3

堆疊和佇列

「堆疊」(stack) 在日常生活中經常可見：各位在自助餐廳用餐時，皆自餐盤堆疊的上方取用餐盤；上下電梯（一個出入口者）時，好的習慣是走入電梯即靠內，方便後來者進入，而離開電梯時，則以後進入者先行離去；疊羅漢完成後，爲了安全讓後加入者先離去；抽取面紙時上方者可先被取出（在裝盒時，它們是較後被放入者）；倉庫貯貨、堆高積木、印表機紙匣內紙的使用、自藥罐內取藥（茶罐中取茶）、…；這些都屬於堆疊，他們皆擁有的共同特性就是：先行進入者將較後離去—先進後出（first in last out, 或簡稱 FILO）、或後來進入者可先行離去—後進先出（last in first out, 或簡稱 LIFO）。

「佇列」(queue) 的實際例子也很多：排隊買票、看診、候車、進場（舉凡排隊皆然）、單線道或單行道上進出的汽車、飲水機旁由上方加入由下方取出的紙杯、…；這些例子的共同特性在於：先行進入者將率先離去—先進先出（first in first out, 或簡稱 FIFO）。

當電腦運用在解決有堆疊或佇列現象的問題時，我們應能夠將這些抽象概念具象程式化。本章的目的即介紹這兩種資料結構的使用技巧與應用時機，各位將會發現許多問題都與堆疊或佇列息息相關，不利用堆疊或佇列可能根本無法順利解決。

3.1 堆疊

堆疊是一個有序串列 (ordered list)，而且只在一個特定的出入口，進行資料的增加和刪除。若 $S = (a_0, a_1, \ldots, a_j, \ldots, a_i, \ldots, a_{n-1})$ 爲一堆疊，我們稱 a_0 爲底端 (bottom) 元素（最早加入者），而稱 a_{n-1} 爲頂端 (top) 元素（最後加入者）如下所示：

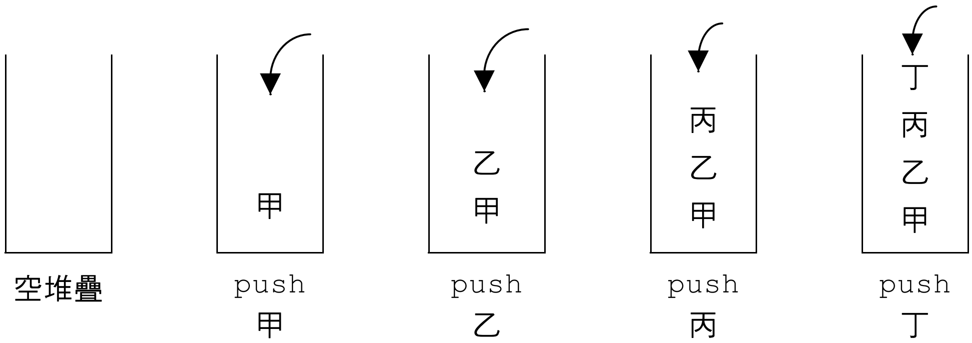
$$\text{底端} \leftarrow S = (a_0, a_1, \ldots, a_j, \ldots, a_i, \ldots, a_{n-1}) \rightarrow \text{頂端}$$

而且只要 $j < i$，則表示 a_i 比 a_j 後加入 S；亦即所有的加入或刪除動作，皆在 S 的頂端進行，如此即形成後進先出 LIFO（或先進後出 FILO）的特質。

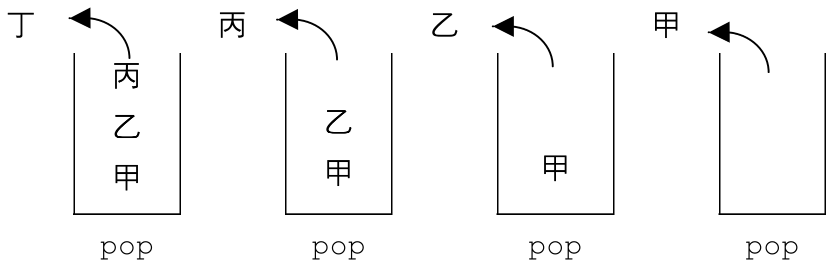
「增加元素進入堆疊」—稱爲 push，「自堆疊中取出元素」—稱爲 pop[1]。我們利用範例 3-1 中的圖 3-1 做爲堆疊結構的邏輯圖示。

範例 3-1

圖 3-1 (a) 描繪了依循增加 (push) 甲、乙、丙、丁，至堆疊 *S* 的邏輯圖示。圖 3-1 (b) 描繪了依循取出 (pop) 堆疊 *S* 內元素的邏輯圖示。



(a) 增加元素入堆疊中



(b) 取出（刪除）堆疊元素

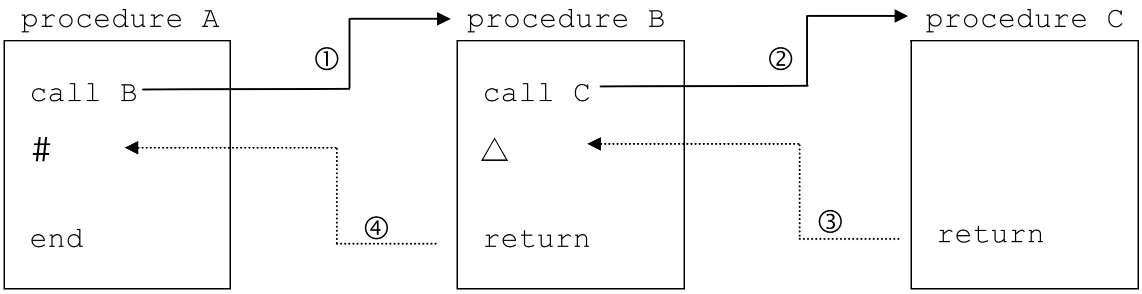
圖 3-1　堆疊的邏輯圖示

1　push 和 pop 與堆疊 stack 之間的關係，以英文原名詮釋十分貼切生動，本書會直接沿用 push 和 pop 來分別表示「增加元素進入堆疊」和「自堆疊中取出元素」。

在程式語言中，「程序呼叫」(procedure call) 執行完成後返回 (return) 呼叫處的動作，就是利用堆疊存放「返回呼叫地址」(return address) 的資訊，以妥善解決程式正確的往返流程，我們用範例 3-2 及範例 3-3 說明程序呼叫與堆疊的關係。

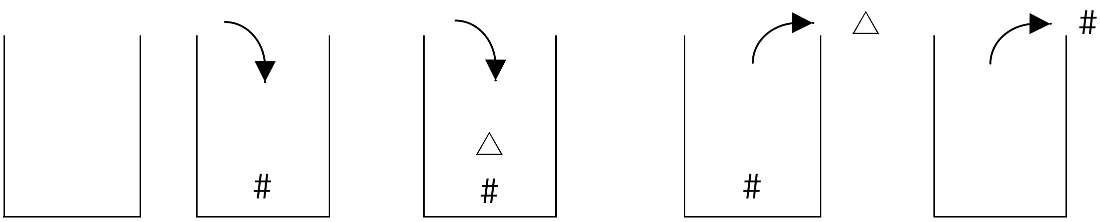
範例 3-2

試想某程式包含如圖 3-2 (a) 的程序呼叫，圖 3-2 (b) 描述利用堆疊解決此呼叫順序的過程。



①~④為程式流程轉移的順序；#和△表示程序呼叫結束後的返回處

(a) 程序呼叫



空堆疊

① 在 A 程序中呼叫 (call) 程序 B，遂將返回位址 # push 入堆疊中，流程轉至 B。

② 在程序 B 中呼叫程序 C，將返回位址 △ push 入堆疊中，流程轉至 C。

③ 程序 C 執行結束，pop 出堆疊頂端元素△，做為流程轉移的目的地 (返回 B)。

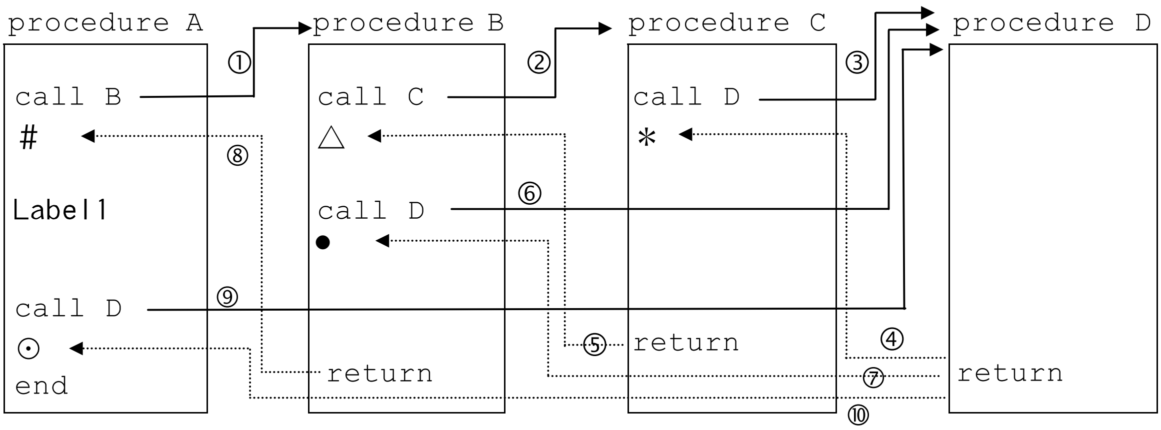
④ 程序 B 執行結束，pop 出堆疊頂端元素 #，做為流程轉移目的地 (返回 A)。

(b) 堆疊解決堆疊程序呼叫時，流程轉移的過程

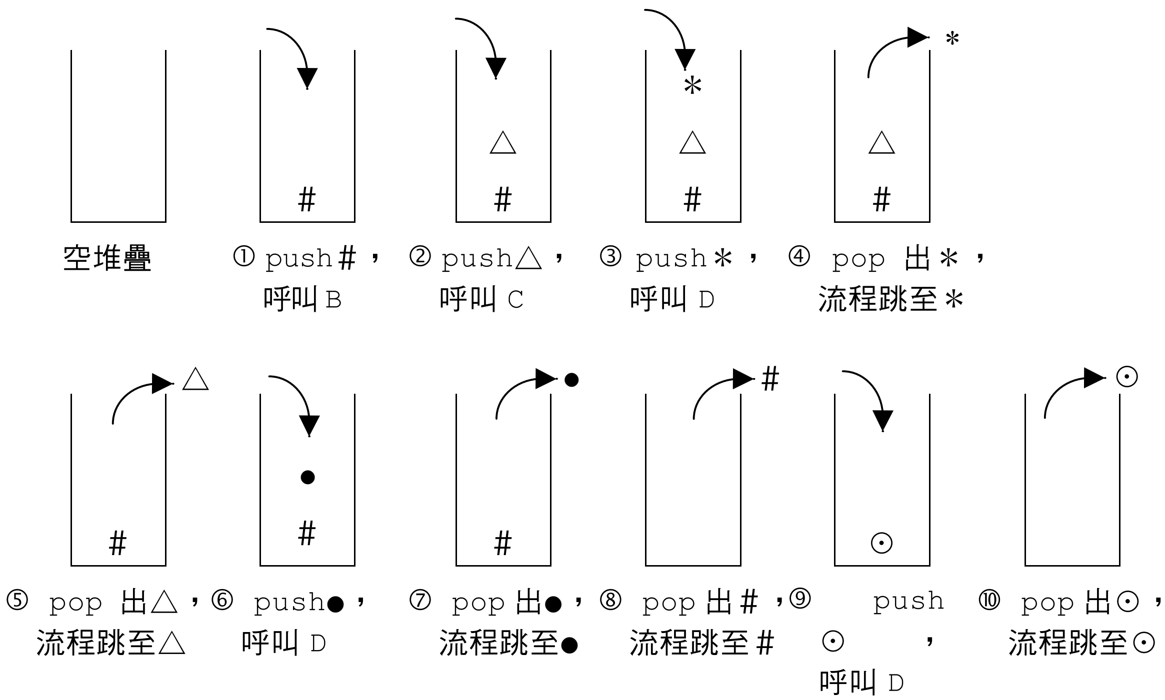
圖 3-2 堆疊保證了程序呼叫時流程轉移的正確性

範例 3-3

我們再舉一個較爲複雜的例子



(a) 程序呼叫



(b) 堆疊的操作過程

圖 3-3　堆疊保證了流程轉移的正確性（續）

由上面兩個範例可知：只要所寫程式的流程沒有錯誤，堆疊的使用可以確保流程轉移正確無誤[2]。以作業系統 (operating system) 或系統程式 (system programs) 的角度來看，堆疊的運用既然保證使用者程式的流程正確，自然是十分重要的資料結構，身為資訊專業的各位絕不能輕忽堆疊的重要性。

