第四章

堆疊與佇列

Stacks and Queues

本章內容

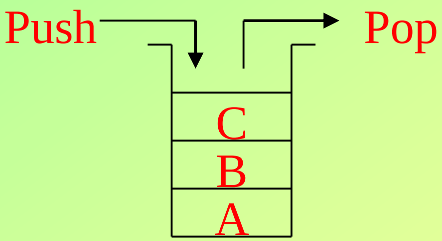
在**尋寶遊戲**中，在取得寶物後，必須沿著相反的路徑回頭。遊戲程式設計者是如何安排你走正確的關卡呢？

答案是使用**「堆疊」(stack)**。遊戲程式設計者將你沿路前進時所經過的關卡放入堆疊中，在你回基地的過程中，再從堆疊中逐一取出正確 的關卡，這樣就不會亂了套而迷路。

4-1 堆疊（stack）

➢ 將一個項目放入堆疊的頂端，此動作稱作 **推入** (Push)

從堆疊頂端拿走一個項目，此動作稱作 **彈出** (Pop)



➢ 堆疊在生活中的例子

集銅板的錢包、裝子彈的彈匣 、自助餐廳的盤疊

後進先出 (*last in first out*, **LIFO**) 是堆疊最重要的特性

➢ 有幾個運算可以作用在堆疊這種結構上面：

1. Push ：將新項目加在堆疊的頂端。
2. Pop ：從堆疊頂端拿走一個項目。
3. TopItem ：看看堆疊頂端的項目為何，（項目並不會被彈出）。
4. IsEmpty ：看堆疊是否為空，空則傳回真，不空則傳回偽。
5. IsFull ：看堆疊是否已滿，滿則傳回真，未滿則傳回偽

