第三章
堆疊(Stack)

本章學習目標

1. 讓學生了解日常生活中有許多例子都是堆疊的應用。
2. 讓學生了解堆疊的運作原理及在運算式的應用。
3. 讓學生了解遞迴的使用及條件，進而設計比較簡潔的程式。
4. 讓學生了解如何利用遞迴函數來解決典型的河內塔問題。

本章內容

3-1 堆疊(Stack)

堆疊(Stack)是一種**後進先出**(Last In First Out, LIFO)的**有序串列**，亦即**資料處理的方式**都是**在同一邊進行**，也就是由**相同的一端**來進行**插入**與**刪除**動作。

我們**日常生活**中，也有一些是**堆疊的例子**，例如**堆盤子**、**書本裝箱**等，都是**一層一層的堆上去**，如果想要**取出**箱子中某一本書，也只能從**最上面開始取出**。

【定義】

1. 一群相同性質元素的組合，即有序串列(ordered List)。
2. 具有後進先出(Last In First Out, LIFO)的特性。
3. 將一個項目放入堆疊的頂端，這個動作稱為Push(加入)。
4. 從堆疊頂端拿走一個項目，這個動作稱為Pop(取出)。
5. Push/Pop的動作皆發生在同一端，則稱此端為Top(頂端)。
6. 要取出資料時，則只能從Top(頂端)取出，不能從中間取出資料。

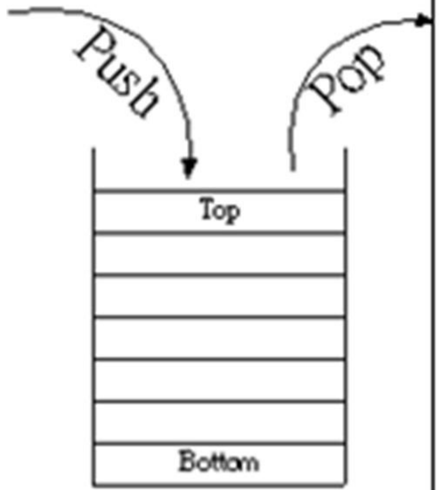
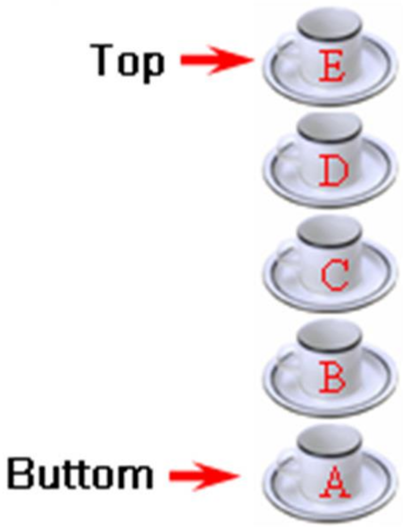
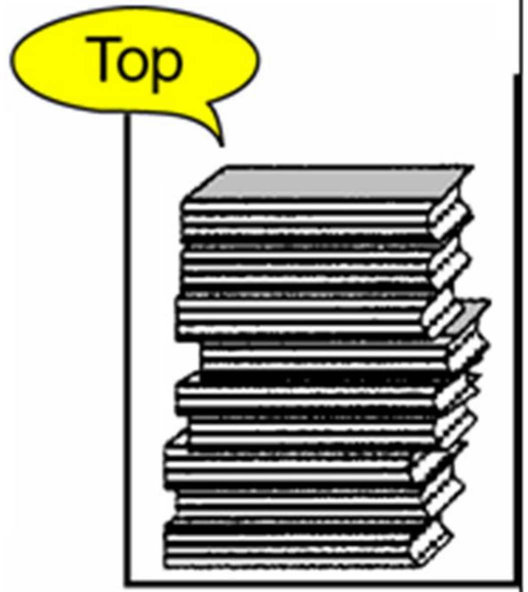
<table>
<tr><th>堆疊的基本運作</th><th>實例 1：自助餐廳的盤疊</th><th>實例 2：書本裝箱</th></tr>
<tr>
<td>
</td>
<td>
</td>
<td>
</td>
</tr>
</table>

圖 3-1 堆疊的例子

【堆疊常用專有名詞】

1. Push　　：加入新項目到堆疊的頂端。
2. Pop　　：取出堆疊頂端一個項目。
3. TopItem　　：查看堆疊頂端的項目內容。
4. IsEmpty　　：判斷堆疊是否為空，若為空則傳回真(True)，否則傳回假(False)。
5. IsFull　　：判斷堆疊是否為滿，若為滿則傳回真(True)，否則傳回假(False)。

【堆疊的兩個動作】

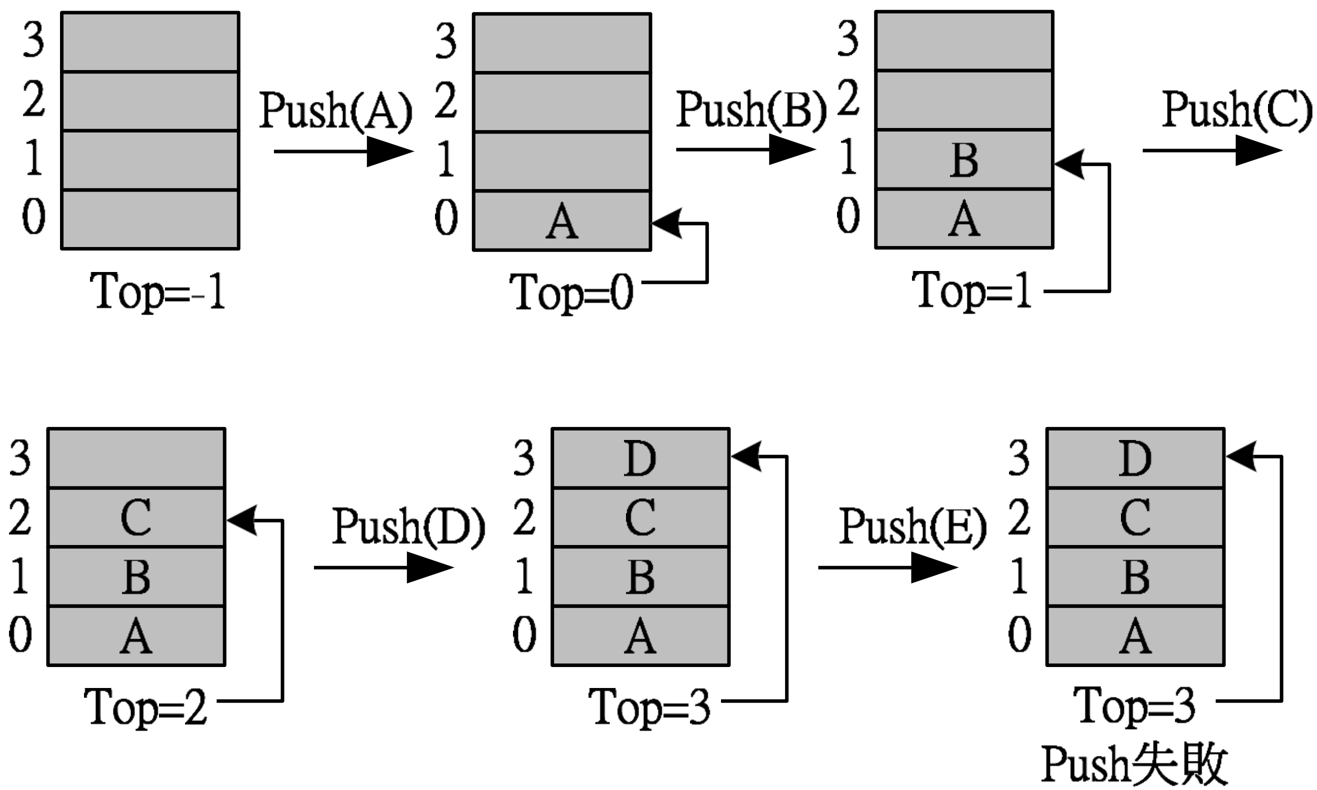
1. Push(**加入**)：**加入**新項目到堆疊的**頂端**。

【演算法】

```
Procedure Push(item, Stack)
 Begin
  if (Top= N-1)                //如果Top指標指到堆疊頂端
      Stack Is Full;           //代表堆疊已滿
  Else                         //如果不是，則
  {
    Top=Top+1;                 //指標位址加1
    Stack[Top]=item;           //再將資料加入到指標所在的堆
疊中
  }
 End
End Procedure
```

【堆疊的兩個動作】(續)

【運作過程】



【堆疊的兩個動作】(續)

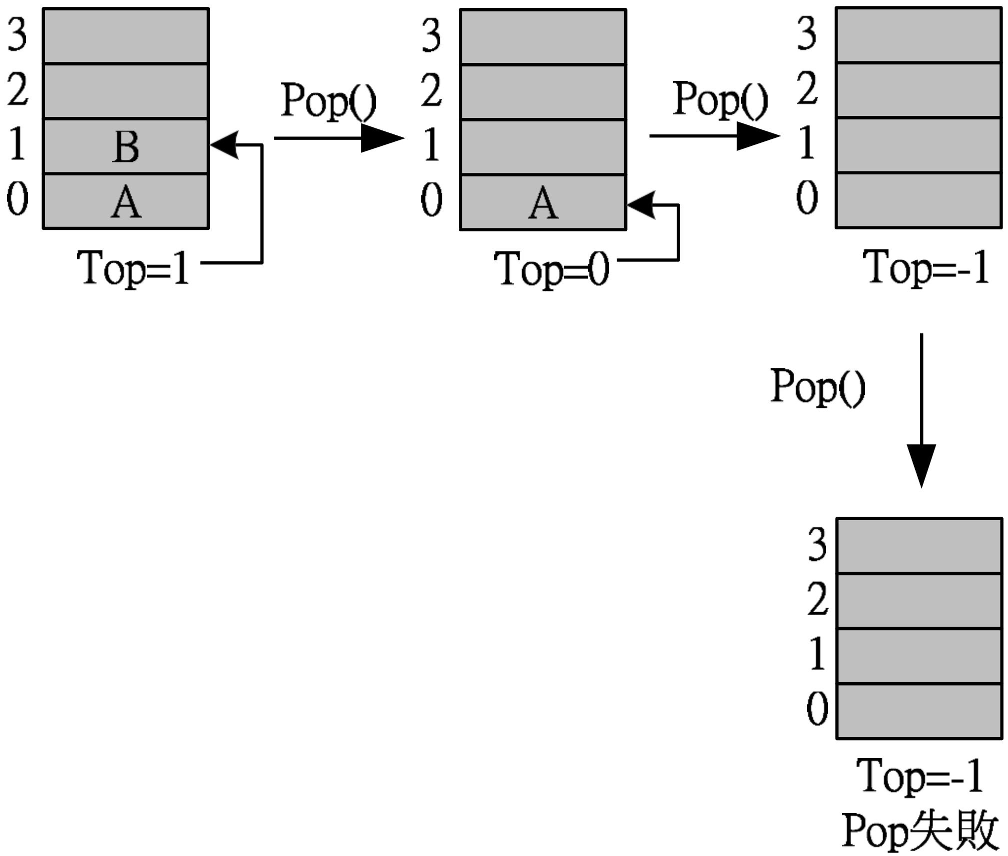
2. Pop ：取出堆疊頂端一個項目。

【演算法】

```
Procedure Pop(item, Stack)
 Begin
  if (Top=-1)                //如果Top指標為-1
       Stack Is Empty;       //代表Stack為空
  else                       //否則
   {
    item=Stack[Top];         //將資料從堆疊頂端取出
    Top=Top-1;               //再將指標位址減1
   }
 End
End Procedure
```

