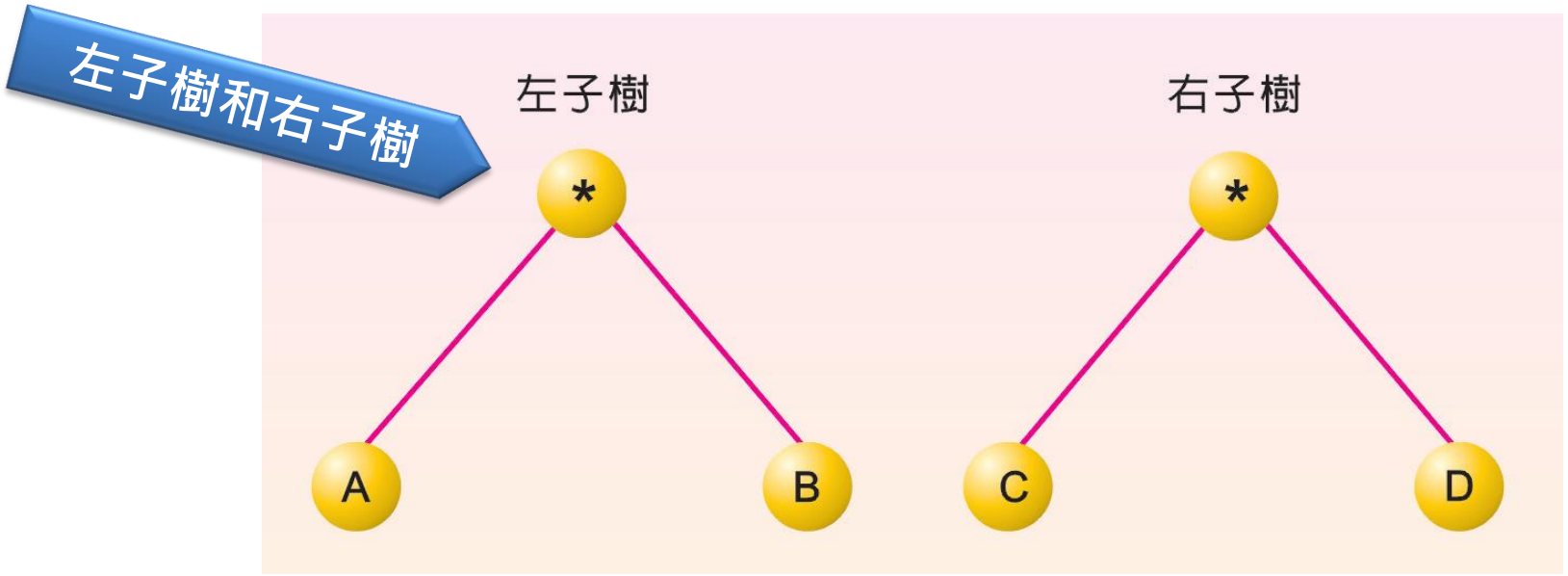


# 10-4 樹狀結構

- 針對二元樹的每一個節點，位於左邊的子節點，稱為左子節點(left child node)；若是以該左子節點為根節點，則所對應的樹稱為左子樹(left subtree)。
- 相同的，我們稱位於右邊的子節點為右子節點(right child node)，而對應的子樹稱作右子樹(right subtree)。

# 10-4 樹狀結構

- 以二元樹的根節點為例，其左子樹和右子樹描繪於下圖中。



# 10-4 樹狀結構

- 接下來我們說明如何實作二元樹，首先定義樹中每一個節點的資料型態，假設每一個節點存放一個字元：

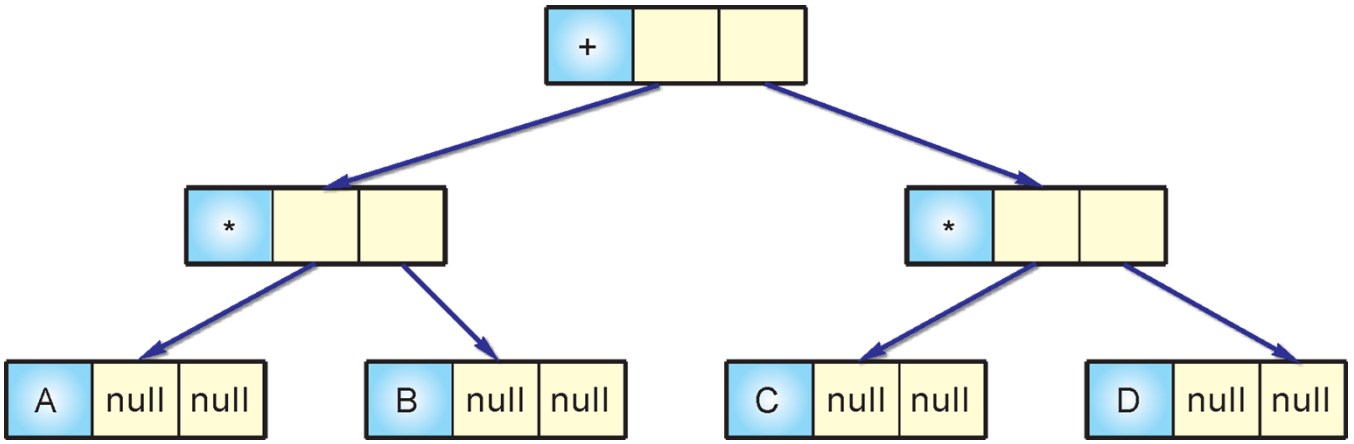
```
struct node
{
        char data;
        struct node *left;
        struct node *right;
};
```