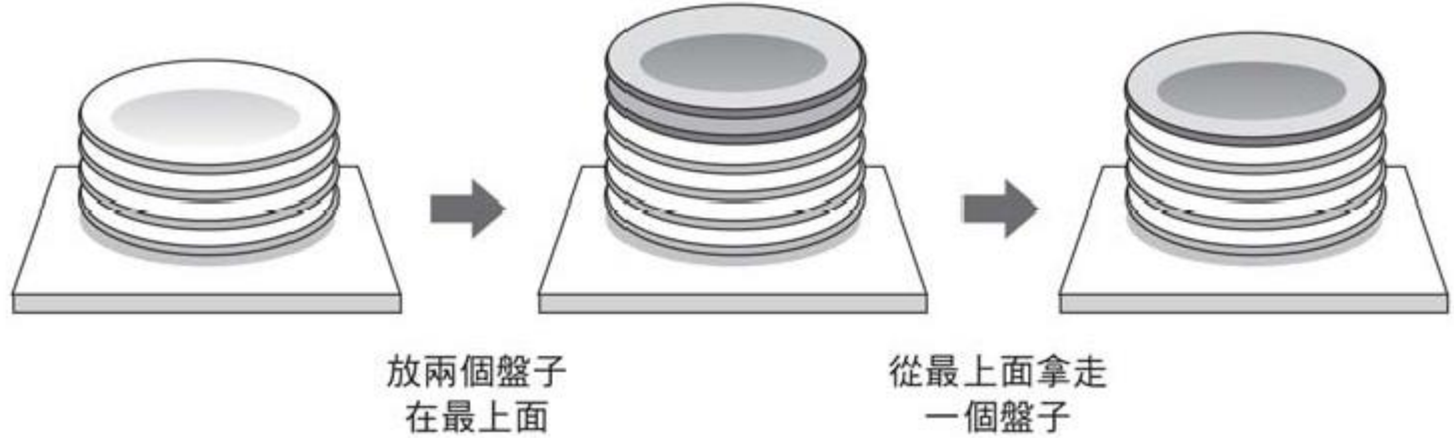


圖 4.1　堆盤子與拿盤子

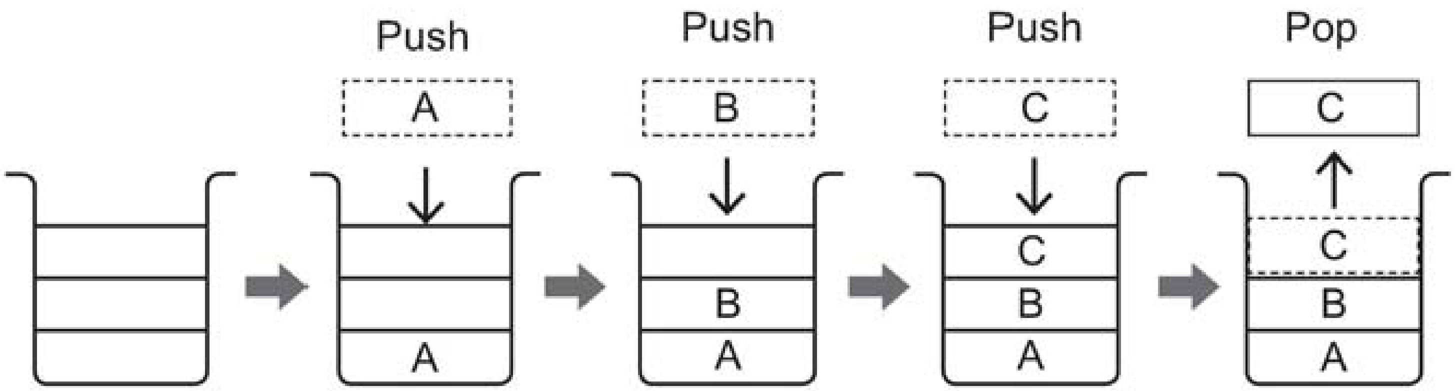


圖 4.2　堆疊的示意圖

4-2 用陣列結構實作堆疊

```
#define  MAX_ITEM      5
typedef  struct    tagSTACK
   {    char        Item[MAX_ITEM] ;
        int                 Top ;
   } STACK;
STACK  S;
```