3.2 堆疊的基本運算

由上節的介紹，各位應知堆疊至少包含了加入（增加、push）與取出（刪除、pop）的動作；但試想堆疊會不會滿了，不夠新資料存放？或堆疊已空了，無法再取出任何元素？於是我們認為這四項動作—加入 (push)、取出 (pop)、檢查是否已滿、檢查是否已空—應是堆疊的基本運算。

堆疊的抽象定義可以用其它資料結構來實作，因此我們透過第二章介紹的陣列來實際建構堆疊。在 C 程式語言中，可用下面的宣告定義出堆疊：

程式 3-1 堆疊的宣告

```
1  #define maxsize 5
2  int Stack[maxsize];
3  int top = -1;
```

第 2 行中宣告了大小為 5 個整數的陣列，將做為堆疊使用，命名成 Stack；第 3 行宣告的 top 整數，將當成堆疊 Stack 頂端元素在陣列中的註標，宣告成-1 表示目前為一空堆疊，沒有任何頂端元素。從上述這樣的宣告即可將堆疊

2 每次程序呼叫，系統會為呼叫者設定其「堆疊框架」(stack frame)，將對應呼叫參數、返回地址、區域變數、前次呼叫堆疊框架的地址…等必要資訊，一併 push 存入系統堆疊，好讓被呼叫程序結束後，系統可經由 pop，取回原呼叫者呼叫當時的狀態，回復原呼叫者的當時情境，繼續該有的流程。

的四個基本運算建構如下：

程式 3-2 push **運算**

```
1   void push(int element)
2   {   if (IsFull()) StackFull();
3       else Stack[++top] = element;
4   }
```

在第 1 行中的宣告，傳入 push 程序的參數為將加入 Stack 的元素 element；第 2 行的陳敘先檢查 Stack 是否已滿了（利用 IsFull()，定義在程式 3-4），若已滿，則應進入『堆疊已滿』的程序（StackFull()）—請自行定義其內容，以給予程式撰寫者適當的回應[3]。第 3 行則將 element 加入陣列堆疊 Stack 中，其註標應是++top(top 加 1 之後的值)，因為 top 是目前頂端元素的位置，++top 即是加入 element 後，成為新頂端元素的位置（如圖 3-4 所示）。

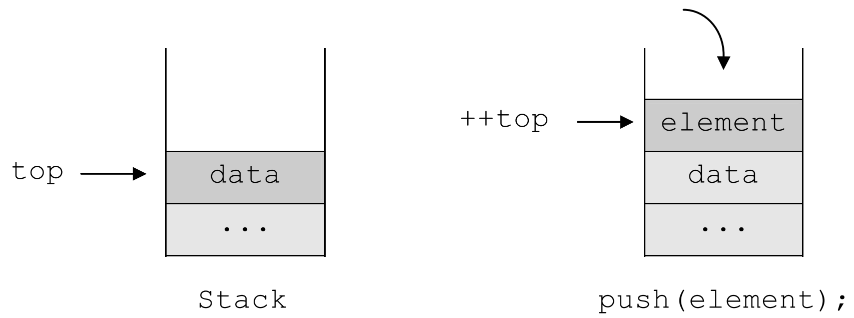


圖 3-4　push 運算

在此（程式 3-1 和 3-2）Stack 的使用乃全域變數 (global variable)；也有人習慣把堆疊也傳入程序 push 中，則程式 3-2 第 1 行的宣告可改為：

```
void push(int Stack[ ], int element)
```

3　StackFull() 可能包括錯誤訊息的輸出、程式停止等指令；也許僅印出「堆疊已滿」的警示訊息。

由於 Stack 本身已宣告成一陣列，則名稱 Stack 在 C 語言中將視為陣列之起始位址，透過傳址法 (call by address)，在 push 程序中即可定址出 Stack 中任一元素。

程式 3-3 pop 運算

```
1  int pop()
2  {   if (IsEmpty())
3      {   StackEmpty();
4          return -1;
5      }
6      return Stack[top--];
7  }
```

在第 1 行中宣告程序 pop 傳出的值為整數；請注意：在此用整數陣列來實作堆疊 Stack，若實際的狀況需要不同的資料型態，第一行中的資料型態宣告應與之一致。第 2 行到第 5 行乃處理『堆疊已空』（利用函數 IsEmpty() 來判斷，定義在程式 3-5） 的情況，各位可以根據實際需求，將堆疊已空的處理程序 (handling routine) 寫在 StackEmpty() 內，例如提醒使用者目前堆疊是空的…。執行完 StackEmpty() 後，傳回-1，讓呼叫 pop() 的程序知道 Stack 是空的，所以 pop 並不成功。第 6 行則將目前在 Stack[top] 位置的元素值，傳回呼叫程序，爾後 top 值少 1，指向下一個頂端元素。注意：雖然需要的 Stack[top] 已傳出，其值實際上並不需要自 Stack 中剔除，只要使 top 不再成為 Stack 頂端元素即可。

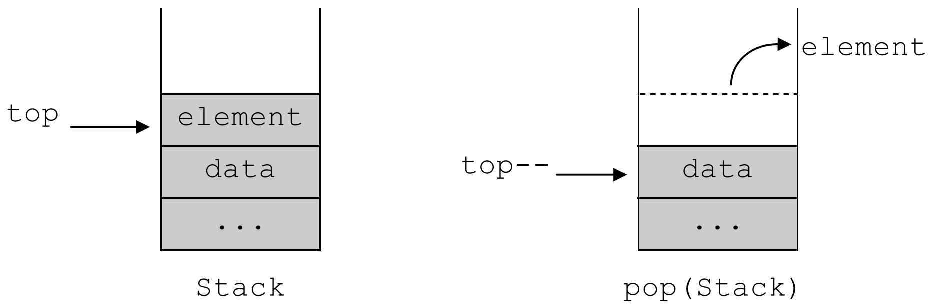


圖 3-5 pop 運算

程式 3-4 IsFull 運算

```
1  int IsFull( )
2  {   if (top == maxsize-1) return 1;
3      else return 0;
4  }
```

第 1 行中 IsFull()被宣告成 int，然其傳回結果只有兩種：1 或 0 （true 或 false），所以各位可用佔更少記憶位元的資料型態即夠用，有些 C 語言的編譯器版本即提供像 Boolean 的資料型態供人使用。在第 2 行中，我們檢查 top 是否已等於 maxsize-1（陣列 Stack 中[0]~[maxsize-1]的位置可能皆放滿元素），來決定是否堆疊已滿？若滿了傳回 1，否則傳回 0。

程式 3-5 IsEmpty 運算

```
1  int IsEmpty()
2  {   if (top == -1) return 1;
3      else return 0;
4  }
```

在第 2 行中我們利用 top 是否等於-1，來判斷堆疊是否爲空的，若空了傳回 1；否則傳回 0。

以上的說明用 C 語言中之最少指令，透過陣列來宣告堆疊，並定義堆疊的基本運算。但亦可將堆疊視爲一自訂資料結構型態 (user defined data type)、或是看成特定的結構物件 (structure object)，即堆疊陣列與頂端元素位置均是自訂資料結構型態或物件之成員；各位可試試如下利用 struct 宣告並搭配的基本運算定義：

程式 3-6 將堆疊視為物件

```
1  #define maxsize 5
2  typedef struct
```

```
3  {   int data[maxsize];
4      int top;
5  } StackType;
6  StackType Stack;
7  int IsFull(StackType *Stack)
8  {   if (Stack->top == maxsize-1) return 1;
9      else return 0;
10 }
11 int IsEmpty(StackType *Stack)
12 {   if (Stack->top == -1) return 1;
13     else return 0;
14 }
15 void push(StackType *Stack, int element)
16 {   if (IsFull(Stack)) StackFull();
17     else Stack->data[++Stack->top] = element;
18 }
19 int pop(StackType *Stack)
20 {   if (IsEmpty(Stack))
21     {   StackEmpty();
22         return 0;
23     }
24     return Stack->data[Stack->top--];
25 }
```

在第 2~6 行中定義 Stack 為一結構物件；此後在程序中 Stack->top 表示取用 Stack 結構中的 top 整數，Stack->data[~]表示取用 Stack 結構中的 data 陣列元素 data[~]；事實上這類自訂資料結構型態或物件，可以利用 C++中的模版 (template) 來定義，那麼整個堆疊可視為一類別 (class)，如此堆疊的使用即成為該類別的物件；這種物件導向設計 (object-oriented design) 的概念在開發大型應用軟體專案時非常有用，各位可自行練習。

3.3 佇列

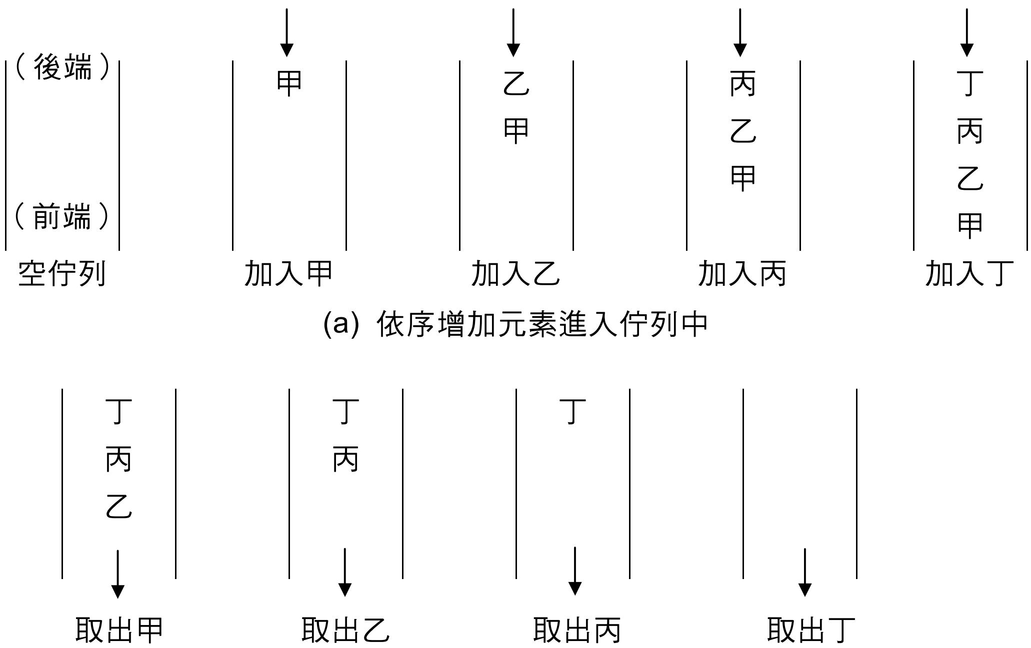
佇列 (queue) 是一個有序串列，為達成先進先出 FIFO 的效果，所有加入的動作皆在固定的一端進行，而所有刪除的動作則在另一端進行；若 $Q = (a_i, a_{i+1}, \ldots, a_j)$ 為一佇列，我們稱 a_i 為前端 (front) 元素，a_j 為後端 (rear) 元素，如下所示：

前端　$Q = (a_i, a_{i+1}, \ldots, a_j)$　後端

而資料的加入乃在 Q 的後端進行，資料的刪除則由 Q 的前端進行，如此即可形成先進先出的資料結構，我們利用範例 3-4 中的圖 3-6 做為佇列的邏輯圖示。

範例 3-4

圖 3-6 (a) 描繪了依循加入甲、乙、丙、丁至佇列 Q 的邏輯圖示，圖 3-6 (b) 描繪了依循刪除佇列 Q 內元素的邏輯圖示。



(b) 刪除佇列中的元素

圖 3-6　佇列的邏輯圖示及其基本運算

如前所述，佇列可模擬各種排隊、等候的情況。也是十分有用的資料結構。

3.4 佇列的基本運算

佇列與堆疊類似，也有加入、刪除、檢查是否已滿、檢查是否已空的基本運算。我們利用陣列實際建構佇列，由於有前端及後端的考量，當使用 C 語言時，我們可宣告佇列如下：

程式 3-7 佇列的宣告

```
1  #define maxsize 10
2  int Queue[maxsize];
3  int front = -1;
4  int rear = -1;
```

第 2 行以陣列宣告了 Queue，使其成爲大小爲 10 個整數的佇列，第 3 行中宣告前端元素註標 front 和後端元素註標 rear 皆設成-1，以表示目前 Queue 中沒有任何元素。下面是自後端加入元素的程序 addQ：

程式 3-8 addQ 運算

```
1  void addQ(int element)
2  {   if (IsQFull())  QueueFull();
3      else  Queue[++rear] = element;
4  }
```