【堆疊的兩個動作】(續)

【運作過程】

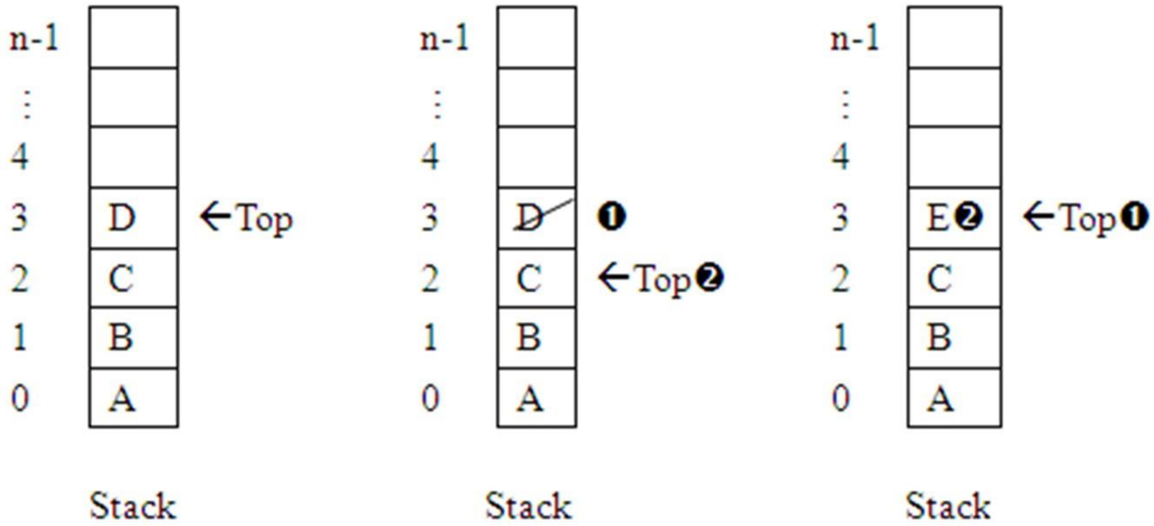


【舉例】假設有一個空的Stack，實施下列的動作：

Push A Push B Push C Push D	Pop	Push E

則最後Stack的內容為何？

【解答】ABCE



❶Pop：刪除D
❷Top=Top-1

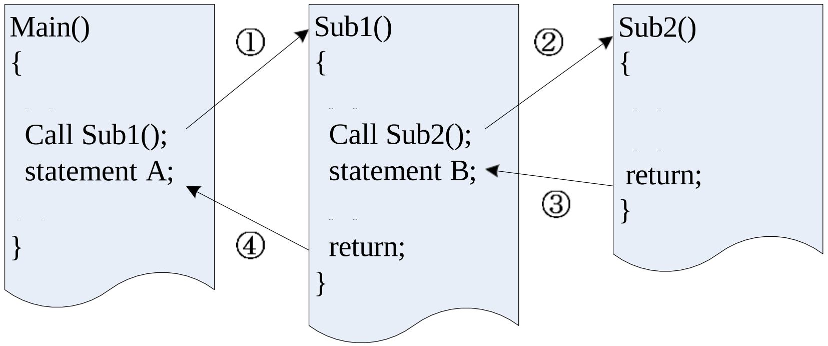
❶Top=Top+1
❷Stack[Top]=E

【堆疊的應用】

1.副程式呼叫與返回	6.巨集呼叫
2.遞迴程式呼叫與返回	7.多元處理
3.運算式之轉換與求值	8.圖形(Graph)的深度搜尋
4.中斷處理	9.資料反序輸出(例如：abc→cba)
5.二元樹追蹤	10.自助餐廳取餐盤的行為。

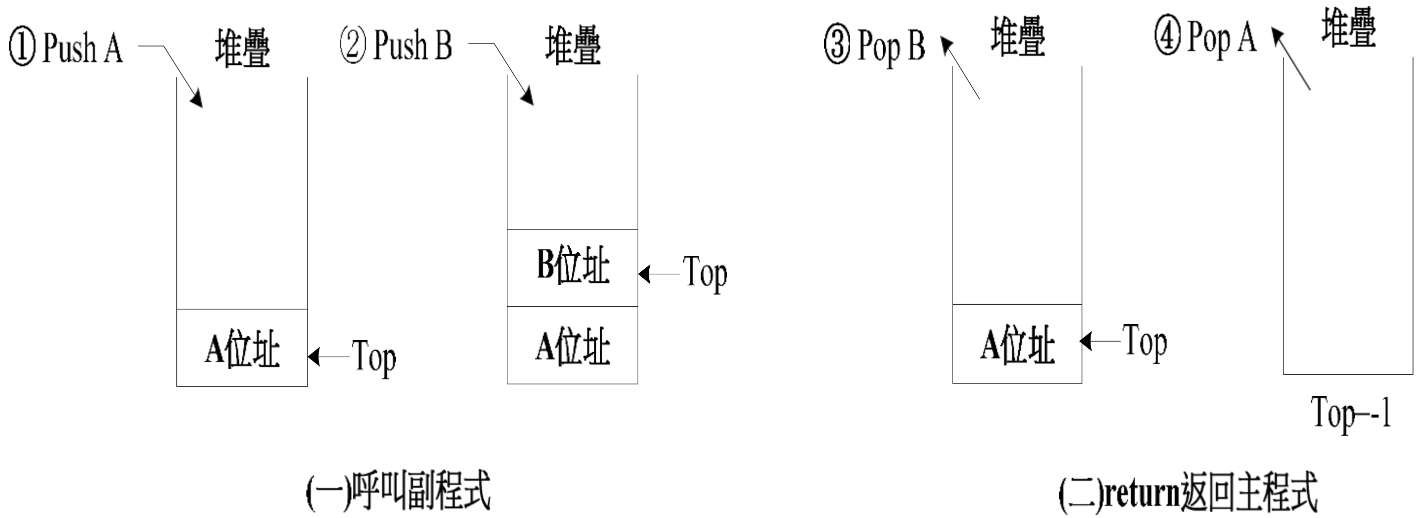
【舉例】

在電腦中，我們可以利用**堆疊**具有**後進先出**的特性，來解決副程式的呼叫。



【作法】呼叫副程式時，必須先將**返回位址**暫時**儲存**到「**堆疊**」中。

【過程】如下所示：



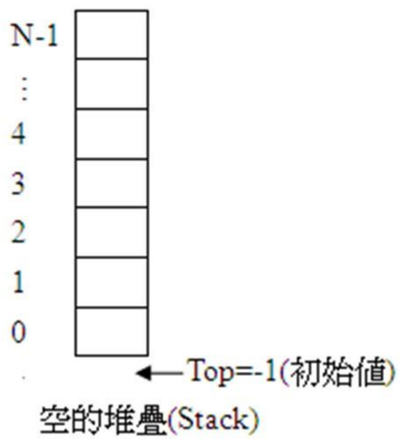
3-2 以陣列來製作堆疊

製作堆疊最常用的方法就是利用**一維陣列**。

1.**建立堆疊結構**：Create(Stack)　//**指建立一個空的**Stack

【演算法】

```
Procedure Create(Stack)
 Begin
 {
    #define N 10         //定義堆疊大小
     char Stack[N];     //以陣列Stack當作堆疊
     int Top=-1;  //Top記錄目前堆疊頂端的索引值，初始值設為-1表示
                        堆疊為空
  }
 End
End Procedure
```



2.將資料加入堆疊：Push(item, Stack)

指將資料(item)插入到Stack中，並且成為Top頂端元素；

如果堆疊已滿，則無法進行。

【演算法】

```
Procedure Push(item, Stack)
 Begin
 if (Top= N-1)                 //如果Top指標指到堆疊頂端
       Stack Is Full;          //代表堆疊已滿
 Else                          //如果不是，則
  {
    Top=Top+1;                 //指標位址加1
    Stack[Top]=item;           //再將資料加入到指標所在的
堆疊中
  }
 End
End Procedure
```

【實例】

假設在空的Stack中**連續加入4個資料項**，分別為A,B,C,D，

在加入之後Stack**的狀態**如下圖所示：

N-1		
⋮		
4		
3	D	←Top=3
2	C	
1	B	
0	A	

Stack

說明：Top**指標**位址每次**先加1**，再將**資料項加入**到**堆疊中**。

3.從堆疊取出資料：Pop(item,Stack)

指刪除Stack的Top頂端元素，如果堆疊是空，則無法進行。

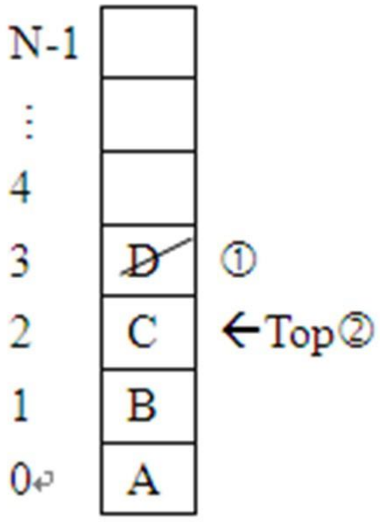
【演算法】

```
Procedure Pop(item, Stack)
 Begin
  if (Top=-1)                   //如果Top指標為-1
      Stack Is Empty;           //代表Stack為空
  else                          //否則
   {
    item=Stack[Top];            //將資料從堆疊頂端取出
    Top=Top-1;                  //再將指標位址減1
   }
 End
End Procedure
```

【實例】

假設在**空的**Stack中**連續加入4個資料項**，分別為A,B,C,D，

則在**刪除D**時，其Stack的變化如下圖所示：



堆疊(Stack)

說明：①**先取出**D

②再將**指標**Top**位址減**1

4.傳回堆疊Top頂端元素：**TopItem(Stack)**

指**傳回**Stack的Top**頂端元素**，如果**堆疊是空**，則**無法進行**。

【演算法】

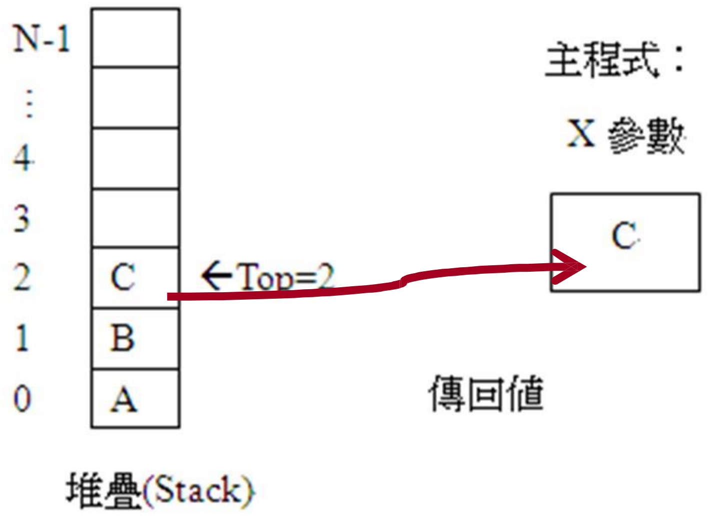
演算法：傳回堆疊**Top**頂端元素

```
Procedure TopItem(Stack)
 Begin
  if (Top=-1)                  //如果Top指標為-1
      Stack Is Empty;          //代表堆疊為空
  else                         //否則
    return Stack[Top];         //傳回堆疊頂端的元素
 End
End Procedure
```