- S.Item 儲存堆疊資料
- S.Top紀錄目前頂端元素所在的位置

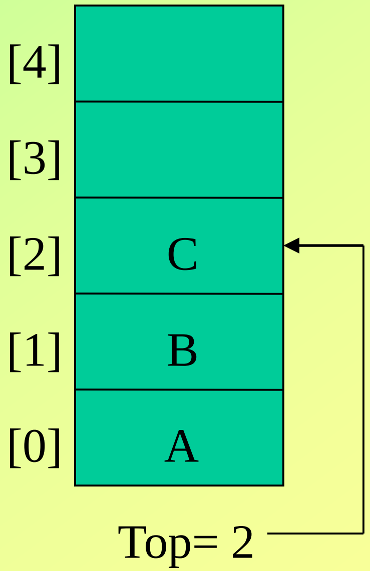
當堆疊為空時，S.Top == -1

當堆疊為滿時，S.Top == MAX_ITEM-1

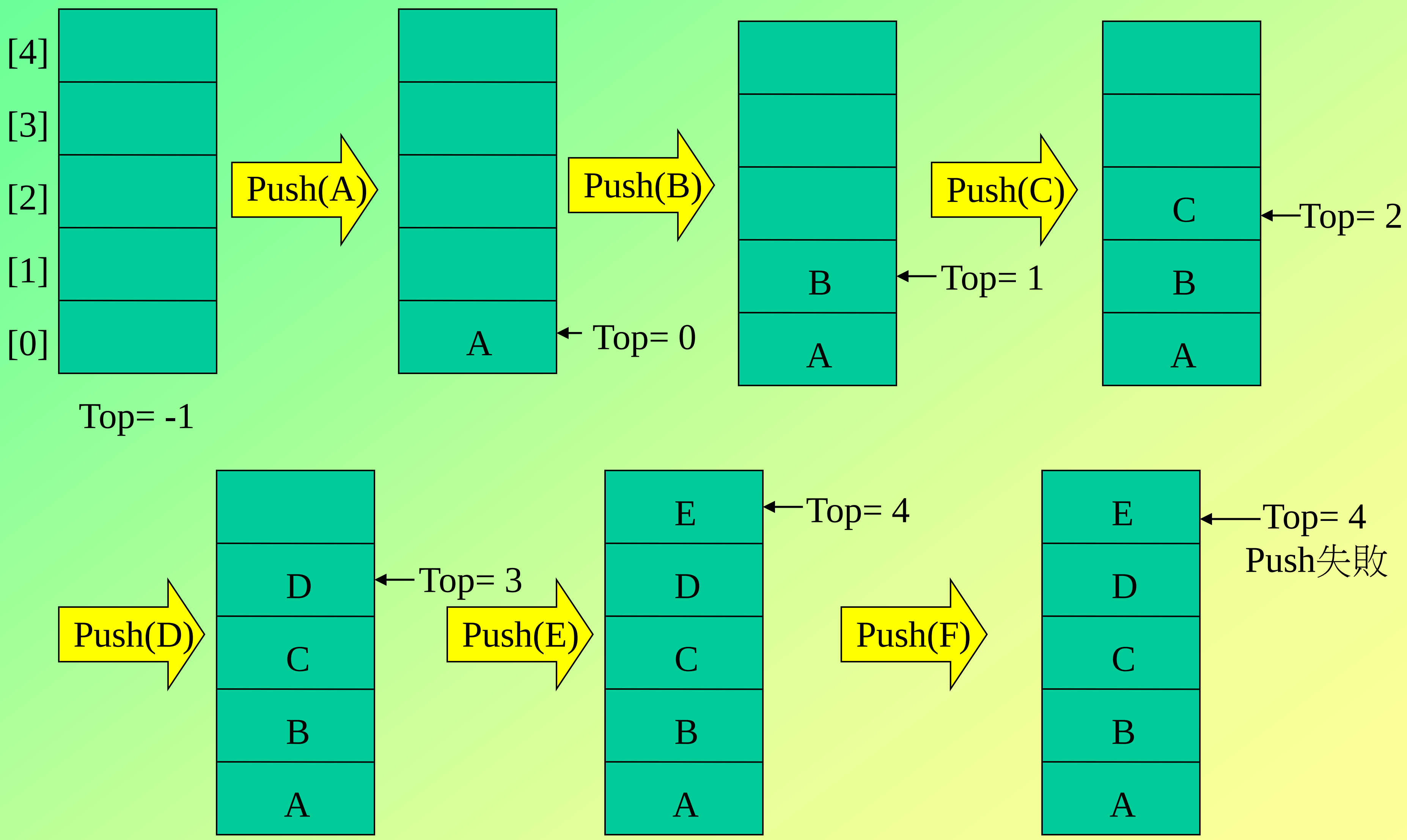
右圖堆疊中共有 3 個元素

C 為頂端元素

注意堆疊陣列倒著畫，比較符合堆疊習慣



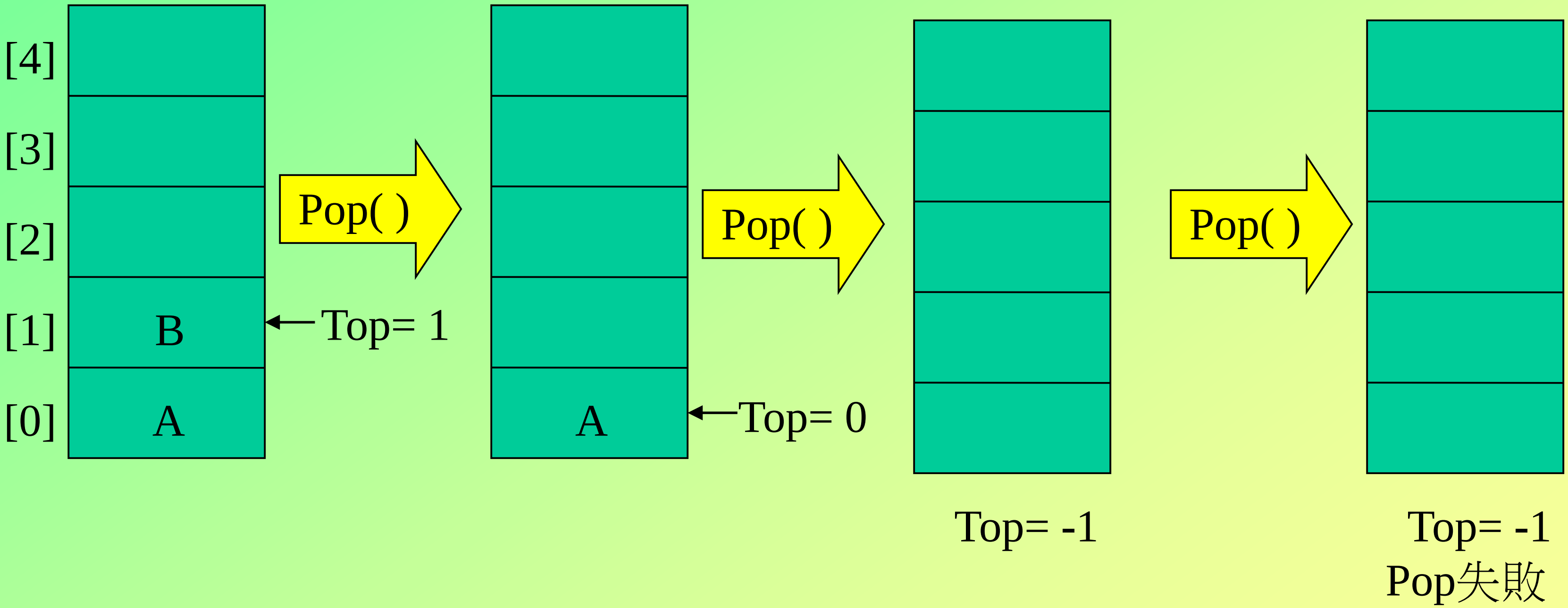
堆疊的資料推入



[4]