在第 1 行中宣告了 addQ 程序，其傳入的參數爲欲加入的元素 element；第 2 行的指令先檢查是否已滿（利用 IsQFull()判斷，定義在程式 3-11），例如檢查後端的註標是否等於佇列的大小，若已滿了，應進入『佇列滿了』的程序（QueueFull()，請自行定義），以給予程式撰寫者適合的回應。第 3 行當 Queue 尚未滿，則將註標 rear 值增加 1（++rear）後，再把 element 放入註標所指定的位置，圖 3-7 說明加入佇列 addQ 運算的邏輯圖示。

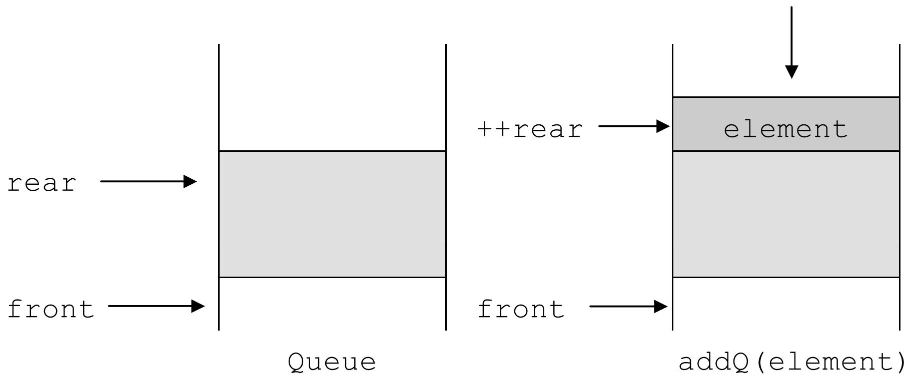


圖 3-7　加入元素進入佇列(addQ 運算)

下面是自前端刪除元素的程序 deleteQ：

程式 3-9 deleteQ **運算**

```
1   int deleteQ()
2   {   if (IsQEmpty())
3       {   QueueEmpty();
4           return 0;
5       }
6       else return Queue[++front];
7   }
```

第 1 行宣告了 deleteQ 的傳回資料型態為整數；第 2 行至第 5 行為『佇列已空』的判定及必要處理，此時傳回值為 0；第 6 行則當 Queue 不是空的時候，傳回 Queue[++front]，正是自該 Queue 中刪去的元素值，而且 front 註標也調整至正確的位置，圖 3-8 說明刪除佇列元素（deleteQ 運算）的邏輯圖示。

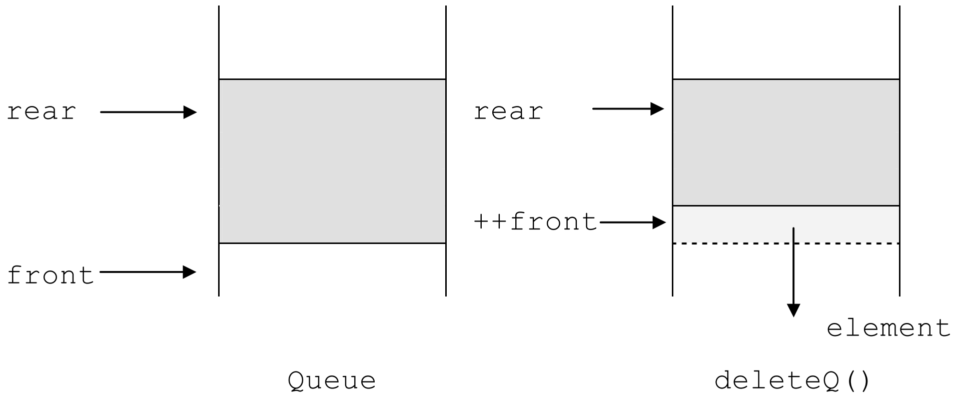


圖 3-8　自佇列中刪除元素（deleteQ 運算）

注意在圖 3-7 和圖 3-8 中，front 所指的位置是空的，試想一開始 front 和 rear 皆為-1，加入一個元素至佇列 Queue 中，則 rear 將加 1，但 front 未做更動，所以 front 仍指在前端元素的前一位置。這樣的安排可使程序 IsQEmpty()的檢測十分容易，如下所示。

程式 3-10 IsQEmpty 運算

```
int IsQEmpty()
{   if (rear == front) return 1;
    return 0 ;
}
```

程式 3-10 IsQEmpty()中不論佇列 rear 端如何增加、front 端如何刪除，只要 rear 等於 front 即表示 Queue 已空！犧牲一個陣列元素的空間，換取檢測佇列已空的便利；然而遇到 rear 和 front 皆等於 maxsize-1 時，則雖然 Queue 是空的，卻不能再加入任何元素（此時如下定義的 IsQFull()會成立）。事實上這個美中不足的現象，在下面的檢測佇列已空的 IsQFull 程序，會更加嚴重。

程式 3-11 IsQFull 運算

```
1   int IsQFull()
2   {    if (rear == maxsize-1) return 1;
3        return 0;
4   }
```

程式 3-11 的 IsQFull()與堆疊中定義的 IsFull()幾乎一模一樣（只在第 2 行的 rear 不同於 IsFull()用的 top），似乎可以表達佇列已空的狀態；但是 rear 等於 maxsize-1 時並不表示佇列實體容量已滿，請見圖 3-9！圖中的灰色區域確實有元素存在，但 Queue[0]~Queue[front]處之空白區域卻是空的；這歸咎於++rear 和++front 只會使陣列的註標逐漸增加，而使堆疊只往註標大的空間「滿」去。此時 addQ 內的指令自然無法使用到圖 3-9 中空白區域的空間。

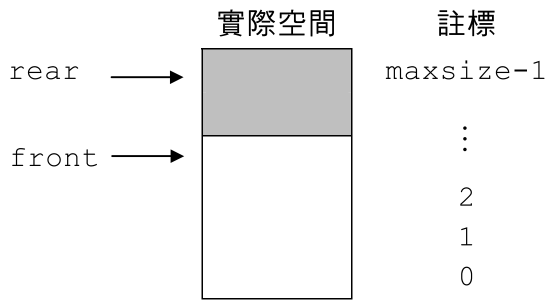


圖 3-9　佇列已滿？

也就因為如上線性 (linear) 地使用佇列無法盡如人意，遂有下面環狀佇列的產生。

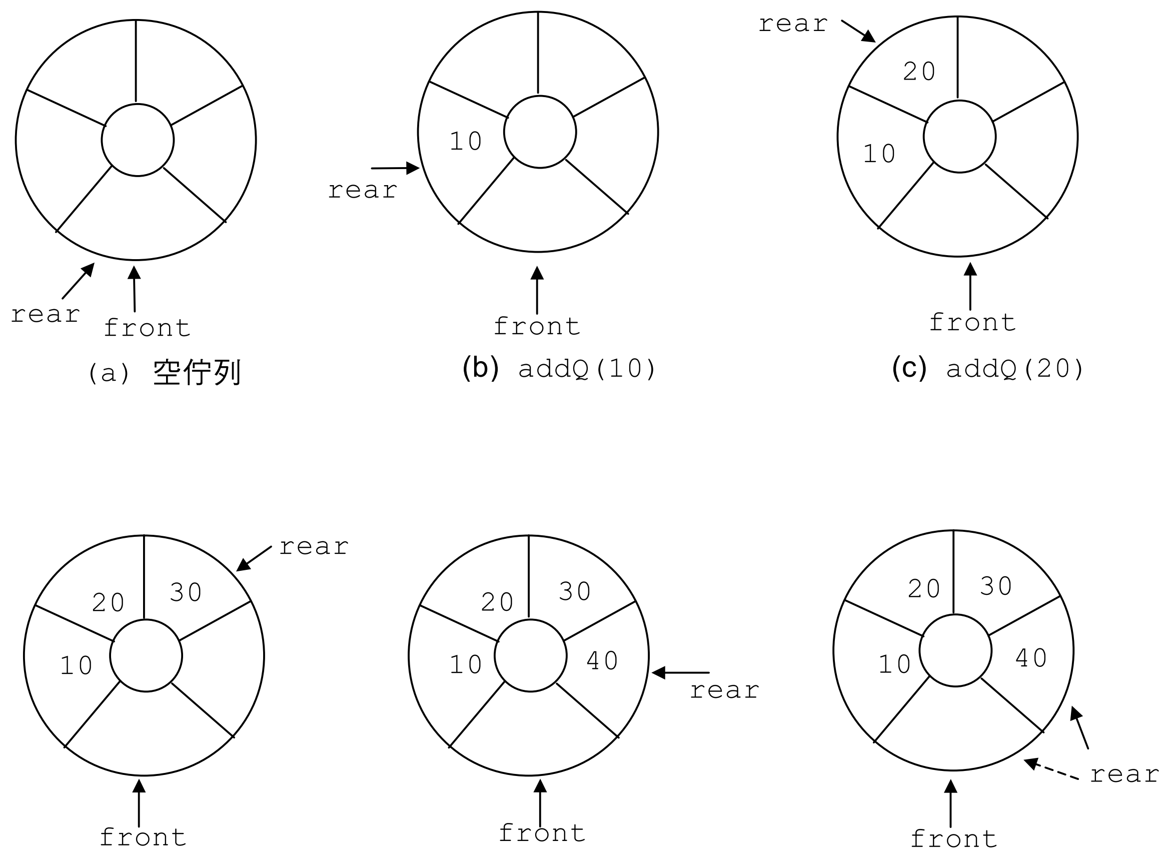
3.5 環狀佇列

線性的陣列因陣列元素位置與註標之對應關係，使其不方便模擬佇列的增加、刪除的消長現象，如圖 3-9 用過的空位無法再行利用；不過若將線性的陣

列折繞成「環狀」(circular) 陣列，就可再次使用到空出的空間，我們用範例 3-5 說明環狀陣列元素的增加及刪除。

範例 3-5

對大小為 5 的整數環狀佇列 `CQueue`，進行 `addCQ(10)`、`addCQ(20)`、`addCQ(30)`、`addCQ(40)`、`addCQ(50)`；`deleteCQ()`、`deleteCQ()`、`addCQ(60)`、`addCQ(70)` 等增加或刪除環狀佇列元素的動作，其運算的結果如圖 3-10 所示。在此我們仍保留 `front` 指在前端元素所在位置的想法。



(a) 空佇列　(b) addQ(10)　(c) addQ(20)

(d) addQ(30)　(e) addQ(40)　(f) addQ(50)但佇列已滿

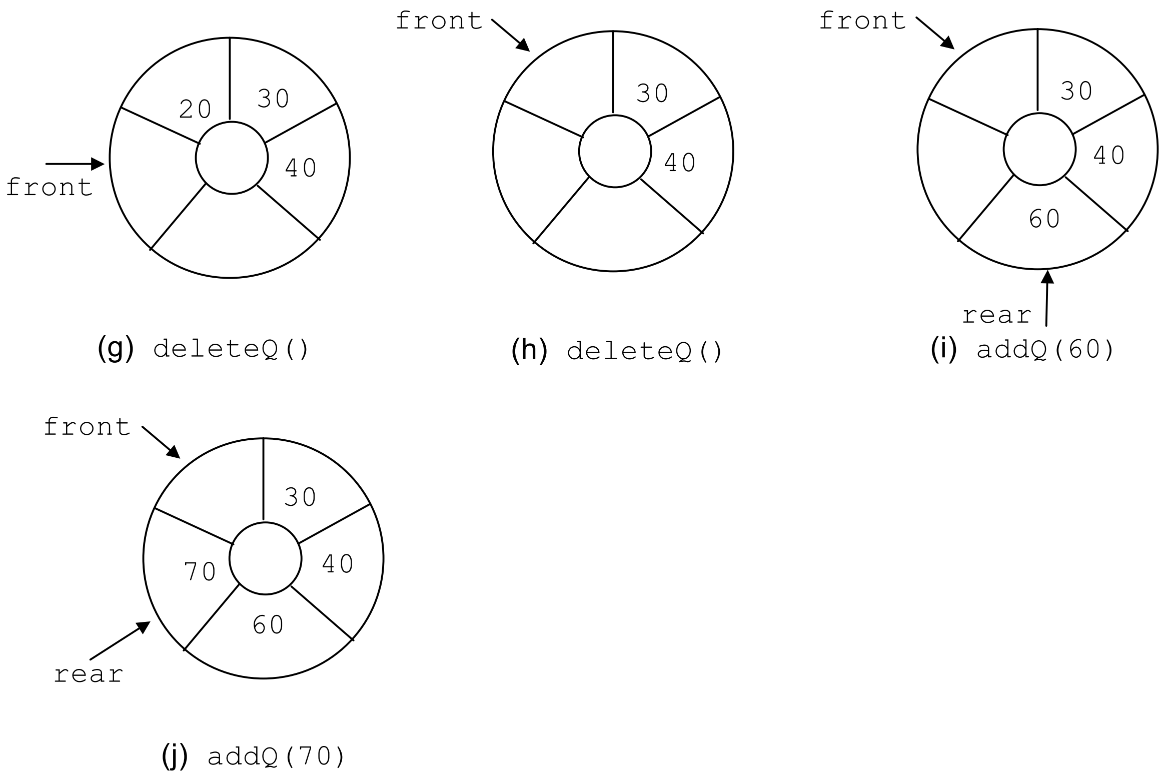


圖 3-10　環狀佇列元素的增加及刪除

範例 3-5 說明了環狀佇列的確可善用空出來的佇列空間，而爲了使註標能夠環狀計數：0, 1, 2, … , n−1, 0, 1, 2, … , n−1, 0, 1 … , …；只須利用模數（取餘數）運算 (modulo) 即可達成（在 C 中 `i%n` 即爲「取 `i` 除以 `n` 的餘數」運算，而 `i` 爲任意整數），請見範例 3-6 的例子。