【實例】

假設目前在堆疊中**已經有三個資料項**，分別為A,B,C，

此時，欲傳回**堆疊頂端的元素**時，其Stack的變化為何？



說明：①先判斷Top指標是否為-1，如果不是，則
②傳回堆疊頂端的元素

5.**判斷堆疊是否已滿**：IsFull(Stack)

指用來判斷Top**指標是否等於N-1，若是則傳回**True**，否則傳回**False。

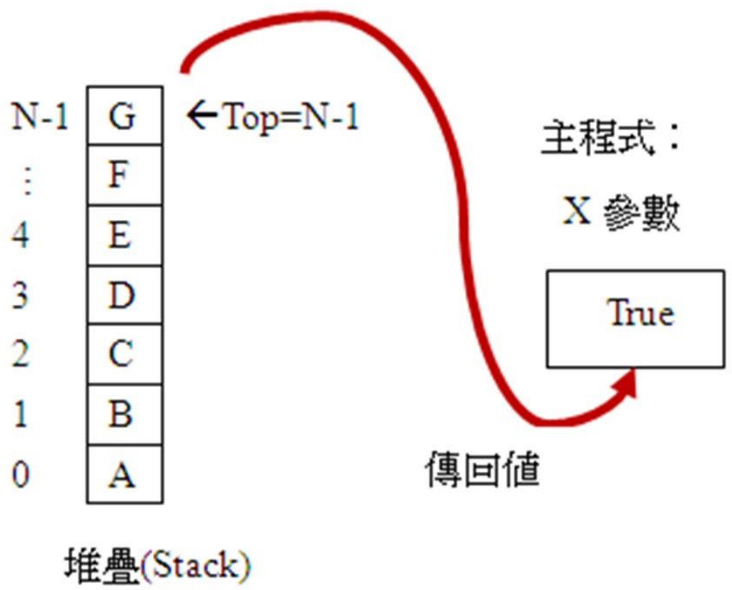
【演算法】

演算法：判斷堆疊是否已滿

```
Procedure IsFull(Stack)
 Begin
  if (Top=N-1)                  //如果Top指標指到堆疊頂端，則
      return True;              //傳回True
  else                          //如果不是，則
     return False;              //傳回False
 End
End Procedure
```

【實例】

假設目前在堆疊中**已經有N-1 個資料項**，此時，欲判斷堆疊**是否已滿**，其Stack的變化為何？



說明：①先判斷Top指標是否為N-1，如果是，則

②傳回True，否則傳回False。

6.**判斷堆疊是否是空的**：IsEmpty(Stack)

指用來判斷Top**值為**-1，**若是則傳回**True，**否則傳回**False。

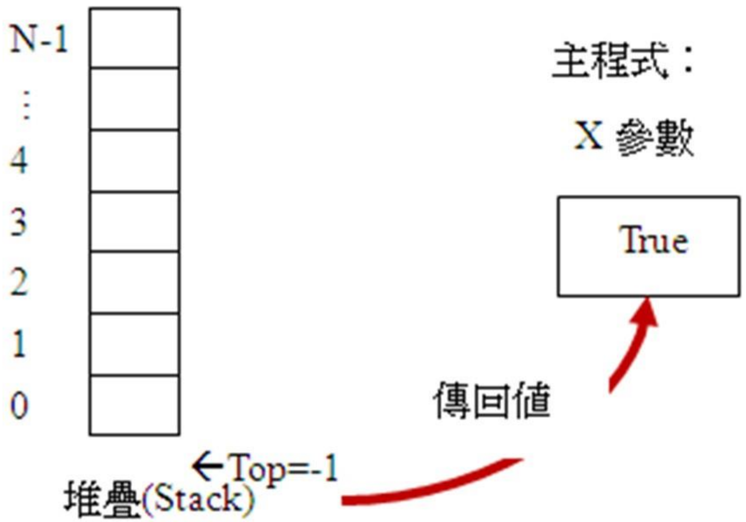
【演算法】

演算法：判斷堆疊是否為空

```
Procedure IsEmpty (Stack)
  Begin
  if (Top=-1)              //如果Top指標為-1，則
        return True;       //傳回True
  else                     //如果不是，則
     return False;         //傳回False
  End
End Procedure
```

【實例】

假設目前在堆疊中沒有任何資料，此時，欲判斷堆疊是否為空，其Stack的變化為何？



說明：①先判斷Top指標是否為-1，如果是，則

②傳回True，否則傳回False。

3-3 堆疊在運算式上的應用

在日常生活中，我們所使用的四則運算都屬於中序(infix order)運算式，亦即運算子位於兩個運算元之間的表示法。

但是，此種表示法電腦並無法直接的處理，因為中序式可能會含有括號，並且運算子可能有不同的優先順序權。

因此，若要使用電腦來處理運算式時，則必須要先將「中序式」轉換成「後序式」(postfix order)。

算術式表示的方式有三種：

一、**中序**(Infix)**表示法**：A+B

二、**前序**(prefix)**表示法**：+AB

三、**後序**(postfix)**表示法**：AB-

一、中序(Infix)表示法

【定義】數學上的表示方式，就是屬於中序式，它是把運算子放在兩個運算元的中間。

【表示式】< 運算元1> < 運算子> < 運算元2>

【例如】A+B。

【缺點】電腦無法一次依序讀取運算式，因運算式可能含有括號，且未定義運算子優先順序。

二、前序 (prefix)表示法

【定義】指將**中序表示法**中的**運算子**和**運算元**重新調整順序，

只是**運算子**的順序是在**運算元前面**。

【表示式】< **運算子**> < 運算元1> <運算元2>

【例如】+AB。

三、後序 (postfix)表示法

【定義】**後序表示法**和**前序表示法**相類似，使得**運算子**放於

運算元後面的表示法。

【表示式】< 運算元1> <運算元2> < **運算子**>

【例如】AB+。

【優點】**不必**使用**括號**，**方便電腦使用**。

3-3.1 運算式的轉換

大部份的**運算式**(expression)都是由**運算元**(operand)與**運算子**(operator)所組成。

【例如】A*B+(C/D)-E

其中： A,B,C,D,E為**運算元**(operand)

+,-,*,/為**運算子**(operator)

【運算原則】

❶括號內先處理

❷優先權較高的運算子先執行

❸同優先權者，則由結合性，來決定是由左而右，還是由右而左執行。

【常見的運算子優先順序】

優先順序	運算子
高	括號：'('　，　')'
	負號：'-'
	乘、除號：'*'　，　'/'
低	加、減號：'+'　，　'-'

一、中序式→前序式

假設有一個**中序式**為：A×B+C×D，欲**轉換**成**前序式**，其步驟如下：

1. 先用**括號**將**優先順序**分出來

((A×B) + (C×D))

2. 將**運算子**移到**左括號右邊**

❶((×AB) + (×CD))

❷(+ (×AB)(×CD))

3. 把**括弧**全部**拿掉**，即為所得。

+ ×AB×CD

二、中序式→後序式

假設有一個**中序式**為：A×B+C×D，欲**轉換**成**後序式**，其步驟如下：

1. 先用**括號**將**優先順序分出來**

((A×B) + (C×D))

2. 將**運算子**移到**右括號左邊**

❶((AB×) + (CD×))

❷((AB×)(CD×) +)

3. 把**括弧**全部**拿掉**，即為所得。

AB×CD× +

3-3.2 中序運算式的表示法及計算

1. 處理**中序運算式**的計算時，基本上，需要使用**兩個堆疊**來存**運算元**和**運算子**。

2. 當**讀取之運算子優先權低於**「運算子堆疊」**頂端運算子**時，則從「**運算子**堆疊」取出**一個運算子**和「**運算元**堆疊」取出**兩個運算元**來**運算**。

【處理步驟】

步驟1. 建立運算元和運算子堆疊，初始為空。

步驟2. 當運算式由左至右尚未讀完時，則依序讀取運算式的一個符號(運算元或運算子)。

步驟3. 若讀取的是運算元，則存入運算元堆疊中。

【處理步驟】<續>

步驟4. 若讀取的是運算子

(1) 若運算子為 "("，放入堆疊。

因為：中序表示式，左括號在堆疊中的優先順序最小，亦即任何運算子都大於它的優先權。

(2) 若運算子為 ")"，依序輸出堆疊中的運算子，直到取出 "(" 為止。

(3)若運算子堆疊為空，則存入運算子堆疊。

(4)若運算子堆疊非空，則與頂端之運算子比較優先權，若較堆疊中的高則存入堆疊中。若較低或等於時，則從運算子堆疊取出一運算子並從運算元堆疊取出所需運算元，運算後將結果存回運算元堆疊。然後將剛剛讀取之運算子存入運算子堆疊。若運算式尚未讀完時，則回到步驟(2)。

【處理步驟】<續>

步驟5. 中序運算式讀完之後，若運算子堆疊非空，則從運算子堆疊取出一運算子並從運算元堆疊取出所需運算元，運算後將結果存回運算元堆疊。重覆此步驟，直到運算子堆疊為空的。