# 10-4 樹狀結構

- 由於每一個節點最多有兩個子節點，我們將左子節點(或左子樹)以指標left表示，而將右子節點(或右子樹)以指標right表示，以此將一棵二元樹建立起來。
- 注意：若是只有一個子節點或沒有子節點的話，就以空指標null表示。

# 10-4 樹狀結構

- 若是將該圖與前面第10-2節的鏈結串列做比較，我們可以看出來，二元樹的每個節點定義了兩個指標，可看作是較複雜的鏈結串列。




二元樹的實作示意圖

# 10-4 樹狀結構

- 將二元樹建立起來之後，一個最常見而基本的運算，就是把整棵樹走一遍，也就是**探訪**(traverse)所有的節點。
- 二元樹的三種探訪順序如下：
  - 前序法(preorder)
  - 中序法(inorder)
  - 後序法(pos torder)。

# 10-4 樹狀結構

- 若是我們將圖10-7分別以這三種探訪順序走一遍，則正好會得到三種不同的運算式表示法。
- 其中，前序法(preorder)為先表示運算子，再表示運算元；中序法(inorder)為先表示第一個運算元，接著是運算子，最後再表示第二個運算元；後序法(postorder)會先表示兩個運算元，最後再表示運算子。

# 10-4 樹狀結構

- 我們將圖10-7的運算樹分別以這三種探訪順序得到的結果列在下面：

| 前序法 (preorder) | 中序法(inorder) | 後序法 (postorder) |
| --- | --- | --- |
| +*AB*CD | A*B+C*D | AB*CD*+ |

# 10-4 樹狀結構

- 所謂的遞迴程序，就是在程序的本體中，又呼叫到自己本身。
- 以大家耳熟能詳的階乘函數(factorial function)為例，在下列的第一式中，我們定義0的階乘為1；至於在第二式中，我們利用n-1的階乘來計算n的階乘，這就是遞迴的觀念：

```
fact(0) = 1;
fact(n) = n*fact(n-1);  (if n >= 1)
```

# 10-4 樹狀結構

- 在處理樹的演算法中，我們常使用到遞迴的觀念，是因為樹中的每一個節點都有相同的特性，而且前面處理的結果，會影響到後面，就如同階乘函數一般。
- 這三個程序會在探訪節點的時候，同時將該節點表示的字元列出來：

# 10-4 樹狀結構

- 第一個是前序法的程序 preorder”， 我們先將參數p對應的節點，也就是父節點的資料先列印出來，接著再遞迴呼叫此程序處理左子節點。
- 等到進入遞迴呼叫時，此左子節點會再度被視作是根節點，然後左子樹會依照一樣的方式被處理。
- 等到左子節點對應的整棵樹都列印出來之後，該遞迴呼叫結束處理，也就是回到最原始的狀態。

# 10-4 樹狀結構

- 此時程式會接著繼續進行下一個遞迴呼叫，也就是「preorder(p->right)」，以處理右子節點(或右子樹)。完整的程序如下所列：

```
void preorder(struct node *p)
{
        if (p != NULL)
        {
                printf( "%c" ,p->data);
                preorder(p->left);
                preorder(p->right);
        }
}
```

# 10-4 樹狀結構

- 第二個是中序法的程序inorder，與前序法不同的是，我們直接遞迴呼叫此程序處理左子節點；等到左子樹都列印出來之後，再列印原先參數p對應的節點，也就是父節點，最後再處理右子節點(右子樹)：

```
void inorder(struct node *p)
{
        if (p != NULL)
        {
                inorder(p->left);
                printf( "%c" ,p->data);
                inorder(p->right);
        }
}
```

# 10-4 樹狀結構

- 最後一個是後序法的程序postorder，我們先進行兩個遞迴呼叫，將左子樹和右子樹的資料都列印出來，最後再處理參數p對應的節點，也就是父節點：

```
void postorder(struct node *p)
{
        if (p != NULL)
        {
                postorder(p->left);
                postorder(p->right);
                printf( "%c" ,p->data);
        }
}
```