範例 3-6

假設 $n = 5$，利用 `i%5` 將循序的數列 `i` 轉變爲呈環狀計數的數列：

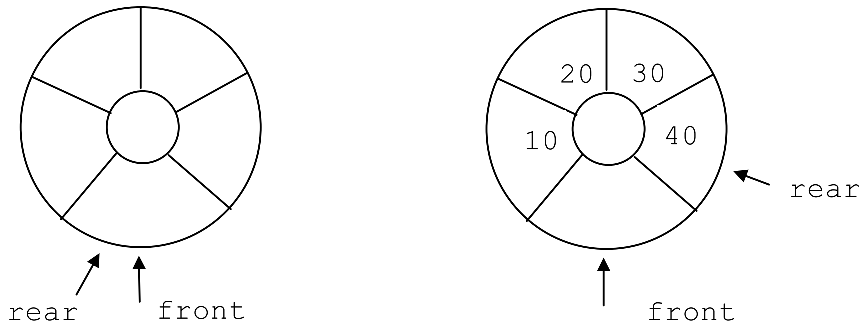
i	0	1	2	3	4	5	6	7	8	9	10	11	12	…
i%5	0	1	2	3	4	0	1	2	3	4	0	1	2	

於是原程式的 `++rear` 或 `++front`，須分別改爲 `++rear%n` 和 `++front%n`，其中 n 爲佇列的大小，即可有環狀佇列的正確註標。

範例 3-7 說明了：雖然少用環狀佇列中的一個空間，但可妥善判斷佇列已滿、和佇列已空的狀況，十分值得。

範例 3-7

在大小為 5 的環狀佇列 Queue 中，圖 3-11 (a) 為空佇列，此時 rear 等於 front；圖 3-11 (b) 為佇列已滿的狀況，此時 (rear+1)%5 等於 front。



(a) 空佇列　　(b) 佇列已滿

圖 3-11　環狀佇列已空或已滿

總結上面討論：front 所指向環狀佇列 CQueue 的位置為特意保留的空位；CQueue[(front+1)%maxsize] 方為環狀佇列 CQueue 的前端元素，而 Queue[rear%maxsize] 則為環狀佇列 Queue 的後端元素。所以重新定義 addCQ(), deleteCQ, IsCQEmpty() 及 IsCQFull() 如下：

程式 3-12 環狀佇列

```
#define maxsize 10
int CQueue[maxsize];
int front = 0;
int rear = 0;
void addCQ(int element)
{   if (IsCQFull()) CQueueFull();
    else CQueue[++rear%maxsize] = element;
}
```

```
int deleteCQ()
{   if (IsCQEmpty())
    {   CQueueEmpty();
        return 0;
    }
    else return CQueue[++front%maxsize];
}
int IsCQEmpty()
{   if (rear == front) return 1;
    return 0 ;
}
int IsCQFull()
{   if ((rear+1)%maxsize == front) return 1;
    return 0;
}
```

上面將堆疊和佇列的基本概念和基本運算做了介紹，接下來各節則利用幾個著名的例子來介紹堆疊或佇列的應用。

3.6 括號平衡

程式中經常會需要計算算術或代數運算式，除了運算式本身必須合乎語法外，所用的括號也必須合理地成對出現—稱其為「括號平衡」(balanced parentheses)。事實上「Dyck 字詞」(Dyck words) 指的就是由平衡的括號所構成的字詞，而平衡的括號所構成詞句即為「Dyck 語言」(Dyck language)[4]。因

4 Dyck 字詞或語言僅由左、右括號形成，利用上下文無關文法(context-free grammar)，可表示成：$S \rightarrow \varepsilon \mid \text{'('}\ S\ \text{')'}\ S$ 或 $S \rightarrow (\text{'('}\ S\ \text{')'})^*$；其中 ε 為空字串。至於 Dyck 之名取自學者 Walther von Dyck。

而編譯器需要判別運算式中的括號是否平衡、運算式是否合乎語法，因判斷的技巧就與堆疊的運用休戚相關。

我們廣義地考慮括號平衡字串：倘若字串中的字元只能是 ‘(’, ‘)’, ‘[’ 和 ‘]’，那麼我們可以定義括號平衡字串如下：

1) 空字串爲括號平衡；
2) 若 α 爲括號平衡，則 (α) 或 $[\alpha]$ 爲括號平衡；
3) 若 α 和 β 皆爲括號平衡，則 $\alpha\beta$ 亦爲括號平衡。

下面範例含有幾個括號平衡和非括號平衡的字串。

範例 3-8

令 A = “(([][]))”, B = “[()[(())][()]]”, C = “[(()))]” , D = “]()[“, E = “([)]“，則 A 和 B 爲括號平衡字串；而 C、D 和 E 則非也。

不知讀者有沒有直覺—這類左、右括號配對的問題，用堆疊恰可捕捉其成對出現的時機：

- 遇見左括號可將之放入堆疊（等待其對應的右括號）；
- 遇見右括號則取出堆疊頂端元素檢測：是否爲對應的左括號？
    - 若是，目前是平衡的；可繼續檢測；
    - 否則，已經不平衡，檢測可以停止了。

程式 3-13 將上述想法付諸實踐—檢測所輸入的字串是否爲括號平衡。

程式 3-13 檢測字串是否括號平衡

```
# define SIZE 256
char Stack[SIZE];
int top = -1;
void push(char x)
```

```
5  {   if (top == SIZE-1) cout << "Stack is full!" << endl;
6      else Stack[++top] = x;
7  }
8  char pop()
9  {   if (top == -1) return '#';    // Stack is empty!
10     else return Stack[top--];
11 }
12 int parenthesesBalanced(char s[], int n)
13 {   int i;
14     for (i=0; i<n; i++)
15     {   if (s[i] == '(' || s[i] == '[') push(s[i]);
16         else
17         {   if (s[i] == ')' && pop() != '(') return 0;
18             else if (s[i] == ']' && pop() != '[') return 0;
19         }
20     }
21     if (top >= 0) return 0;
22     else return 1;
23 }
```

程式 **3-13** 第 12 行 parenthesesBalanced 會取得檢測的字串 s 與其長度 n，第 14~20 行逐一檢查字元 s[i]：若其為兩種左括號之一，則直接 push 入堆疊(第 15 行)；否則該右括號就與所 pop 出的堆疊頂端元素比對，看它們是否為對應的左右括號(第 17~18 行)，一旦不是就回傳 0。第 20 行看一下堆疊是否還有字元，若有也不應該，也回傳 0；堆疊內確定沒任何字元-所有左右括號皆配對成功，即可回傳 1 了。

主程式希望先檢驗所讀入的字串是否符合所需，若有其他非限定的括號，即不再檢測之；而檢測結果須印出必要訊息；請見程式如下。

```
24 int main()
25 {   char s [size];
```

```
26     int len, NG;
27     while (top=-1, (cin >> s))
28     {  cout << "Dyck word = " << s << " ==> ";
29        for (NG=0, len=0; s[len] != '\0'; len++)
30            if (s[len]!='(' && s[len]!=')' &&
                  s[len]!='[' && s[len]!=']')
31            {   cout << "Illegal expression!" << endl;
32                NG = 1;
33            }
34        if (!NG)
35        { if (parenthesesBalanced(s, len)) cout << "YES" << endl;
36         else cout << "NO" << endl;
37        }
38     }
39     return 1;
40  }
```

由於字串的結尾會存入'\0'，第 29 行的迴圈即利用此條件來計算輸入字串 s 的長度 len；而且 29~33 行同時檢測 s 字串是否合乎題意－合則呼叫 parenthesesBalanced，依其結果做必要的輸出（35~36 行）；不合即換下一筆輸入。此程式所檢測的字串可見如下範例 3-9。

範例 3-9

```
Dyck word = (([])[[()[()]()()(())]]) ==> YES
Dyck word = [((([()()([][])]())()))] ==> YES
Dyck word = )[]( ==> NO
Dyck word = [[()(()[])]] ==> YES
Dyck word = (([]))[][]()) ==> NO
Dyck word = 0 ==> Illegal expression!
Dyck word = ((( ==> NO
Dyck word = (([](([([])[]])[][])[[()]])) ==> YES
```

3.7 老鼠走迷宮

老鼠走迷宮是個有趣的心理實驗，把老鼠放入迷宮中，老鼠可靠單純的嘗試錯誤 (trial and error) 法，即找到出口。這種嘗試錯誤的策略並不考慮運算時間是否經濟，但對走迷宮這類毫無線索的問題，依然是個可行的策略，我們或稱之爲「窮舉」(enumeration) 的策略。

我們可用電腦程式模擬老鼠走迷宮的過程，採用的策略即爲嘗試錯誤法，但是曾經走錯的路我們不應再次嘗試，雖然實際上老鼠不見得記得住曾走錯的路，但電腦在記憶方面可高明的多，當走錯（或往下沒路了）應退回「上一步」（後進者先取出），該步又沒路了，則退回「上上一步」…。事實上用來「記住窮舉過程中已走過的路徑」最好的資料結構就是堆疊！我們現在逐步介紹如何構思解決老鼠走迷宮問題的演算法。

首先應先設想必須的資料結構[5]：

- 迷宮的表示（用二維陣列如何？隔牆、通路如何區隔？）
- 路徑的表示（用表示迷宮二維陣列的註標如何？）
- 走過的路徑（用另一個與迷宮二維陣列一樣大小的二維陣列，記錄是否走過如何？）
- 遭遇錯誤後的移動（利用堆疊記得從何而來（上一步）？）

於是我們用一個二維陣列 `maze[i][j]`，$1 \le i \le m$，$1 \le j \le p$，來表示一個 m×p 大小的迷宮；並以 1 表示不能走的區域（牆面、隔板…），0 表示可以走的區域；簡單起見，假設入口爲 `maze[1][1]`，出口在 `maze[m][p]`。圖 3-12 是一個大小爲 10×12 的迷宮。

5　各位應養成思考問題時，同時融入資料結構考量的能力。

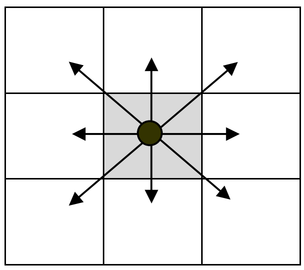
入口 → maze[1][1]

0	0	0	1	0	0	0	0	0	1	0	1
1	1	0	1	0	1	1	1	0	1	0	0
0	1	0	0	0	0	1	0	0	1	0	1
1	1	0	1	1	0	1	0	1	1	0	1
1	1	0	1	0	0	1	0	0	0	0	1
1	1	1	1	0	1	0	0	1	1	1	1
0	1	1	0	0	1	0	1	1	0	0	0
0	0	0	0	1	1	0	0	0	0	1	0
0	1	1	0	0	0	1	0	1	0	1	0
0	1	1	1	1	0	0	0	0	0	0	0

→ 出口 maze[10][12]

圖 3-12 可能的迷宮

路徑的表示用「陣列 maze 上的註標」是個很好的選擇。在 maze[i][j] 上的老鼠可能的移動有八個選擇，如圖 3-13 (a) 所示；而此八個可能的移動選擇，其座標變化會如圖 3-13 (b) 所示。



(a) 老鼠可能的移動選擇共有八個

NW		N		NE
	[i-1][j-1]	[i-1][j]	[i-1][j+1]	
W	[i][j-1]	[i][j]	[i][j+1]	E
	[i+1][j-1]	[i+1][j]	[i+1][j+1]	
SW		S		SE