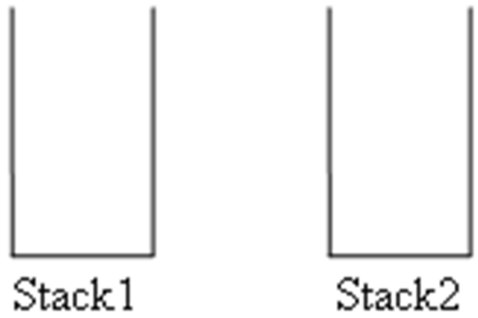
步驟6. 運算元堆疊的最後內容即為運算式之計算結果。

【舉例】中序運算式為：A+B*C-D

【解答】

假設：運算元堆疊設為Stack1，運算子堆疊設為Stack2

步驟1: 建立運算元和運算子堆疊，並設初始為空的

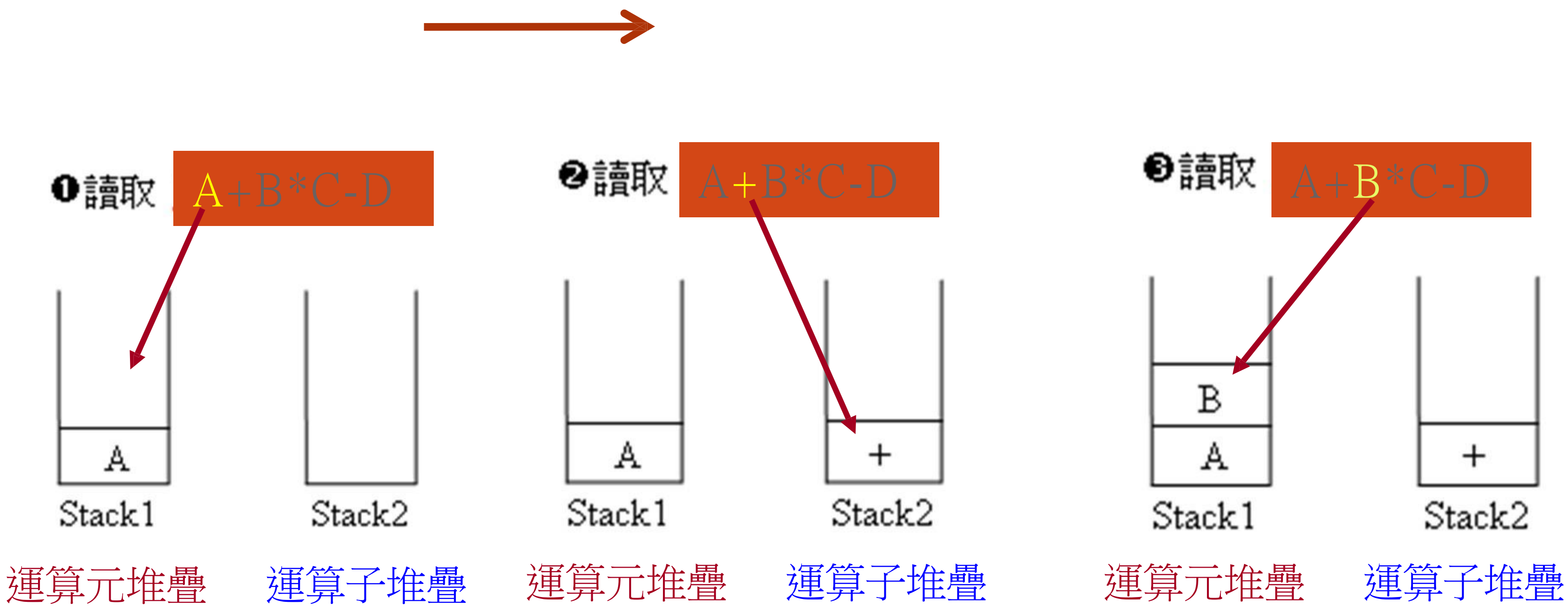


運算元堆疊　運算子堆疊

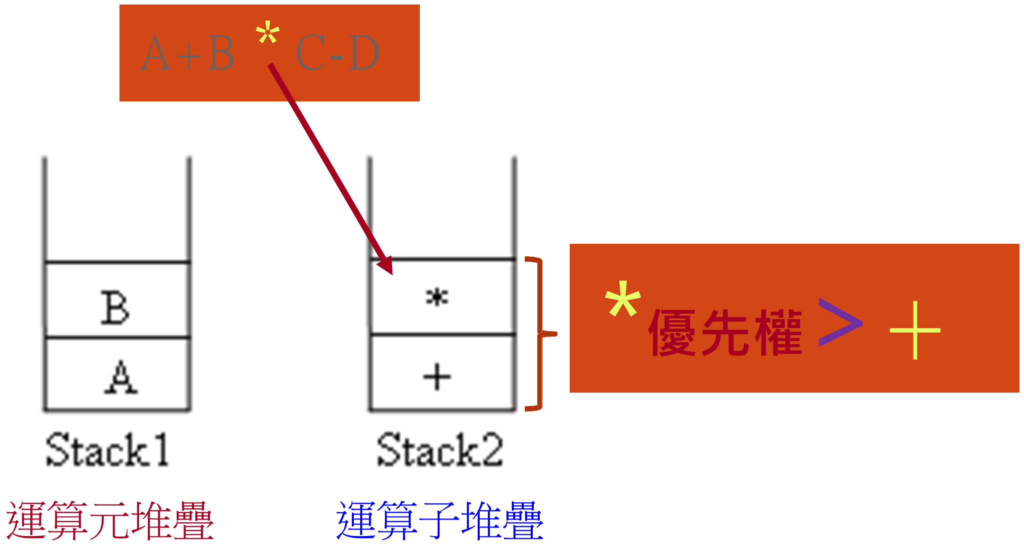
步驟2: **從左到右**讀取**運算式**，讀到**運算元**則置入**運算元堆疊**(Stack1)；

若讀到**運算子**則置入**運算子堆疊**(Stack2)。

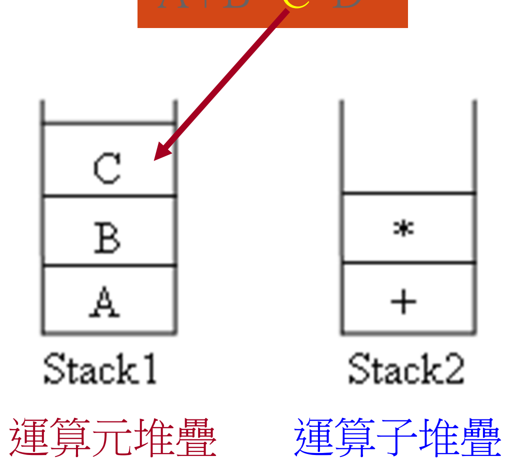
運算式：A+B*C-D



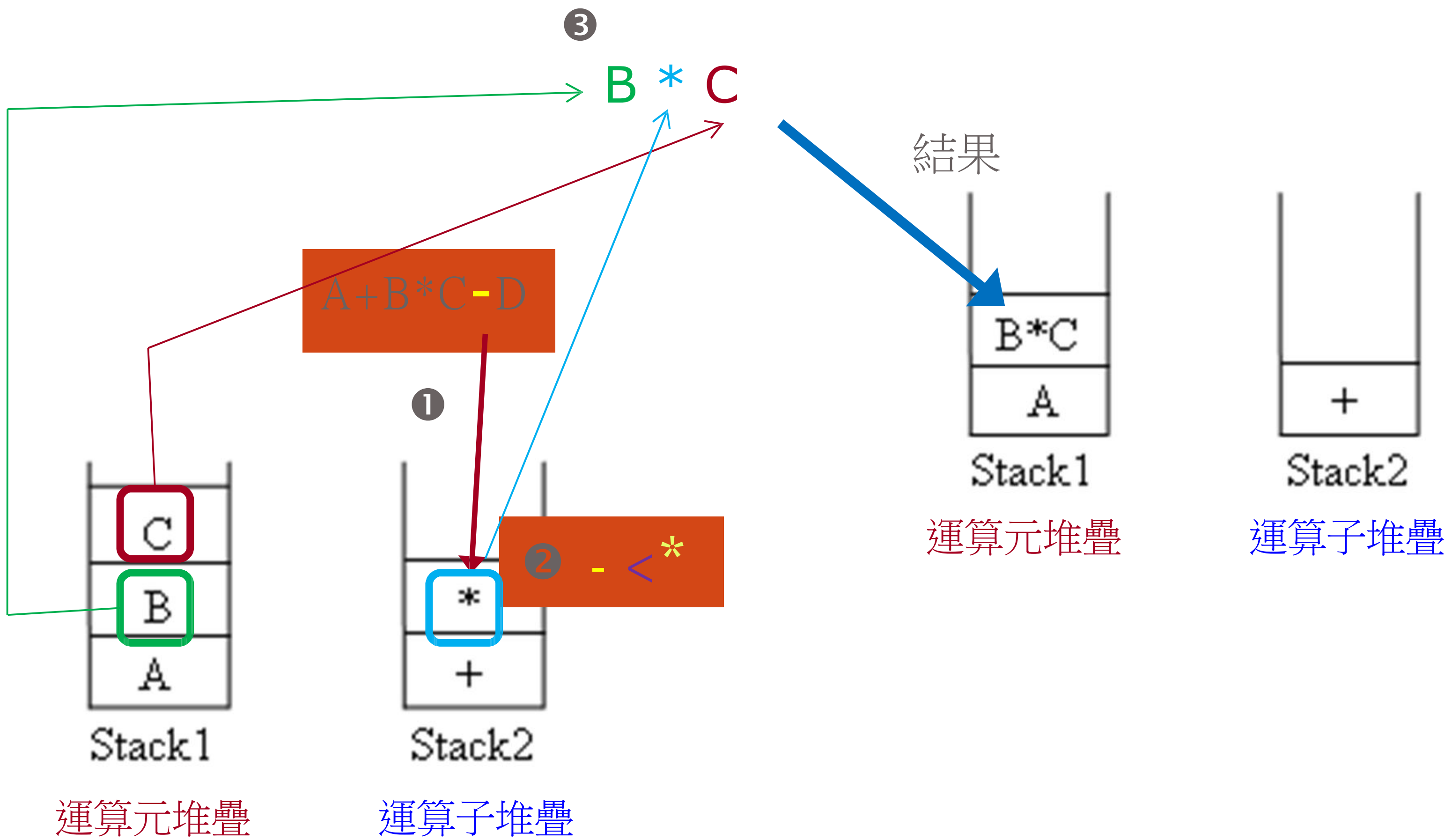
步驟3: 讀取“ * ” 時，由於“ * ” 的優先權高於Stack2的top 頂端運算子 “ + ” ，故將“ * ” 存入運算子堆疊。