(b) 八個可能的移動選擇與其座標

圖 3-13 老鼠可能移動的選擇

而如何在程式中讓老鼠順利自 maze[i][j] 處移動到下一個位置呢？從註標的觀點來看，這八個可能的移動位置，只須利用註標加 1、減 1 或維持不變的運算即可完成，如圖 3-13 (b) 所示，我們將此加 1、減 1 或維持不變的量稱爲「位移」(offset)，利用位移量來決定註標的增減，是個程式設計常用的技巧[6]，我們將可能的移動位移製表如圖 3-14。

dir	N	NE	E	SE	S	SW	W	NW
move[dir].x	-1	-1	0	1	1	1	0	-1
move[dir].y	0	1	1	1	0	-1	-1	-1

圖 3-14　移動方位與座標位移的對照表

要利用圖 3-14 的移動位移表仍需要一些小技巧：我們利用「列舉資料型態」(enumeration data type) 將 N, NE, E, … , NW 此八個方向，定義成具有順序、可做爲陣列註標用的列舉資料型態 directions：

```
enum directions {N, NE, E, SE, S, SW, W, NW};
```

對 C 而言，如此定義的列舉資料型態 directions 將使八個整數：0, 1, 2, …, 7，分別與 N, NE, E, … , NW 對應；即 N=0, NE=1,E=2, … ,W=7[7]（有可能因編譯器之不同而有所差異）。如此一來若老鼠目前的位置在 maze[i][j]，想移動的方向爲 d，則欲移動的位置即爲 maze[i+move[d].x][j+move[d].y]；這樣的設計可使我們用較爲精簡的程式碼，掌握老鼠在迷宮中的移動。

6　在作業系統中，改變位移也常用在記憶體的定址、在使用者記憶體間切換、…等場合。

7　若想使 N=1, NE=2,…，可如下宣告：

```
enum directions {N=1, NE, E, SE, S, SW, W, NW};
```

至於走過的路徑，可由二維陣列 `maze` 的二個註標，以及移動到下一步的方向（或移動到下一步的下一個方向）所形成的有序串列來表示，如圖 3-15。

0	0	0	1
1	1	0	1
0	1	0	1
1	1	0	0

(a)可能的路徑

(1,1,E)	(1,2,E)	(1,3,S)	
		(2,3,S)	
		(3,3,SE)	
			(4,4,E)

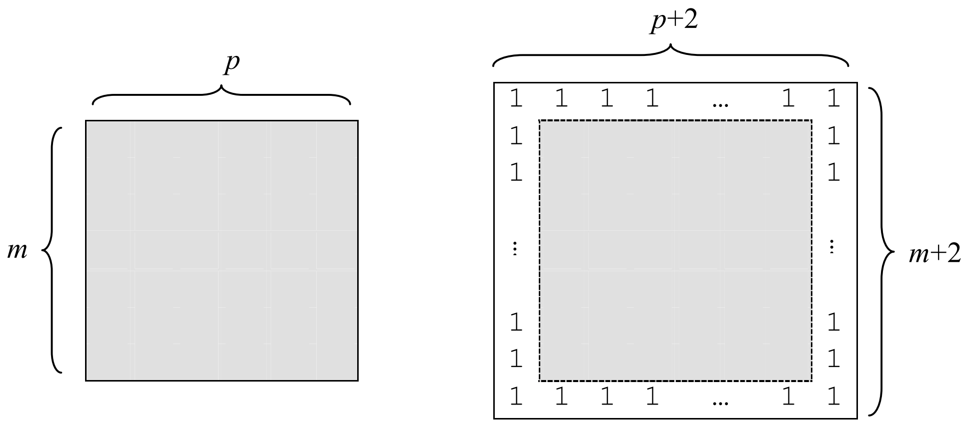
(b) 對應 (a) 路徑的有序串列

圖 3-15　走過的路徑可用有序串列來表示

注意：並不是每個位置上的老鼠，都有八個可能的移動選擇，如圖 3-16 中標示灰色的迷宮四周邊緣都不足八個可能的移動。因此爲使程式碼更具一般性，節省檢查是否位於邊緣的麻煩，我們可將迷宮外圍加上隔板，即迷宮大小改成 $(m+2)\times(p+2)$，如圖 3-17 (b)，而外圍區域皆設爲 `1`，表示不可走；如此一來，在原迷宮內的每個位置就都有八個鄰點了。

0	0	0	...
1	1		
0			
...			

圖 3-16　陣列邊緣灰色區域的位置不足八個鄰點（可能的移動選擇）

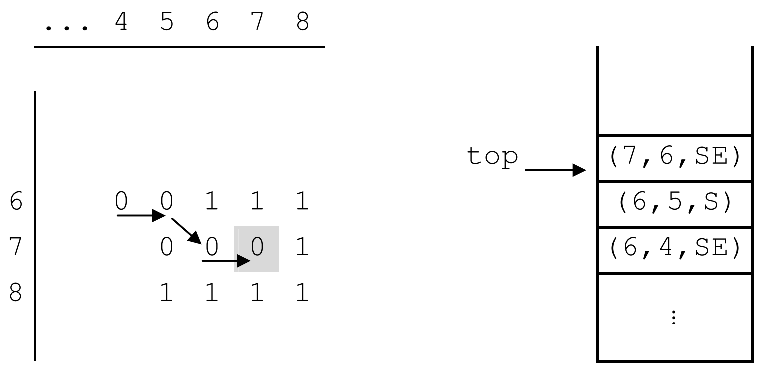


(a) 原 $m \times p$ 的迷宮　　(b) 加上外圍的 $(m+2)\times(p+2)$ 迷宮

圖 3-17　加上外圍的迷宮

自起點開始，如何決定下一步的判斷如下：每個位置的八個可能方向，都是嘗試錯誤的對象，我們可讓老鼠依預先規定的順序逐一嘗試（在之前 `enum directions` 宣告中所定的方向是由 `N` 開始，依順時鐘方向，訂出其他方向的順序）。範例 3-10 考慮了老鼠走了若干步後，面臨無路繼續的情形。一旦確定下一步了，放入堆疊的是：（`x,y,d`）；其中（`x,y`）是目前位置的座標、`d` 是目前從（`x,y`）移動到下一步所採用方向的下個可能（亦即退回到這一位置時，應該繼續嘗試的方向）。

範例 3-10

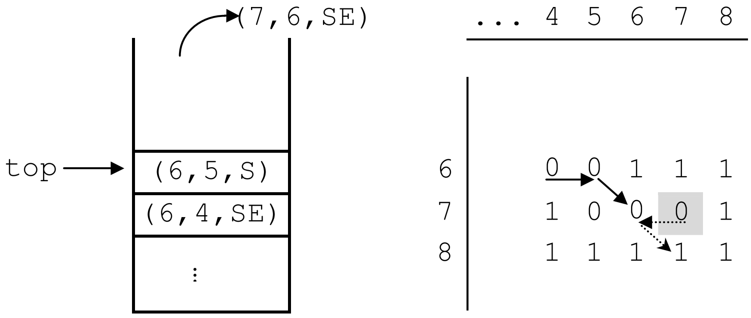


(a) 老鼠無路可走　　(b) 走過的路徑存在堆疊中

圖 3-18　利用堆疊存放走過的路徑，以解決無路可走時的窘境

如圖 3-18 (a) 所示，在 (7,7) 處的老鼠，其八個可能嘗試(含自 (7,6) 走來的上一步)皆不能繼續走：我們應退回上一步 (7,6)，嘗試該處的下一個可能方向 SE；這種後進先出的特性，即可利用堆疊了！圖 3-18 (b) 顯示了當時堆疊中存放的已走過路徑。範例 3-11 說明了利用堆疊掌握「返回行蹤」(backtrack) 行蹤的技巧。

範例 3-11



(a) pop 堆疊找出上一步　(b) 走回上一步並取得下個嘗試的方向

圖 3-19　利用堆疊掌握返回行蹤

老鼠在 (7,7) 處沒有往下走的選擇，遂對圖 3-18 (b) 的堆疊做 pop，如圖 3-19 (a) 所示；得到的前一步爲 (7,6,E)，如圖 3-19 (b) 所示—足見堆疊掌握老鼠行蹤之效。於是回到 (7,6)，依順時鐘方向，依序考慮：SE, S, SW, W...，然 SE, S, SW 皆爲不可走的方向，所以下個前往方向爲 W，即往西前往 (7,5)；此時將目前位置和前往方向 W： (7,6,W) push 入堆疊中。然在 (7,5) 處的第一個嘗試爲 N，即往 (6,5) 走，但 (6,5) 之前已走過！重覆再試已走過的路徑實在不夠高明；因此我們另外宣告一個二維 $(m+2)\times(p+2)$ 陣列 mask，一開始與 maze 一模一樣，每次老鼠來到 maze[i][j]即把 mask[i][j]改成 1，表示老鼠已走過此處。於是上述在 (7,5) 處，因 mask[6][5]爲 1，不應再嘗試往 (6,5) 走訪。

如果不必保留 maze 此二維陣列所表示的迷宮，則若 maze[i][j]爲老鼠

的嘗試，可直接把 maze[i][j]改成 1，可省下 mask 二維陣列的空間。綜合以上討論，下面是解決老鼠走迷宮問題的演算法：

演算法 3-1 老鼠走迷宮：

輸入：迷宮，以二維陣列 maze 表示。
輸出：從入口至出口的路徑。

```
1   (i,j,d)=(1,1,E);   //已知入口處上方皆不可走
2   push((i,j,d));     //一旦 pop 出(1,1)其下個嘗試方向為 E
3   while (堆疊仍有資料) do
4   {   (i,j,d) = pop();
5       while (在(i,j)處仍有路可走) do      //d<=NW
6       {   (u,v) = 自(i,j)處欲嘗試的下一步座標;   //利用 d 查位移表
7           if ((u==m) && (v==p))
8           {   成功找到出口，輸出路徑，可以停止了  }
9           if ((!maze[u][v]) && (!mark[u][v]))
10          //(u,v)可以走(!maze[u][v]成立)，且不曾走過(!mark[u][v]成立)
11          {   mark[u][v] = 1;    //記錄(u,v)已走過 (以 dir 此方向)
12              d = 下一個嘗試的方向;
13              push((i,j,d))
14              i=u; j=v; dir=N;
15          }
16          else d = 嘗試下一個可能的方向;
17      }
18  }
```

其中第 2~4 行是利用堆疊時常用的程式技巧！各位會發現先在第 2 行中 push 資料入堆疊中，可使第 3 行的迴圈結束測試更具一般性，即堆疊爲空的情況可以正常結束，而一開始堆疊原爲空的狀況，會因第 2 行預先 push 第一

步 (1,1,E)，而不復存在 (初始狀態的考量已自然融入)。第 4 行的 pop 則自堆疊中取出上一步資料，開始迴圈內的動作[8]。爲使 push/pop 入堆疊中的路徑資料更具結構化，我們宣告輔助的資料結構如下：

程式 3-14 路徑資料的結構化宣告

```
1  #define possible_direction 8
2  struct offset
3  {   int dx,dy;
4  };
5  enum directions {N, NE, E, SE, S, SW, W, NW};
6  struct offset move[possible_direction];
7  struct position
8  {   int x,y;
9      directions dir;
10 };
```

第 2 行至第 4 行將老鼠走一步在 x-y 座標上的位移量，宣告成一結構 struct offset，而每一個可能移動的方向皆對應了一組位移量（即一組 (dx, dy) ，型態爲 offset），共計有 8 個方向，每個方向皆需要一組 offset 型態的位移量，遂在第 6 行，爲這 8 個可能的移動方向宣告了：

```
struct offset move[possible_direction];
```

有了如上的宣告，再將圖 3-14 的位移表填入位移結構 move 中，則若老鼠現在位於 (i,j) 上，想往北方 N 移動，其所到座標 (u,v) 即可求出：

8 迴圈內其實是「返回上一步」的動作；一開始我們在第 2 行中 push 第一步，第 3 行的檢測自然過關而進入迴圈，在第 4 行（迴圈內）隨即 pop 之，即可將「返回上一步」的動作，套用在這第一步上。初始設定因而與「堆疊是否仍有元素而須繼續執行」的迴圈巧妙地融合在 3~4 行中。

```
u = i+move[N].dx;  v = j+move[N].dy;
```

第 7 行至第 10 行則定義出結構 struct position，包含了型態為整數的 x、y 座標值和型態為 directions 的移動方向 dir，這組結構將做為 push 入和 pop 出堆疊的基本元素資料型態，或可以說堆疊的元素型態即為 struct position，在下面的程序中我們會宣告此堆疊。於是演算法 **3-1** 可改寫成下面的程序：

程式 3-15 老鼠走迷宮（上接程式 3-14）

```
int m,p,top;
void push(struct position data)
{   if (top == (m*p-1)) StackFull();
    else Stack[++top] = data;
}
struct position pop()
{   if (top == -1) StackEmpty();
    else return Stack[top--];
}
void path(int m, int p)
{   struct position Stack[m*p];
    struct position step;
    int i,j,u,v;
    directions d;
    step.x=1;  step.y=1;  step.dir=E;
    push(step);
    while (top!=-1)    //堆疊內仍有元素
    {   step=pop();
        i=step.x;  j=step.y;  d=step.dir;
        while (d<=NW)
        {   // (i,j)處還有可移動的選擇
            u=i+move[d].dx;  v=j+move[d].dy;
```

```
23              if ((u==m) && (v==p))
24              {   // 輸出(u,v),(i,j)，再將 Stack 內的所有元素
25                  // 逐一 pop 輸出，則構成一條由出口至入口的路徑
26                  return;
27              }
28              if ((!maze[u][v]) && (!mask[u][v]))
29              {   mask[u][v]=1;
30                  step.x=i;  step.y=j;  step.dir=d+1;
31                  push(step);        // 記錄(i,j,dir)在堆疊中
32                  i=u;  j=v;  d=N;   // 前往下一步
33              }
34              else d++;       // 嘗試下一個可能的方向
35          }
36      }
37      // 輸出："此迷宮無出路"之訊息
38  }
```

程式 3-15 中第 3 和第 7 行分別用 top==(m*p-1)和 top==-1 來檢查堆疊是否已滿和已空了！第 20 行用了 d<=NW 的比較指令，因 d 的資料型態爲 directions，是一列舉資料型態，它可被當成整數運算，自然可以與同爲 directions 的常數 NW 比較大小；第 30 行的 step.dir=d+1 是 pop 至此時的下個嘗試方向；這個程式的輸出部份尚未詳予寫出，各位可視自己的需要完成之，在第 24 至 25 行中輸出(u,v)、(i,j)以及堆疊中的各個位置座標，它們構成了由出口回溯至入口的完整路徑。

第 11 行中堆疊的大小宣告爲 $m \times p$，這是個高估的上限值，這是把直覺的最差情況：「所有迷宮的位置都可能被老鼠走過而加入堆疊中」，直接做爲堆疊的上限值；至於詳細推敲老鼠走迷宮的最差狀況—請看範例 3-12。其中圖 3-20 (c) 中的假想迷宮大小爲 $m \times p$，而沿第一列一路往東，至第 p 欄後再一路往南，即可走出迷宮，這種情況被放入堆疊的元素只不過爲 $m+p$ 個。而老鼠走迷宮的最壞情況應如圖 3-20 (d) 所示的情況，需有$(m/2) \times p+m/2$ 個位置被放入堆疊中，而實際在程式中宣告的堆疊大小，用 $\max(m/2 \times (p+1), p/2 \times (m+1))$ 就已足夠。

範例 3-12

圖 3-20 (a) 和 (b) 顯示了演算法 3-1（程式 3-14 和 3-15）經實作後求解一個迷宮的例子：(a) 中的通通呈現 * 或 @ 符號者是爲老鼠曾經走過的路徑，而呈現 @ 者則爲自出口回溯至入口的路徑；(b) 只留下了出口回溯至入口的路徑——改以  圖案呈現（曾經嘗試但不爲該路徑成員者不顯示）。圖 3-20 (c) 爲老鼠走迷宮的最佳狀況，而 (d) 爲最差狀況。

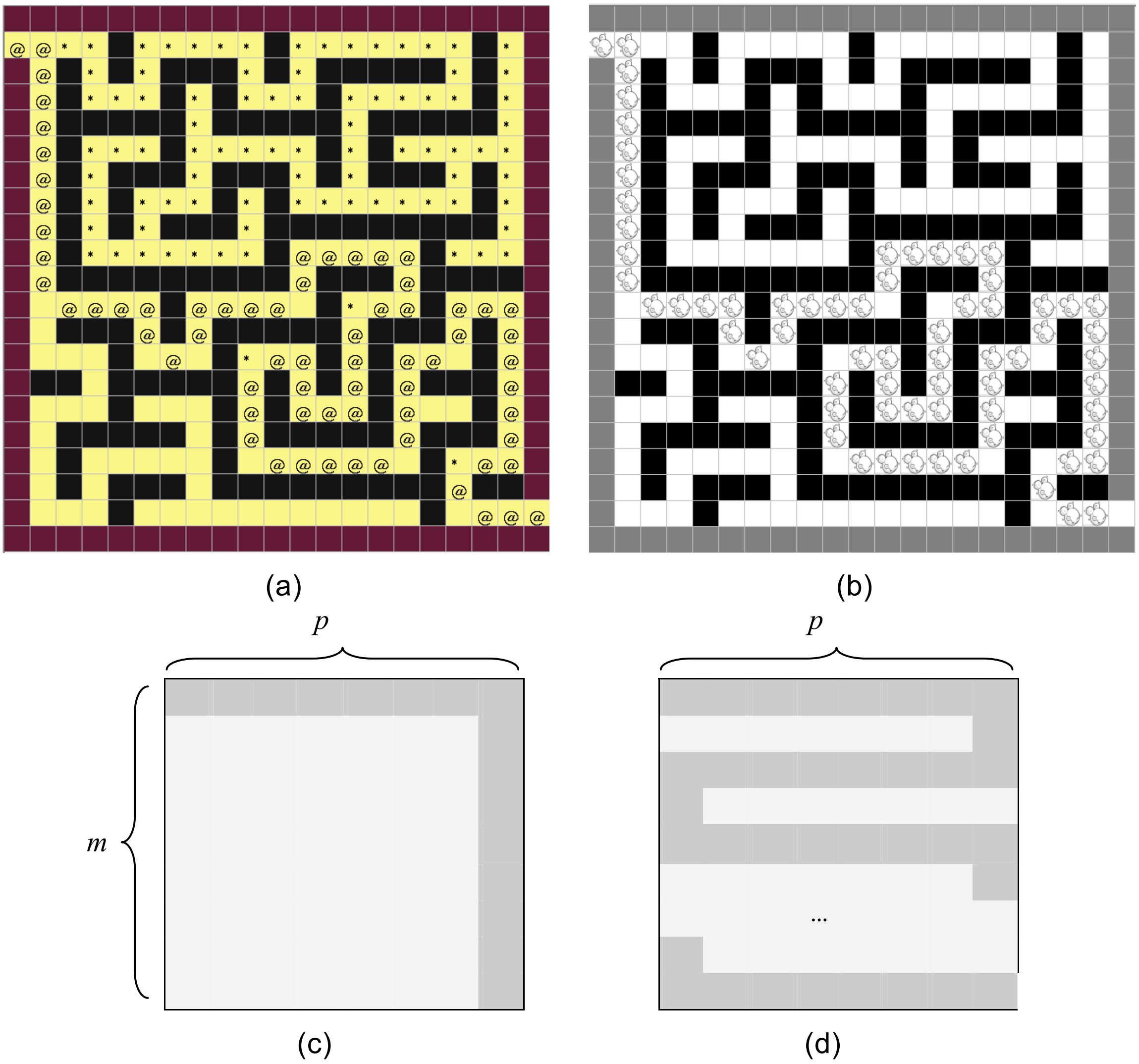


圖 3-20　老鼠走迷宮的可能路徑

老鼠走迷宮的時間複雜度亦爲 $O(mp)$；理由是每一個位置至多被走訪（處理：檢視可否走訪、push 至堆疊、pop 出堆疊…）一次。

3.8 運算式的轉換和求值

在電腦中處理算術運算式 (arithmetic expression)，需要堆疊的協助，在本節中即介紹兩者之間的關係。

3.8.1 算術運算式

在程式語言中，算術運算式經常被用來計算希望的結果，例如下面的指定敘述 (assignment statement)：

$$X = A+(B-C)*D$$

等號右邊即爲一算術運算式；一個算術運算式包括了運算子 (operator) 如：+, –, *, /, …等；和運算元 (operand) 如上式的 A, B, C 和 D。除此之外，運算式的計算得依據運算子的優先順序 (priority)，方可算出正確的結果。數學上運算子的優先順序（數字大者優先順序高）如表 3-1 所示：

表 3-1 運算子的優先順序（以 C 語言為例）

優先順序	運算子
7	負號、! (邏輯否定)
6	*、/、%
5	+、-
4	<、<=、>=、>
3	==、!=
2	&&
1	||

運算子依表 3-1 的優先順序（數值愈大，優先順序愈高），將其對應的運算元加以計算；相同順序的運算子，則依由左至右的順序進行計算；若有括號內者應先計算（先內層後外層）。於是下式與上式是一樣的運算式：

$$X = (A+((B-C)*D))$$

若沒有好的演算法來決定運算的先後順序，用電腦來解決運算式求值 (evaluation) 的問題依然相當困難。在下一節中我們會介紹後序 (postfix) 運算表示法，並且搭配先前所介紹的堆疊，即可順利地解決運算式求值的問題。

3.8.2 後序運運算表示式

傳統的算術運算式將運算子放在運算元的中間，遂稱之為「中序運算表示式」(infix expression notation)[9]。而「後序運算表示法」(postfix expression notation) 則將運算子放在對應運算元之後；類似地，「前序運算表示法」(prefix expression notation) 則將運算子放在對應運算元之前。範例 3-13 列出幾個運算式的不同表示法。

範例 3-13

中序表示	後序表示	前序表示
B-C	BC-	-BC
(B-C)*D	BC-D*	*-BCD
A+(B-C)*D	ABC-D*+	+A*-BCD

9　這裡的運算子指的是二元運算子 (binary operator)；單元運算子 (unary operator) 如：負號、否定、…等並未在本節中討論。

事實上只需加上適當的括號，我們可以輕易地轉換運算式的不同表示法，請見範例 3-14：

範例 3-14

將中序運算式 A+(B-C)*D 加上適當的括號：

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

把運算子挪放到對應右括號的左邊，再去掉括號，即形成其後序運算式：

(A+((B-C)*D)) ⇨ ABC-D*+

把運算子挪放到對應左括號的右邊，再去掉括號，即形成其前序運算式：

(A+((B-C)*D)) ⇨ +A*-BCD

試想若編輯器中只存放中序運算式，而只會自左向右檢查式中的優先順序，則編輯器須多次解讀此中序運算式：

A+(B-C)*D

第一次：	A+X_1*D	（X_1 存放 B-C 的結果）
第二次：	A+X_2	（X_2 存放 X_1*D 的結果）
第三次：	X_3	（X_3 存放 A+X_2 的結果）

這種「逐次自左向右挑出順序最高者執行」方法的效率是很差的。本節將介紹如何利用後序運算式，配合堆疊，解決運算式求值的問題。請看範例 3-13：

範例 3-15

以 A+(B-C)*D 爲例，令

$$X_1=B-C、X_2=(B-C)*D、X_3=A+(B-C)*D；$$

則其後序運算式 ABC-D*+的求值運算，只須運用堆疊 *S*，並自左向右查看後序運算式一次即可完成。過程中遇見運算元就直接 push 入堆疊中，遇見運算子則 pop 堆疊兩次，取得的兩個運算元即以該運算子做計算，再將結果 push 入堆疊中，直至後序運算式的每一元素皆看完爲止。請看圖 3-21：