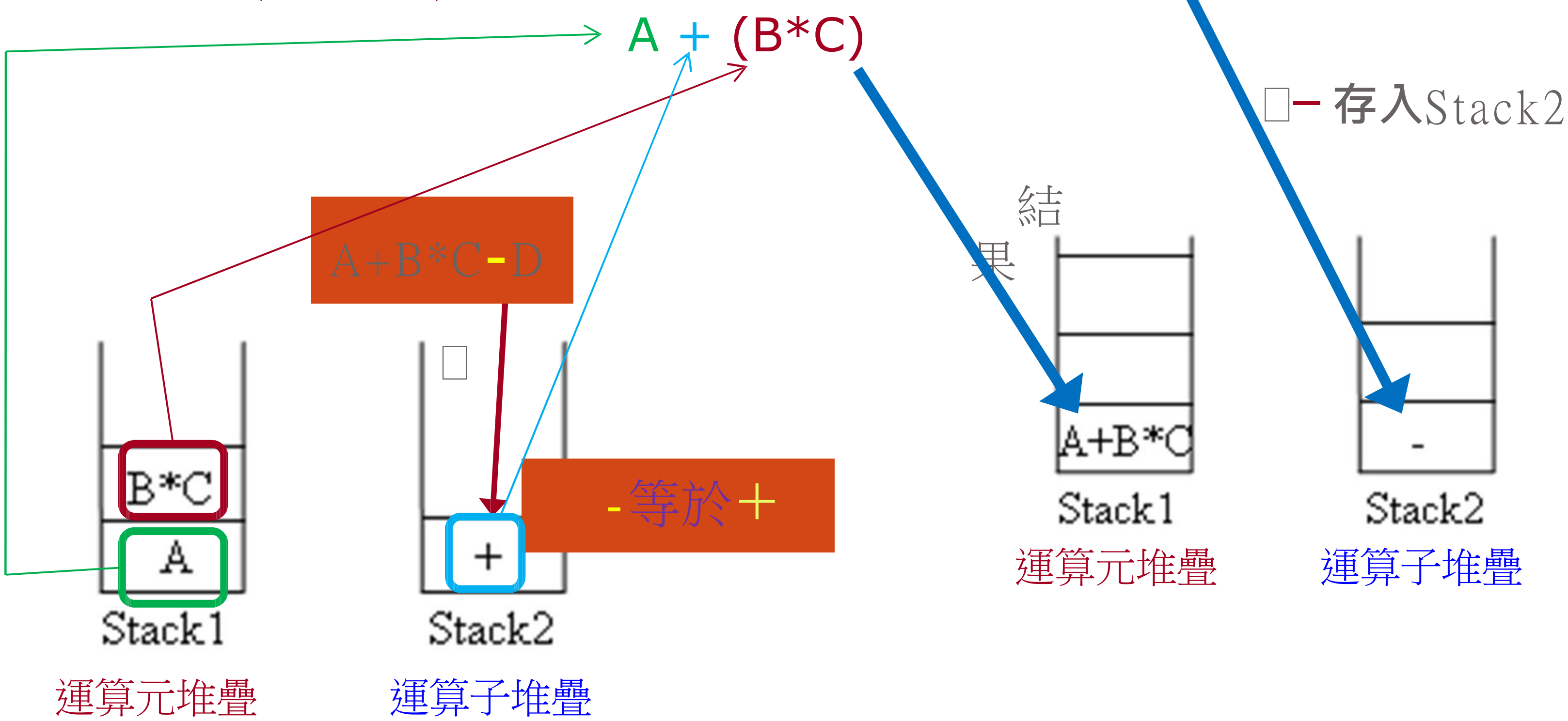
步驟4: 讀取 A+B*C-D



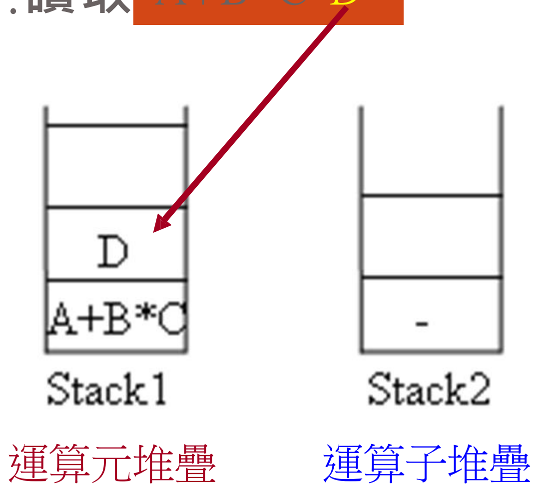
步驟5: 讀取" - "時，因為" - "的**優先權低**於Stack2的top **頂端運算子** **" * "** ，故**取出**" * "與**兩個運算元**進行**計算**，並將**結果**存回**運算元堆疊**(Stack1)中。



步驟6: 因為" - "時，優先權**等於**Stack2的頂端運算子" + " ，故**取出" + "**與**兩個運算元**來進行**計算**，並將**結果**存回**運算元堆疊(Stack1)**中。將剛才**未存入堆疊**的" - "存入**運算子堆疊(Stack2)**。

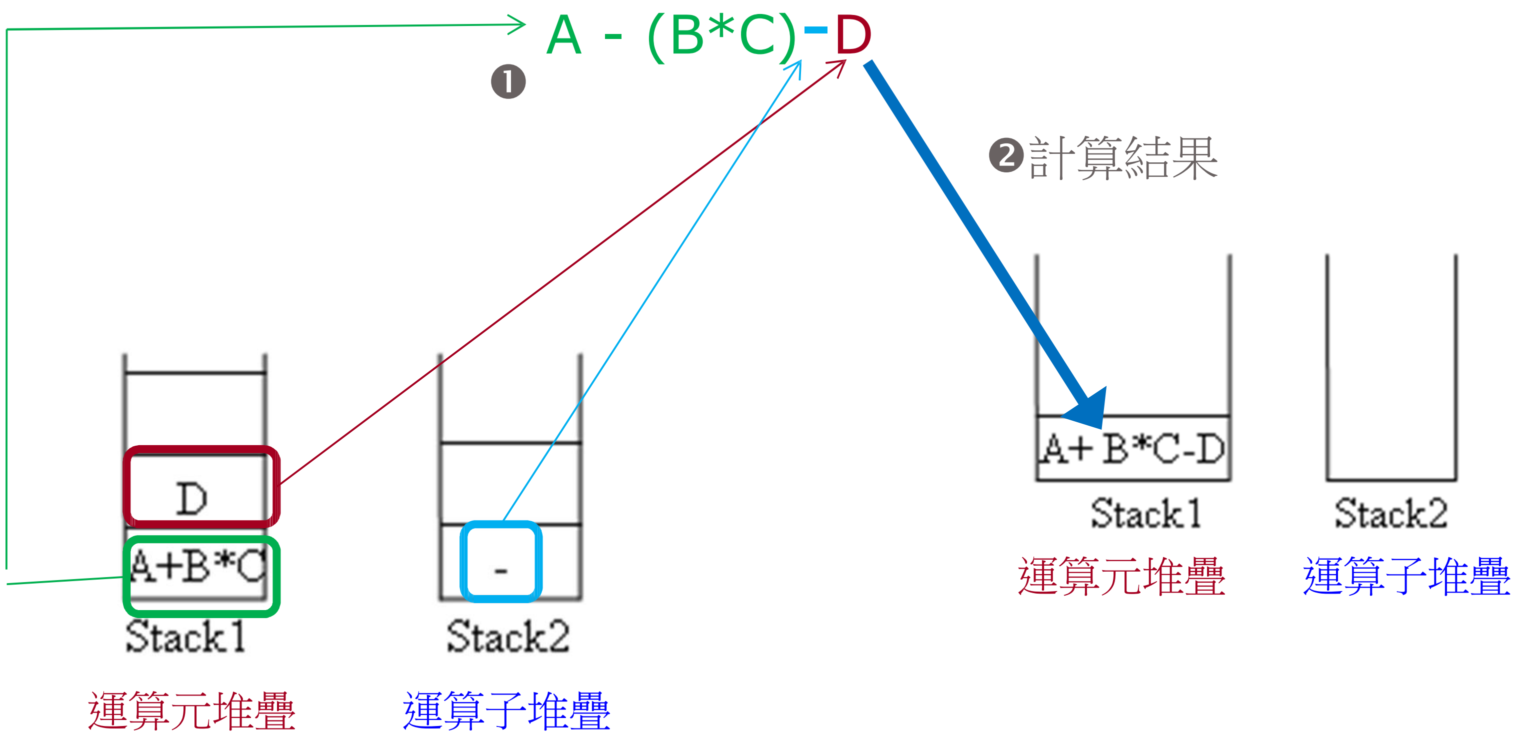


步驟7:讀取 A+B*C-D



步驟8: 取出運算子" - " 及運算元"A+B*C" 及"D" 進行計算，

結果存回運算元堆疊(Stack1)



3-3.3 中序表示法轉換成前序表示法

由**中序表示法轉換**成**前序表示法**的方法有**兩種：**

1. 加括號去除法：這種方法是**人類使用的方法**，主要是**應付於考試，**
想快速得到前序表示法時使用。

2. 堆疊處理法：「**由右而左**」**掃描**資料，依據資料是**運算元**或**運算子**
作不同的處理，而**運算子**還要考慮其**優先次序**。

一、加括號去除法

【例如】假設有一個**中序式**為：**A+B*(C-D)**，欲**轉換**成**前序式**，其步驟如下：

1. 先用**括號**將**優先順序**分出來

   (A+(B*(C-D)))

2. 將**運算子**移到**左括號右邊**

   ❶(A+(*B(-CD)))

   ❷(+A(*B(-CD)))

3. 把**括弧**全部**拿掉**，即為所得。

   +A*B-CD

二、堆疊處理法

步驟1. 由右至左依序取得資料di(data item)。

步驟2. 如果di是運算元，則直接輸出。

步驟3. 如果di是運算子（含左右括號），則：

(1) 如果di=")"，放入堆疊。

(2) 如果di="("，依次輸出堆疊中的運算子，直到取出")"為止。

(3) 如果di不是“)”或“(”，則與堆疊頂點的運算子ds(data of stack)作優先順序比較：

❶當di較ds優先時，則di放入堆疊，迴圈輸出堆疊資料，直到優先次序相等。

❷當di不較ds優先或相等時，則ds輸出，di放入堆疊。

步驟4. 如果運算式已讀取完成，而堆疊中尚有運算子時，依序由頂端輸出。

步驟5. 反轉輸出的字串

【實例】A+B*(C-D)

請利用**堆疊處理法**將**中序式**為：A+B*(C-D)， 欲**轉換**成**前序式**

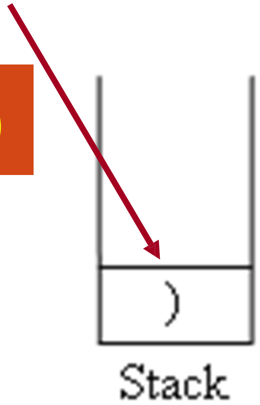
A+B*(C-D)

由右至左讀取資料

其步驟如下：

(1)**讀取**：‘)’　> Stack頂端的運算子**優先權**

A+B*(C-D)



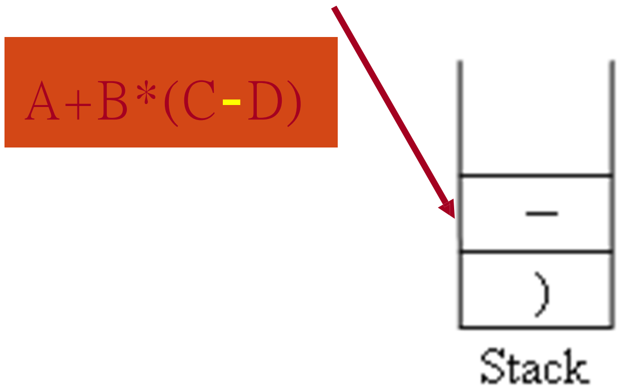
(2)**讀取**：‘D’

A+B*(C-D)

輸出：D

目前輸出的字串：D

(3)**讀取：' - '** > Stack頂端的運算子**優先權**



註：中序轉成**前序**時，**右括號**在堆疊中的**優先順序最小**，

亦即**任何運算子**都**大於**它的**優先權**。

(4)讀取：'C' → 輸出：C

A+B*(C-D)

目前輸出的字串：DC

(5)讀取：'(' 後，依序輸出(POP)堆疊中的運算子，直到取出") "為止

A+B*(C-D)

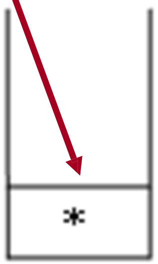
輸出： -

目前輸出的字串：DC -

(註：左右括號不會輸出顯示)

(6)**讀取**：' * ' > Stack頂端的運算子**優先權**

A+B*(C-D)



Stack

(7)**讀取**：'B'

輸出：B

目前輸出的字串：DC - B

A+B*(C-D)

(8)讀取：'+' < Stack頂端的運算子優先權，則' * ' 輸出，
' + ' 放入堆疊

A+B*(C-D)