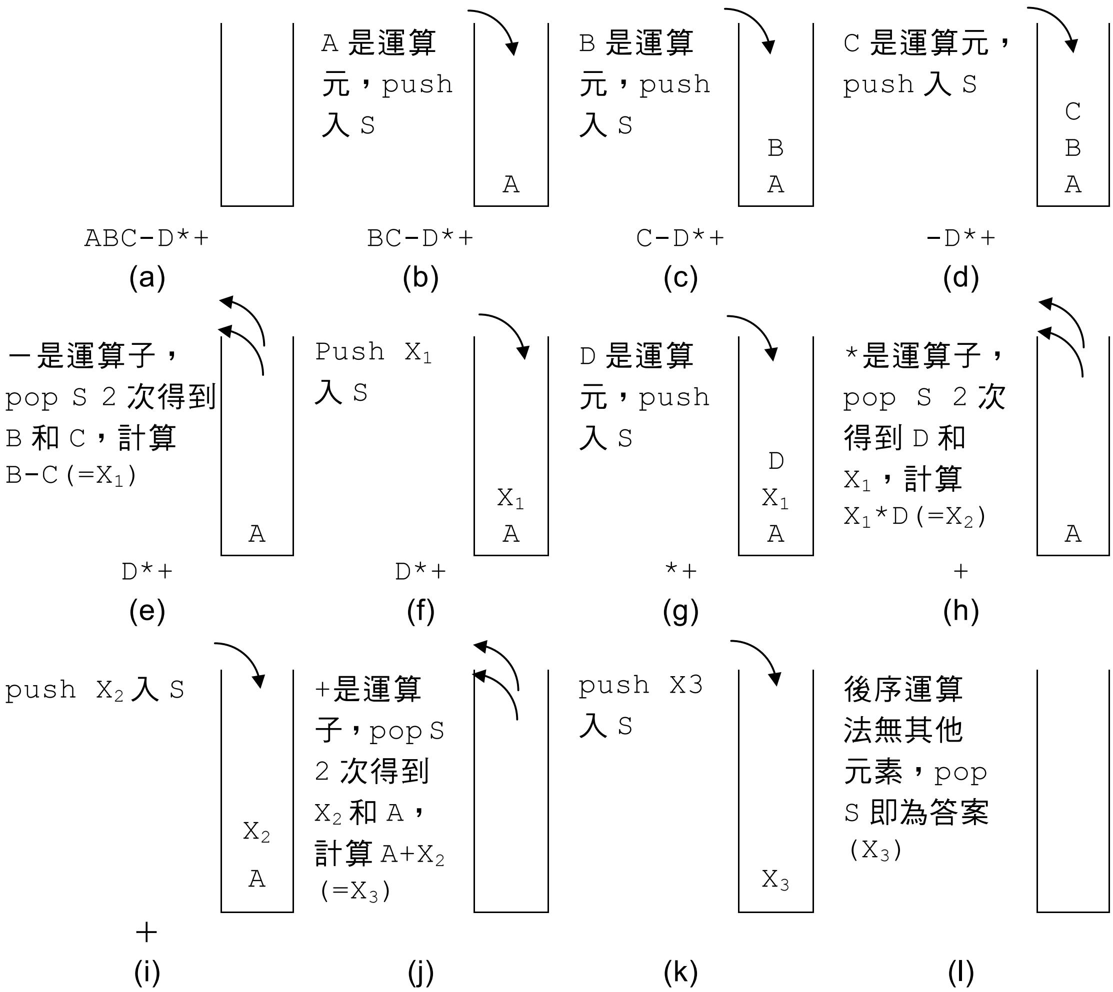


圖 3-21　後序運算式的求值過程

我們將範例 3-15 的想法撰寫成演算法如下：

演算法 3-2 後序運算式的求值運算：

```
輸入：後序運算式 e
輸出：計算出後序運算式 e 的值
1  n = parsing(e, token);
2  for (i=0; i<n; i++)
3  {   if (token[i]為運算元) push(token[i]);
4      else
5      {   pop 出 token[i]此運算子所需的運算元;
6          計算出 token[i]此運算的值，令為 x;
7          push(x);
8      }
9  }
```

在第 1 行中，我們利用一子程序 parsing(e,token)將傳入的後序運算式 e 的所有運算元與運算子，分別依序存在 token 陣列中，共計 n 個運算元與運算子，並傳回 n 值。自第 2 行起的迴圈，即逐一檢測 token[i]是運算元、亦或運算子？若是運算元，則 push 入堆疊中（第 3 行）；若是運算子，則 pop 兩次，取得 token[i]此運算子所需的兩個運算元，據以計算出 token[i]此運算的值，再將此結果 push 入堆疊中（第 5~7 行）。至於如何將傳統的中序運算法，轉成後序運算式[10]，請見下一節。

各位應可發現這樣的求值方法，不必考慮運算子的優先順序，只要自左向右掃描後序運算式一次，遇見運算元即 push 入堆疊中，俟遇見運算子則 pop 出其所需的運算元數（二元運算子即是二個、單元運算子即是一個），進行計

10　後序運算式的表示法，已將運算子優先順序隱含 (embedded) 在其中。

算，再將結果 push 入堆疊中，如此即可完成求值，比起中序運算式的求值有效率許多。其時間複雜度是 $O(n)$，n 為運算式中運算子和運算元個數的和。

3.8.3 中序運算式轉為後序運算式

請觀察圖 3-22 的例子，不難發現兩者運算元出現的順序都一樣！

中序運算式：A+(B-C)*D

後序運算式：ABC-D*+

圖 3-22

事實上在範例 3-13 中，我們已經知道中序轉後序、或中序轉前序的轉換過程，只須更動運算子的位置，運算元的順序並無任何更動。

至於計算順序的決定，我們可看出一些規則，如：

- B-C 先運算乃因其在括號內；
- + 的位置在 * 後，乃因 + 的執行順序低於 *。

由此可知在中序轉後序時，中序運算式中 * 雖比 + 後看到，但要先輸出，可見我們需要一個堆疊—存放後到但先出的運算子。原則上目前遇到的運算子 x 應先 push 入堆疊中，因為 x 能否輸出，得由 x 之後的運算子來決定；而 x 進入堆疊前，則可決定 x 之前的運算子是否可輸出了。我們再把思緒整理如下：

自左向右掃描輸入中序運算式，遇見運算元可直接輸出，因運算元的順序不會更動;遇見運算子 x 應與堆疊中的頂端運算子 y 比較,若運算優先順序(x) > 運算優先順序(y)，則 x 應 push 入堆疊；若運算優先順序(x) <= 運算優先順序(y)，則 y 應先 pop 並輸出，若堆疊中仍有頂端運算子 y' 且運算優先順序(x) <= 運算優先順序(y')，則 y'亦應先 pop 輸出，直至遇見頂端運算子 y''，滿足運算優先順序(x) > 運算優先順序(y'')，x 方可 push 入堆疊中。總

而言之：運算式的任一運算子輸出前，都得先行進入堆疊；而堆疊中的運算子，唯在後來欲進堆疊的運算子優先順序較其為低時，方可確定可以輸出。

若遇到左括號 "(" ，因括號內的運算得先執行，遂應無條件 push 入堆疊中；亦即左括號此時應有最高的優先順序，但一旦進入堆疊中，為保証左括號之後的運算子可順利進入堆疊內，把進入堆疊之後之左括號的優先順序降為最低。至於若遇上右括號 ")"，則應將堆疊內的運算子皆依序 pop 輸出，直至 pop 到左括號為止，亦即此組括號內的運算子理應輸出了。茲將優先順序的定義重新整理如下 (數值愈大表示優先順序愈高)：

表 3-2 運算子的優先順序（中序轉後序用）

運算子 x	優先順序 進入堆疊前 p(x)	優先順序 進入堆疊後 q(x)
(	9	0
^ （乘方）	8	8
*、/、%	7	7
+、-	6	6
&&	2	2
||	1	1
#	–1	–1

綜合上面的討論，中序運算式轉後序運算式的演算法可撰寫如下：

演算法 3-3 中序運算式轉後序運算式：

輸入：中序運算式 e

輸出：e 之後序運算式

```
1    n = parsing(e, token);
2   push("#");
3   for (i=0; i<n; i++)
4   {   s = token[i];
5       if (s 是一運算元) output(s);
6       else if (s == ")")
7               //將堆疊中第一個"("之前的運算子皆 pop 出並印出之
8               while ((x=pop()) != "(") output(x);
9           else
10          {   while ( p(s) <= q(Stack[top] )
11              {    x = pop();
12                   output(x);
13              }
14              push(s);
15          }
16  }
17  while (Stack[top] != "#")
18  {    x = pop();
19       output(x);
20  }
21  x = pop();
22  // while (x=pop() != “#”) output(x);
```

第 1 行沿用了演算法 3-2 介紹的 parsing(e, token)，於是可知總共有 n 個運算元和運算子，分別依序存放在 token 陣列中。在第 2 行我們先行 push 一識別字元 # 且令 q(#)的優先次序爲最小者（如表 3-2 的-1），俟第 17~20 行

中輸出剩餘運算子時可資識別；第 21 行會 pop 之。第 5 行判斷目前的 s 是否為運算元，若是，可直接輸出；第 7~8 行處理 s 為右括號時的步驟：pop 堆疊中的運算子並印出，直至遇到左括號為止。第 10~13 行將堆疊頂端運算子優先順序大於或等於 s 之優先順序者，pop 並印出之；俟堆疊中頂端運算子優先順序小於 s 者了，方將 s push 入堆疊中（第 14 行）。請留意：第 22 行的寫法使 17~21 行的處理更為簡潔。

這個演算法在第 1 行中呼叫

```
n = parsing(e, token);
```

將輸入的中序運算式 e 逐一掃描過所有字母（3~16 行的迴圈），求得所有運算元和運算子，放在字元陣列 token 中，花的時間為 $O(n)$[11]，n 為運算元和運算子的總數；見到每一運算元即印出，只花 $O(1)$ 的時間；每個運算子皆在堆疊中 push 和 pop 一次，花的時間亦為 $O(1)$，於是所有的執行時間為 $O(n)$ 。

我們用範例 3-16 說明整個中序轉後序的過程。

範例 3-16

若有一中序運算式：

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

則圖 3-23 顯示轉換成其後序運算式

AB/CD+E*-AC*+

的過程。在程式開始執行前，我們先在堆疊中 push 識別字元 #。

11 在此我們假設輸入式的變數（運算元）皆為單一字元，若允許變數為多字元者，則時間複雜度亦為該輸入式的字元總數。

A / B - (C + D) * E + A * C	執行動作	堆疊	輸出
↑	印出 A	#	A
↑	p(/)>q(#); push(/)	/ #	
↑	印出 B	/ #	AB

↑	p(-)<q(/);
	x = pop();
	印出 x;
	push(-);
		-
		#
			AB/

↑	p(()>q(-) push(()	(- #	
↑	印出 C	(- #	AB/C
↑	p(+)>q(() push(+)	+ (- #	
↑	印出 D	+ (- #	AB/CD
↑	while Stack[top] !=”(” { x=pop(); 印出 x; }	- #	AB/CD+

A	/	B	-	(	C	+	D	)	*	E	+	A	*	C	執行動作	堆疊	輸出
									↑						p(*)>q(-) push(*)	* - #	
										↑					印出 E	* - #	AB/CD+E
											↑				p(+)<q(*) x = pop(); 印出 x	- #	AB/CD+E*
											↑				p(+)=q(-); x=pop(); 印出 x;	#	AB/CD+E*-
											↑				p(+)>q(#) push(+);	+ #	
												↑			印出 A	+ #	AB/CD+E*-A
													↑		p(*)>q(+); push(*);	* + #	
														↑	印出 C	* + #	AB/CD+E*-AC
														↑	x=pop(); 印出 x;	+ #	AB/CD+E*-AC*
														↑	x=pop(); 印出 x;	#	AB/CD+E*-AC*+

圖 3-23　中序轉後序的過程

3.8.4 中序運算式轉為前序運算式

觀察圖 3-24 的例子，不難發現中序式和前序式運算元出現的順序也一樣，而運算元對應的運算子出現在其之前！

中序運算式：A+(B-C)*D

前序運算式：+A*-BCD

圖 3-24

運用 3.8.3 節中序轉後序的經驗，不難設計出中序轉前序的演算法。我們利用一個堆疊存放運算子（並依其運算順序決定其是否該進入或離開），利用另一個堆疊存放運算元：亦即堆疊 1 存放運算子，而堆疊 2 存放運算元。一旦運算子由堆疊 1 pop 出，就自堆疊 2 中 pop 出其對應的運算元（他們肯定位於堆疊頂端，請想想其原因）。

演算法 3-4 描述了中序運算式轉前序運算式的演算法：

演算法 3-4 中序運算式轉前序運算式

輸入：中序運算式 e

輸出：e 的前序運算法

```
n = parsing(e, token);
push("#");
for (i=0; i<n; i++)
{   s = token[i];
    if (s 是一運算元) push_opn(s);
    else if (s == ")")
            //將堆疊中第一個"("之前的運算子皆 pop 並印出
            while((x=pop())!="(") push_opn(get_prefix(x));
        else
```

```
10          {   while ( p(s) <= q(Stack[top]) )
11              {   x = pop();
12                  push_opn(get_prefix(x));
13              }
14              push(s);
15          }
16  }
17  while (Stack[top]!="#")
18  {   x = pop();
19      push_opn(get_prefix(x))
20  }
21  x = pop();// pop out "#"
22  // while (x=pop() != "#") output(x);
23  return pop_opn();
24  //下方為結合出前序的程序，String 用來宣告字串型態變數，不同 C 的工具可能不同
25  String get_prefix(String x)
26  {  String a = pop_opn();
27     return x+pop_opn()+a;
28  } //在此 + 爲字串串接的運算
```

演算法 3-4 與 3-3 的流程頗爲相似，演算法 3-4 中的 push(s)、pop() 是堆疊 1 的運算（處理運算子）；而 push_opn(s)、pop_opn() 是堆疊 2 的運算（處理運算元）。25~28 行定義了 get_prefix(String x) 程序，它依據傳入的字串參數 x （運算子），取得其對應的運算元（在堆疊 2 中—在此假設 x 必爲二元運算子；若要處理單元運算子，請自行修繕），結合出其前序運算式（字串）後回傳之。有關 String 和字串+的說明，請見第 24、28 行。