輸出：*
目前輸出的字串：DC - B*

(9)讀取：'A' → 輸出：A
目前輸出的字串：DC - B*A

A+B*(C-D)

(10)**輸出剩餘**的Stack中的資料(**運算式已經讀取完成**，而堆疊中

尚有運算子時, 依序由**頂端輸出**)

輸出：+

目前輸出的字串：DC - B*A+

反轉輸出

(11)**反轉輸出**的字串：+A*B - CD

3-3.4 中序表示法轉換成後序表示法

由**中序表示法轉換**成**後序表示法**的方法有**兩種**：

1. **加括號去除法**：這種方法是**人類使用的方法**，主要是**應付於考試**，想**快速得到後序表示法**時使用。
2. **堆疊處理法**：「**由左而右**」掃描資料，依據資料是**運算元**或**運算子**作不同的處理，**運算子**還要考慮其**優先次序**。

一、加括號去除法

【例如】假設有一個**中序式**為：A+B*(C-D)，欲**轉換**成**後序式**，

其步驟如下：

1. 先用**括號**將**優先順序**分出來

   (A+(B*(C-D)))

2. 將**運算子**移到**右括號左邊**

   ❶(A+(B(CD-)*))

   ❷(A(B(CD-)*)+)

3. 把**括弧**全部**拿掉**，即為所得。

   ABCD-*+

二、堆疊處理法

步驟1. 由左至右依序取得資料di(data item)。

步驟2. 如果di是運算元，則直接輸出。

步驟3. 如果di是運算子(包含左右括號)，則：

(1) 如果di= "(", 放入堆疊。

(2) 如果di= ")"，依序輸出堆疊中的運算子，直到取出 "(" 為止。

(3) 如果di不是"("或 ")"，則與堆疊頂點的運算子ds(data of stack)作優先順序比較：

❶當di較ds優先時，則di放入堆疊。

❷當di不較ds優先或相等時，則ds輸出，di放入堆疊。

步驟4.如果運算式已經讀取完成，而堆疊中尚有運算子時，依序由頂端輸出

【實例】A+B*(C-D)

請利用**堆疊處理法**將**中序式**為：A+B*(C-D)，欲**轉換**成**後序式**，其步驟如下：

A+B*(C-D)

由左至右讀取資料

(1) **讀取**：'A' → 輸出：A

目前輸出的字串：A

A+B*(C-D)

(2) **讀取**：'+' > Stack**頂端的運算子優先權**

A+B*(C-D)

Stack

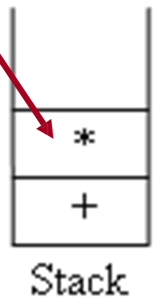
(3) 讀取：'B' → 輸出：B

目前輸出的字串：AB

A+B*(C-D)

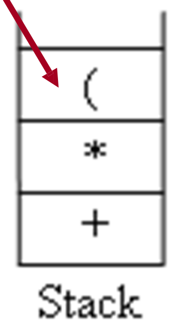
(4) 讀取：'*' > Stack頂端的運算子優先權

A+B*(C-D)



(5) **讀取**：'(' 放到Stack中

A+B*(C-D)



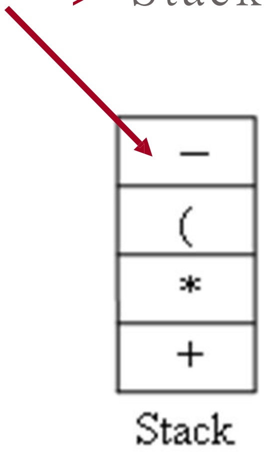
(6) **讀取**：'C' → 輸出：C

目前輸出的字串：ABC

A+B*(C-D)

(7) **讀取**：'-'> Stack**頂端的運算子優先權**

A+B*(C-D)



註：**中序轉成後序**時，**左括號**在堆疊中的**優先順序最小**，

亦即**任何運算子**都**大於**它的**優先權**。

(8) **讀取**：'D' → 輸出：D

目前輸出的字串：ABCD

A+B*(C-D)

(9) **讀取**：')' 後，**依序輸出**(POP)堆疊中的**運算子**，**直到取出"(" 為止**

A+B*(C-D)

輸出： -

目前輸出的字串：ABCD -

(**註**：**左右括號**不會輸出顯示)

(10)輸出剩餘的Stack中的資料(運算式已經讀取完成，而堆疊中尚有運算子時，依序由頂端輸出)

輸出：*+

最後輸出的字串：ABCD - *+

3-3.5 前序表示法轉換成中序表示法

在上面介紹的方法是將「**中序**」轉換成「**前序與後序**」，但是如何將「**前序**」**轉換**成「**中序**」呢？

我們一樣可以利用前面提到的「**加括號去除法**」與「**堆疊法**」來進行**轉換**。

但是，轉換方式略**有一點點不同，其說明如下**。

一、加括號去除法

【例如】請將**前序式**：+A*B-CD，**轉換**成**中序式**。

【作法】

1. 依「**運算子**+ **運算元**1 + **運算元**2」的原則括號

   (+ A (* B (- C D)))

2. 將**左邊**的運算子**移到兩個運算元**之中

   (+ A (* B (- C D)))

   ∴(A + (B *(C- D)))

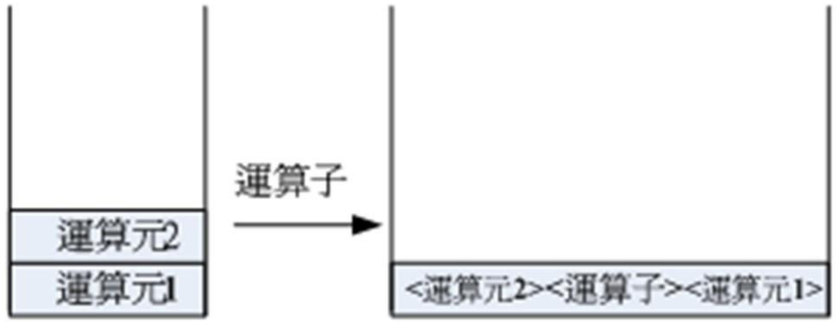
3. **拿掉未具優先權的括弧**

   A+B*(C-D)

二、堆疊處理法

以堆疊法來處理**前序轉換**成**中序**時，必須要**遵守以下的原則**：

1. **由右至左**依序取得**資料**di
2. 如果di是**運算元**，則**放入堆疊中**。
3. 如果di是**運算子**，則從堆疊中**取出兩個運算元**，依照中序式的結合方式(<**運算元**2><**運算子**><**運算元**1>)後，再將**結果**放入**堆疊中**。如下圖所示：



前序轉成中序：<運算元 2><運算子><運算元 1>

【實例】

假設有一個**前序式**：+A*B - CD ，**轉換**成**中序式**

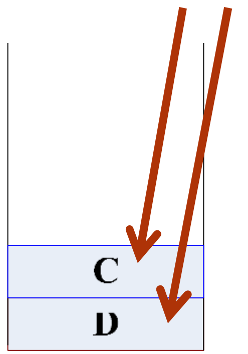
【作法】

1.**由右而左**依序取得**資料**di

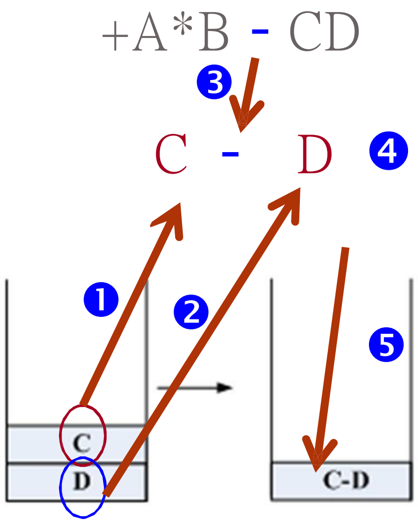
+A*B - CD

2.如果di是**運算元**，則**放入堆疊**中。

+A*B - CD



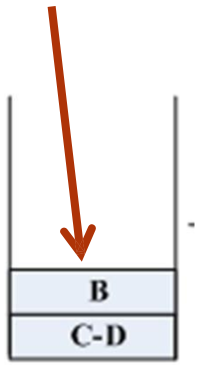
3.如果di是運算子，則從堆疊中取出兩個運算元，依照中序式的結合方式(<運算元2><運算子><運算元1>)後，再將結果放入堆疊中。



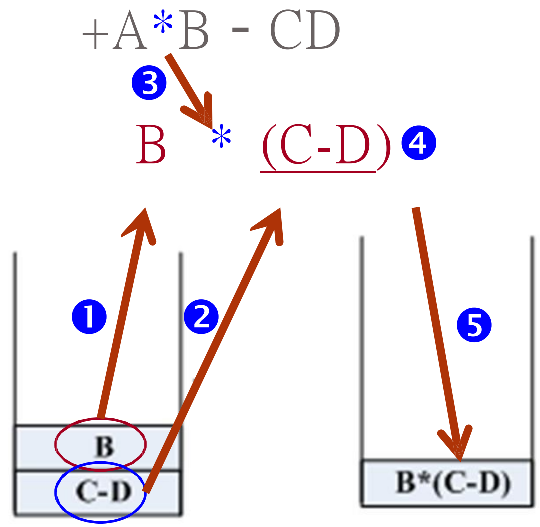
說明：取出❶C與❷D，再結合為❸C-D，再將❹結果放入❺堆疊中。

4.如果di是**運算元**，則**放入堆疊**中。

+A*B - CD



5.如果di是**運算子**，則從堆疊中**取出兩個運算元**，依照**中序式的結合方式**(<**運算元**2><**運算子**><**運算元**1>)後，再將**結果**放入**堆疊中**。

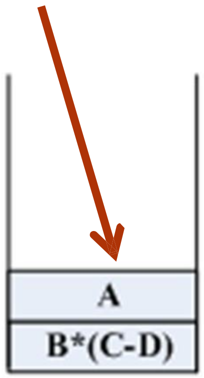


說明：**取出**❶B**與**❷(C-D)，再**結合為**❸B*(C-D)，

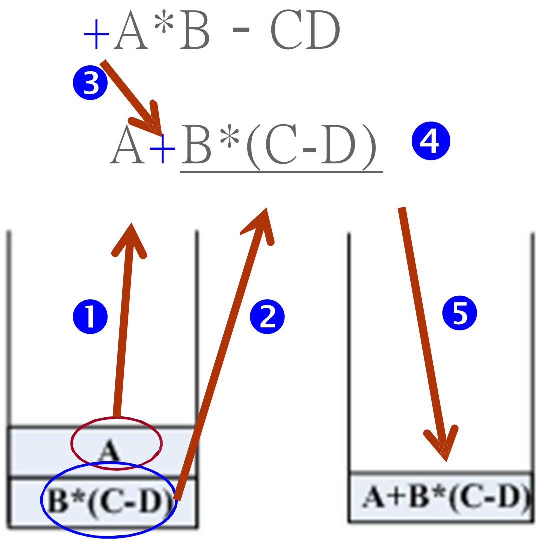
再將❹**結果**放入❺**堆疊中**。

6.如果di是**運算元**，則**放入堆疊**中。

+A*B - CD



7.如果di是**運算子**，則從堆疊中**取出兩個運算元**，依照**中序式的結合方式**(<**運算元**2><**運算子**><**運算元**1>)後，再將**結果**放入**堆疊中**。



說明：**取出**❶A與❷B*(C-D)，再**結合為**❸A+B*(C-D)，

再將❹**結果**放入❺**堆疊中**。

3-3.6 後序表示法轉換成中序表示法

一、加括號去除法

【例如】請將**後序式**：ABCD-*+ **轉換**成**中序式**。

【作法】

1. 依「**運算元**1＋**運算元**2＋**運算子**」的原則括號
   (A (B (C D -) *) +)
2. 將**右邊的運算子移到兩個運算元之中**

(A (B (C D -) *) +)

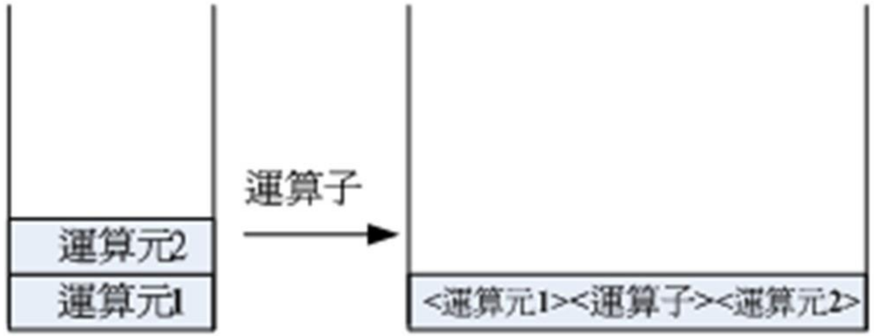
∴(A+(B *(C- D)))

3.**拿掉末具優先權**的括弧
   A+B*(C-D)

二、堆疊處理法

以堆疊法來處理**後序轉換**成**中序**時，必須要**遵則以下的原則**：

1. **由左至右**依序取得**資料**di。
2. 如果di是**運算元**，則**放入堆疊中**。
3. 如果di是**運算子**，則從堆疊中**取出兩個運算元**，依照**中序式**的**結合**方式(<**運算元**1><**運算子**><**運算元**2>)後，再將**結果**放入**堆疊中**。如下圖所示：



後序轉成中序：<運算元 1><運算子><運算元 2>

【實例】

假設有一個**後序式**：ABCD-*+，**轉換**成**中序式**

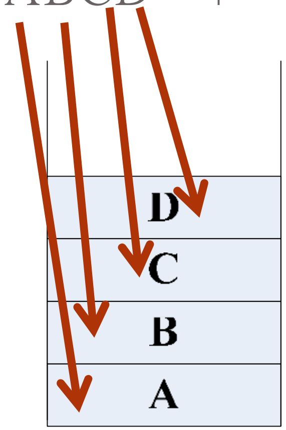
【作法】

1. **由左而右**依序取得**資料**di

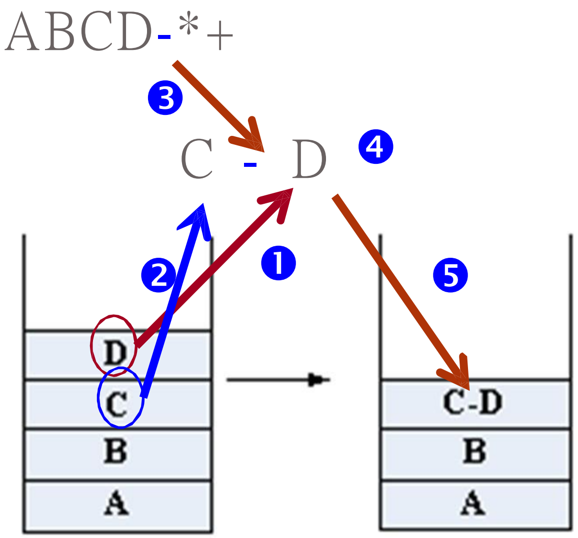
ABCD-*+

2. 如果di是**運算元**，則**放入堆疊中**。

ABCD-*+

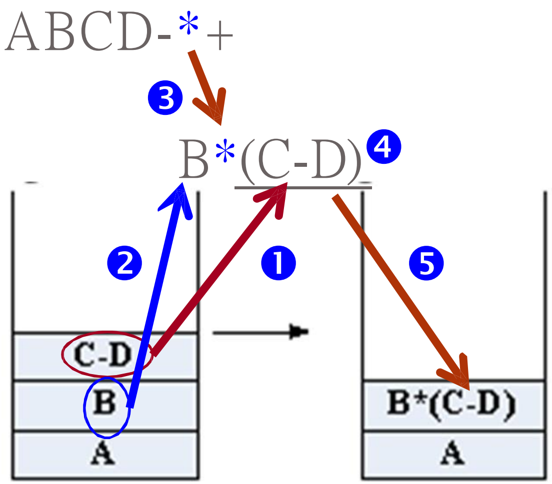


3.如果di是運算子，則從堆疊中取出兩個運算元，依照中序式的結合方式(<運算元1><運算子><運算元2>)後，再將結果放入堆疊中。



說明：取出❶D與❷C，再結合為❸C-D，再將❹結果放入❺堆疊中。

4.如果di是運算子，則從堆疊中取出兩個運算元，依照中序式的結合方式(<運算元1><運算子><運算元2>)後，再將結果放入堆疊中。



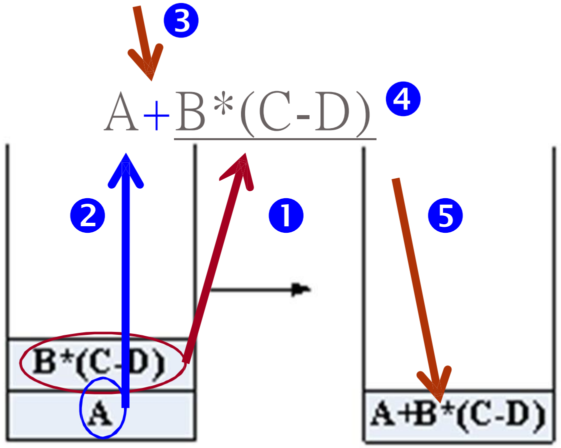
說明：取出❶(C-D)與❷B，再結合為❸B*(C-D)，

再將❹結果放入❺堆疊中。

5.如果di是**運算子**，則從堆疊中**取出兩個運算元**，依照**中序式**的**結合**

方式(<**運算元**1><**運算子**><**運算元**2>)後，再將**結果**放入**堆疊中**。

ABCD-*+

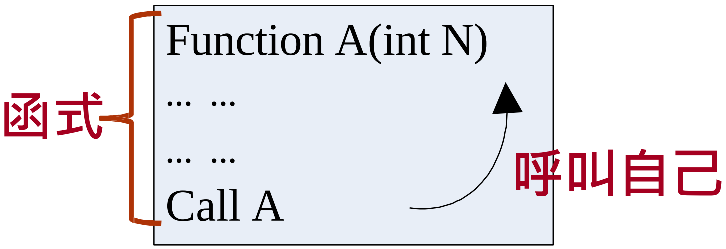


說明：取出❶B*(C-D)與❷A，再結合為❸A+B*(C-D)，

再將❹**結果**放入❺**堆疊中**。

3-4 遞迴(Recursion)

何謂「遞迴(Recursion)」是指**函式本身**又可以**呼叫自己**的**副程式**。



我們在撰寫程式時，如果**適時**的**使用遞迴方式**，將可以使得程式變得**比較簡潔**；但是在撰寫時必須要**先了解題意**，並且要非常**謹慎小心**使用，否則很**容易產生無窮迴圈**或**無法預期的錯誤**。

如果沒有設定終止條件

例如：我們設計一個遞迴程式來計算n!時，如果沒有設定「終止條件」時，將會無限制地呼叫自己，因此就會造成無窮迴圈現象。

遞迴程式

```
Procedure fact(int n)
begin
        int result;
        if(n == 1) then result = 1;
        else
           result = n * fact(n-1);
        return result ;
end
```

不會產生無窮迴圈

有設定終止條件

在目前的**程式語言**中，**並非**每一種程式語言**都具有遞迴呼叫**的功能，例如：

(1)**不具遞迴呼叫**的程式語言：BASIC、FORTRAN、COBOL等。

(2)**具遞迴呼叫**的程式語言：C、C++、PASCAL等。

遞迴函數

　　如果將程式模組化為一支獨立的函數，如果該函數可以反覆地自己呼叫自己，因此我們稱這個函數為「遞迴函數」。

在遞迴函數中，最典型的例子就是計算n 階層的程式。

一、數學上：n階層的概念如下：

題目：$n!=n\times(n-1)\times(n-2)\times(n-3)\times\ldots\times1$	舉例：$5!=5\times4\times3\times\ldots\times1$
$n! = n\times(n-1)!$ 第一次遞迴呼叫 n=5 $=n\times[(n-1)\times(n-2)!]$ $=n\times(n-1)\times[(n-2)\times(n-3)!]$ $\vdots$ $\therefore n!=n\times(n-1)\times(n-2)\times(n-3)\times\ldots\times1$	$5! =5\times4!$ 第二次遞迴呼叫 n=4 $=5\times(4\times3!)$ 第三次遞迴呼叫n=3 $=5\times4\times(3\times2!)$ 第四次遞迴呼叫 n=2 $=5\times4\times3\times(2\times1!)$ 第五次遞迴呼叫n=1 $=5\times4\times3\times2\times(1!)$ $\because 1!=1$ $\therefore 5!= 5\times4\times3\times2\times1$ 停止遞迴呼叫