範例 3-17 舉例說明整個中序轉前序的過程。

範例 3-17

若有一中序運算式：

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

則其前序運算式為：

+-/AB*+CDE*AC

圖 3-25 顯示其轉換過程。

A	/	B	-	(	C	+	D	)	*	E	+	A	*	C	執行動作	堆疊 1	堆疊 2
↑															push_opn(A)	#	A
	↑														p(/)>q(#); push(/)	/ #	
		↑													push_opn(B)	/ #	B A
			↑												p(-)<q(/); x = pop(); push_opn(/AB); push(-);	- #	/AB
				↑											p(()>q(-) push(()	(- #	/AB
					↑										push_opn(C)	(- #	C /AB
						↑									p(+)>q(() push(+)	+ (- #	C /AB

A / B - (C + D) * E + A * C	執行動作	堆疊 1	堆疊 2
↑	push_opn(D)	+ (- #	D C /AB
↑	while Stack[top]!=”(” { push_opn(+CD); }	- #	+CD /AB
↑	p(*)>q(-) push(*)	* - #	+CD /AB
↑	push_opn(E)	* - #	E +CD /AB
↑	p(+)<q(*) x = pop(); push_opn(*+CDE) ;	- #	*+CDE /AB

↑	p(+)=q(-);
	x=pop();
	push_opn(-/AB*+
	CDE);
		#	-/AB*+CDE

↑	p(+)>q(#) push(+);	+ #	-/AB*+CDE
↑	push_opn(A);	+ #	A -/AB*+CDE
↑	p(*)>q(+); push(*);	* + #	A -/AB*+CDE

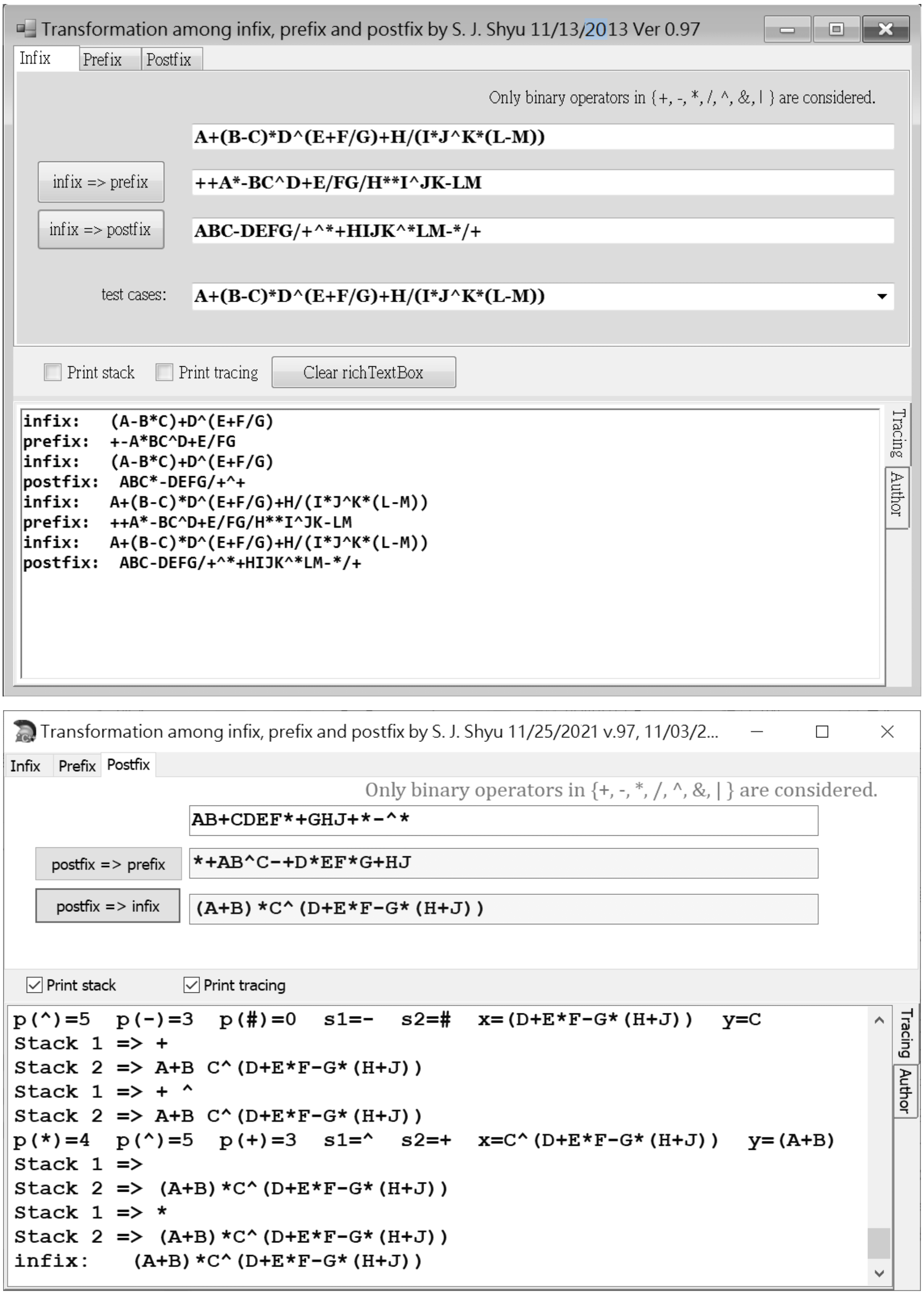
A / B - (C + D) * E + A * C	執行動作	堆疊 1	堆疊 2
	↑ push_opn(C)	* + #	C A -/AB*+CDE
	↑ x=pop(); push_opn(*AC);	+ #	*AC -/AB*+CDE
	x=pop(); ↑ push_opn(+-/AB* +CDE*AC);	#	+-/AB*+CD E*AC
	x=pop(); return ↑ pop_opn();		

圖 3-25 中序轉前序的過程

範例 3-18 顯示了實作演算法 3-3 和 3-4 中序、後序和前序式互相轉換的程式畫面。

範例 3-18

圖 3-26 (a) 和 (b) 是個在個人電腦視窗作業系統中，分別採用 Visual Studio C++ 和 C++ Builder 實作中序、前序和後序式互相轉換的程式執行結果；其中 (a) 輸入中序式，轉為對應的前序和後序式，(b) 輸入輸出後序式，輸出其前序和中序式（包含了必要的括號），以及過程中兩個堆疊（分別供運算元、運算子使用）的內容。



(a)
(b)

圖 3-26　中序、前序和後序運算式互相轉換的實作程式畫面

本章習題

1. 圖 3-27 描繪了一個鐵路調度軌道，每一節車廂都編號 1, 2, 3, ..., n，並按順序從左到右停放在軌道上，車廂可以從任何一條橫向軌道一次一台的移進垂直軌道，而移進垂直軌道的車廂也可以一次一台的移到任何一條橫向軌道，則垂直軌道就像一個堆疊，新移進車廂都在最上層，能移出的車廂也是最上層的那一個。假設 $n = 3$ 時，我們可以先移車廂 1 進垂直軌道，再來是車廂 2，最後是車廂 3，然後我們就得到一個新的順序 3, 2, 1；當 $n = 3$ 和 4 時，可以得到哪幾種有可能的車廂排列？有哪幾種排列是不可能的？

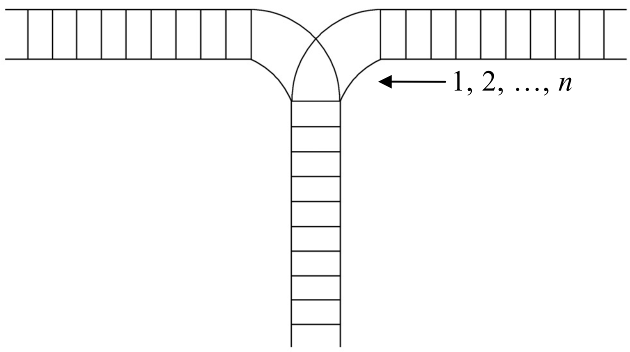


圖 3-27　鐵路網

2. 對一堆疊依序加入 (push) 1, 2, 3, 4, 5，其間可輸出 (pop) 元素，請列出所有可能的輸出？亦請列出所有不可能的輸出。

3. 一個「雙端佇列」（double-ended queue，也稱 deque）是一個線性串列，它可以在佇列任一端加入或刪除元素。設計一個表示法，使用一維陣列來表示雙端佇列，再寫出從任一端加入或刪除雙向佇列的演算法。

4. 用陣列實作一個線性串列，使用兩個變數 *front* 和 *rear*，使其變成「環狀」。

(a) 使用 *front*, *rear* 和 n，設計一個公式來求出串列中元素的個數。

(b) 寫一個演算法來刪除串列中第 k 個元素。

(c) 寫一個演算法來立即插入元素 y 在第 k 個元素後面。

你設計的演算法 (b) 和 (c)，其時間複雜度爲多少?

5. 一個 $m \times p$ 的迷宮，全部有可能的路徑中，最長的長度爲多少？

6. 判斷以下的數學形式爲何種形式（即中序、前序或後序），然後再轉換成另外兩種：

   (a) `1+2-3×4/5×6/7-8/9`

   (b) `+×-×123×45/×678`

7. 寫出下列式子的後置式：

   (a) `A×B×C`

   (b) `-A+B-C+D`

   (c) `A×-B+C`

   (d) `(A+B)×D+E/(F+A×D)+C`

   (e) `A&&B||C||!(E>F)`

   (f) `!(A&&!((B<C)||(C>D)))||(C<E)`

   (g) `A and B or C or not (E>F)`

   (h) `(A+B)*C^(D+E*F-G*(H+J))`

8. 使用表 3-1 和 3-2 各運算子和其優先次序，再加上 "("、")" 和 "#"，來回答下面的問題：

   (a) 在演算法 3-2 中，假如一個運算式 e 內含有 n 個運算子和標點符號，則堆疊中最多會放進多少個元素？

   (b) 假如運算式 e 有 n 個運算子，且括號最深有 6 層(如：(..(..)..) + (..(..(..)..)..) 此式括號最深就是 3 層），則 (a) 的答案會是多少？

9. 另外一種容易計算且不必加括號式子的表示法就是前序表示，這種表示法就是將運算子放在運算元的前面，請見範例 3-13：

   (a) 將習題 6 表示成對應的前序表示。

   (b) 寫一個演算法來計算前置表示法 e。

   (c) 寫一個演算法把中置式 e 轉成相等的前置式，假設每個 e 都以符號 "#" 做開頭，且其前置式的也要以 "#" 做開頭。

   你的演算法 (b) 和 (c) 的複雜度爲多少？每個演算法需要多少空間？

10. 寫一個演算法將前置式轉成後置式，詳細描述任何有關輸入式子的假設。你的演算法需要多少時間和空間？

11. 有兩個堆疊用這節討論的方法儲存在陣列 $M[m]$，寫一個演算法 `Add` 和 `Delete` 在堆疊 i 中，$0 \le i \le 1$，做加入和刪除的動作，你的演算法必須保證只要兩個堆疊的元素總和小於 m 就能加入元素，而且執行時間必須在 $O(1)$ 內。

12. 設計一個資料表示法，將 n 個佇列循序對應到陣列 $M[m]$；在 M 中每個佇列就像是環形佇列，爲此表示法寫出函式 `Add`、`Delete` 和 `QueueFull`。

13. 設計一個資料結構，將 n 個資料物件連續對應到陣列 $M[m]$，其中 n_1 個物件爲堆疊，剩下的 $n_2 = n-n_1$ 爲佇列。寫出加入和刪除的演算法，只要陣列

中還有空間沒使用到，此演算法就要提供空間給第 i 個資料物件。注意：一個有 r 個空間的環形佇列只能儲存 r–1 個元素。

14. 下列是 Hanoi 塔 (Hanoi tower) 的遞迴演算法：

演算法 3-5 Hanoi 塔

```
輸入：整數 N：大小不一（1~N 由小至大排列）的盤子數，A:起點、B：
      終點、C：輔助點
輸出：將 N 個盤子從 A 藉助 C，搬到 B；在 A、B 和 C 任一點上，大盤
      子皆得在小盤子之下
Tower(N, A, B, C)
{  if (N>0)
   {   Tower(N-1, A, C, B);
       搬第 N 個盤子，從 A 搬到 B;
       Tower(N-1, C, B, A);
   }
}
```

請將上述遞迴演算法改用非遞迴方式。