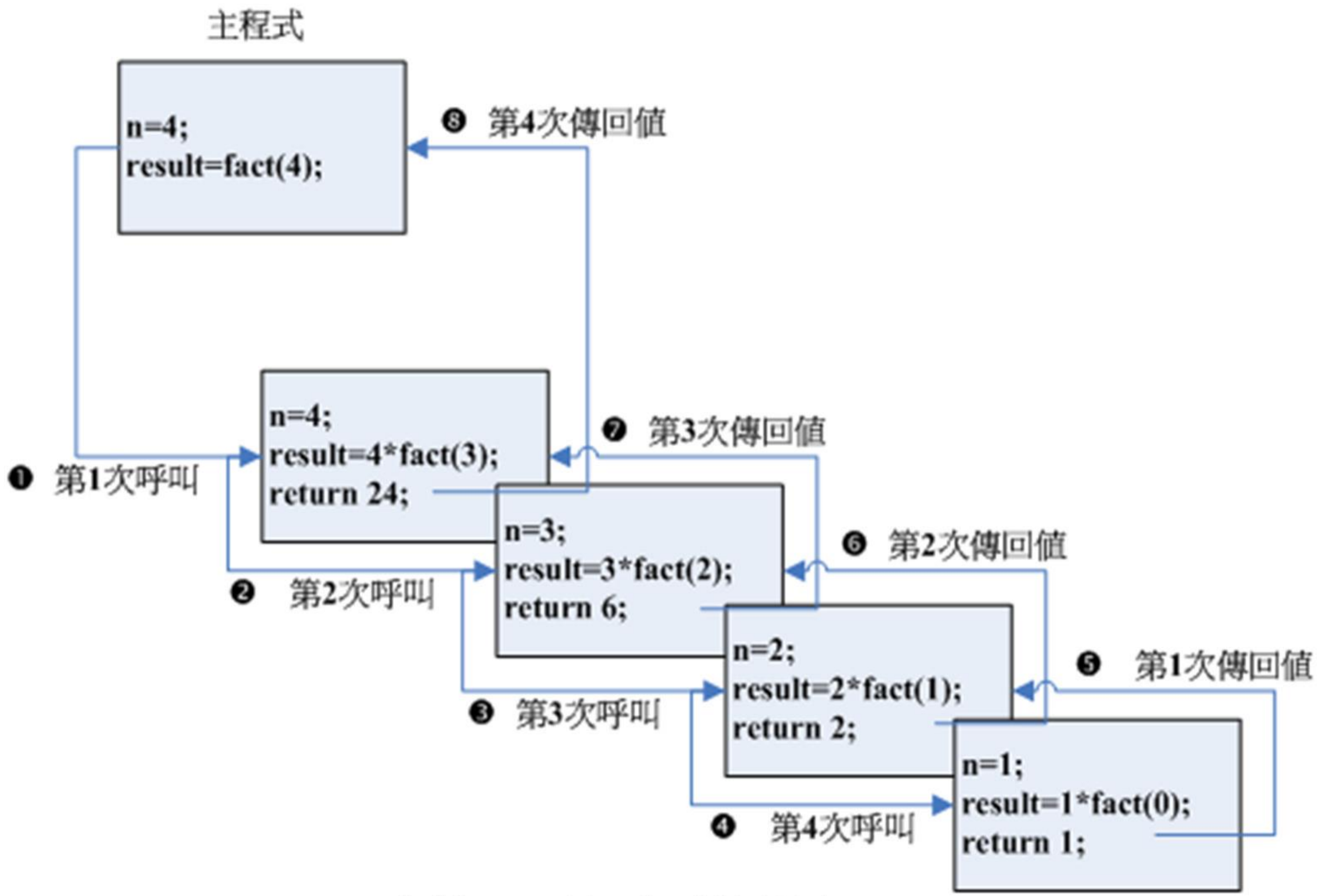
說明：在撰寫一個n階層的遞迴函數程式時，則該函數將具備兩個主要特徵：

1. 該遞迴函數可以自己反覆地呼叫自己
   (1)第一次呼叫時的參數為n，
   (2)第二次呼叫時的參數為n-1，
   (3)第三次呼叫時的參數為n-2，...，
   (4)參數的值會逐次遞減。
2. 當參數值等於1時，必須停止遞迴呼叫。

二、演算法上：n階層的概念如下：

```
Procedure fact(int n)
 Begin
   int Result;
   if(n == 1) then result = 1;                    //終值設定為1，以防止產
生無窮迴圈
   else                                           //如果否是終值，則
        result = n * fact(n-1);                   //遞迴呼叫，並且每次要更
新值，即每次減1
   return Result ;
 End
End Procedure
```

三、以圖解來說明：遞迴函數呼叫的過程



遞迴函數呼叫的過程

3-4.1 遞迴函數種類

一般而言**遞迴函數**可以**分為兩種**：

1. **直接(Direct)遞迴**：程序自己直接呼叫自己。

```
Function A(int N)
… …
… …
Call A
```

2. **間接(Indirect)遞迴**

是指兩個以上函數，彼此呼叫對方，形成迴路。

```
Function A(int N)
……
……
Call B
```

```
Function B(int N)
……
……
Call A
```

說明：在A遞迴函數內呼叫B遞迴函數，並且在B遞迴函數內呼叫A遞迴函數。

3-4.2 遞迴與非遞迴的比較

(1)**遞迴**：是指**函數本身**又可以**呼叫自己**的**副程式**。

(2)**非遞迴**：是指**函數本身沒有呼叫自己**的**程式**。

【例如】設計n!的**遞迴**與**非遞迴**的比較，如下所示：

遞迴(Recursion)	非遞迴(Non-Recursion)

int Sum(int n)
{
    int Result;
    if(n == 1)
        Result = 1;
    else
        Result = n * Sum(n-1)
    return Result ;
}
	int Sum(int n)
	{
	    int i, Result = 1;
	    for (i=n; i<=1; i--)
	        Result *= i;
	    return Result;
	}

【遞迴函數的優、缺點】

	遞迴 (Recursion)	非遞迴 (Non-Recursion; Iterative)
優點	❶程式較簡潔明確 ❷節省記憶體空間 ❸表達力較強 ❹區域變數與暫存變數較少	❶較節省執行的時間 ❷不需額外的**Stack**空間
缺點	❶執行時參數的存取較費時間 ❷需額外堆疊**(Stack)**空間支援	❶程式較長 ❷浪費記憶體空間 ❸表達力較弱 ❹區域變數與暫存變數較多

3-5 遞迴的應用

遞迴在**實務上**的應用非常的廣，例如：計算n **階乘**(n!)、**費氏數列**、**河內塔**的**圓盤搬移過程**、求兩個正整數的**最大公因數**等。

3-5.1 n階乘(n!)

【定義】n ! = n×(n-1)×(n-2)×(n-3)×⋯×1

【假設】階乘的**公式**如下：

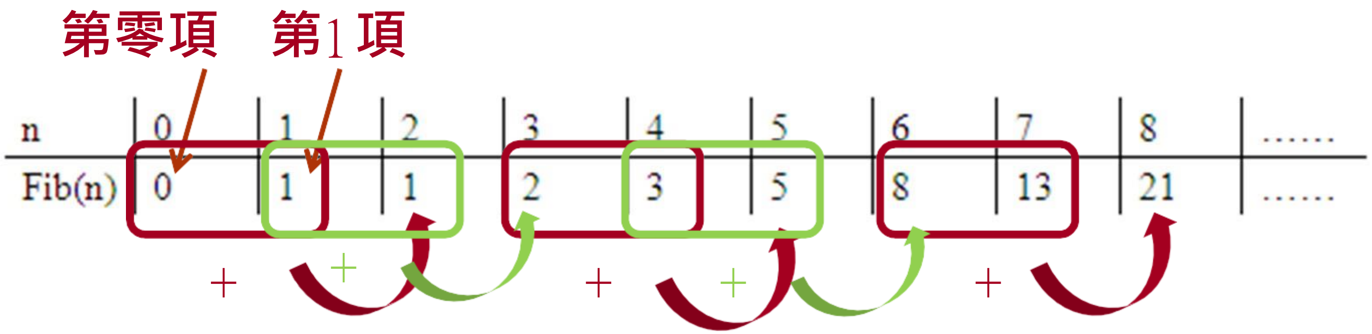
$$f(n)=\begin{cases}1 & ,\ if \quad n=1\\ n\times f(n-1) & ,\ if \quad n\geqq 2\end{cases}$$

【演算法】

```
Procedure fact(int n)
 Begin
   int Result;
   if(n == 1) then result = 1;        //終值設定為1，以防止產生無窮迴圈
   else                               //如果否是終值，則
        result = n * fact(n-1);       //遞迴呼叫，並且每次要更新值，即每次減1
   return Result ;
 End
End Procedure
```

3-5.2 費氏數列

【定義】某一數列的**第零項**為0，**第1 項**為1，其他每一個數列中項目的值是由本身**前面兩項的值之和**。



【假設】級數Factorial的**公式**如下：

$$
Fib(n)=\begin{cases} 0 & ,\text{if} \quad n=0 \\ 1 & ,\text{if} \quad n=1 \\ Fib(n-1)+Fib(n-2) & ,\text{if} \quad n\geq 2 \end{cases}
$$

【演算法】

```
Procedure Fib(int n)
 Begin
   if(n=0) return 0;                        //第零項為0
   if(n=1) return 1;                        //第1項為1
    if(n>=2) return Fib(n-1)+Fib(n-2);   //其他某一數為其前二個數的和
 End
End Procedure
```

3-5.3 最大公因數

【定義】利用**尤拉**的**輾轉相除法**，求兩數之<u>**最大公因數**</u>演算法。

即利用**兩數反覆相除**，**直到「餘數」為0**，則其「**除數**」

即為**最大公因數**。

【假設】最大公因數的**公式**如下：

$$GCD(A,B)=\begin{cases} B & \text{, 若 } A\%B=0 \\ GCD(B, A\%B) & \text{, 若其他情況} \end{cases}$$

其中%代表取餘數

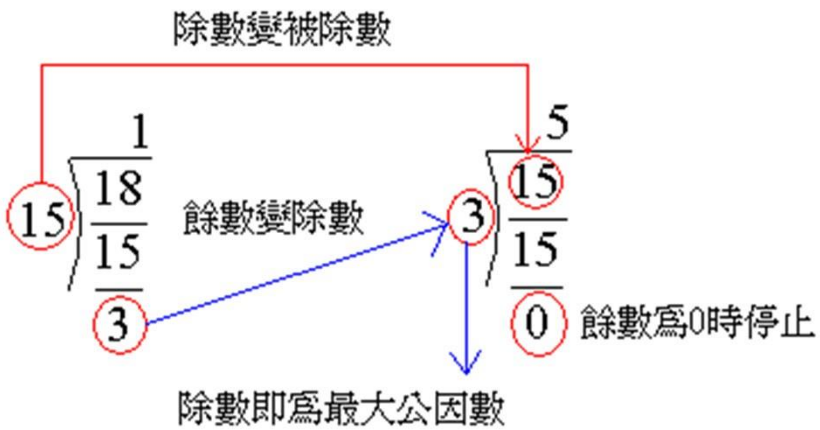
【演算法】

```
Procedure GCD (int a,int b)
 Begin
  c = a % b;                //a除b取餘數
  if (c == 0)               //判斷餘數是否為0，如果是，則
   return b;                //將其除數即為最大公因數
  else                      //如果不是，則
   return GCD(b, c);        //函式自己又可以呼叫自己
 End
End Procedure
```

【舉例】

以下範例說明**輾轉相除法過程**：以18及15為例，必須要利用「輾轉相除法」，求**最大公因數**。

其概念如下所示：



3-5.4 河內塔問題

【問題】假設有A、B、C三根柱子，A柱子上有3個直徑不同的中空盤子，由大而小疊放在一起，如下圖(3個盤子)所示。

試問最少需要搬多少次，方能將這些盤子從A柱搬到C柱？

【規則】1.一次只能移動一個盤子。

2.在搬移過程中，小的盤子不能放在大的盤子下方。

原始狀態	完成後之狀態
	

【演算法】

演算法：河內塔問題

```
Procedure  Towers(n, p, q, r)
 Begin
  if (n==1)                          //如果只有一個盤子
        printf(p ″to″ r);           //只須將一個盤子從p柱搬到r柱
    else                             //如果有二個或二個以上盤子，則
   {
      Towers(n-1, p, r, q);         //先將上面n-1個盤子從p柱搬到q柱
      printf(p ″to″ r);             //再將最大的盤子從p柱搬到r柱
      Towers(n-1, q, p, r);         //最後，再將剩下的n-1個盤子從q柱搬到r柱
    }

 End
End Procedure
```

【河內塔的運作過程】