[3]
[2]
[1]
[0]
Top= -1
Push(A)
A
Top= 0
Push(B)
B
A
Top= 1
Push(C)
C
B
A
Top= 2
Push(D)
D
C
B
A
Top= 3
Push(E)
E
D
C
B
A
Top= 4
Push(F)
E
D
C
B
A
Top= 4
Push失敗

堆疊已滿卻仍要 Push，這個狀況稱為
「溢位」(overflow)，
這是與堆疊相關的一個重要名詞。

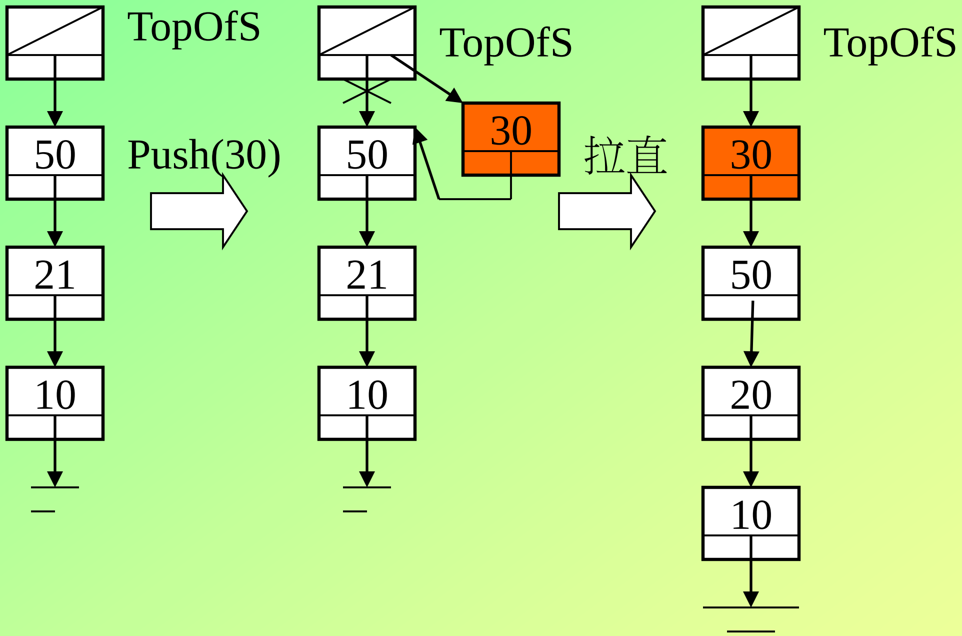
堆疊的資料彈出



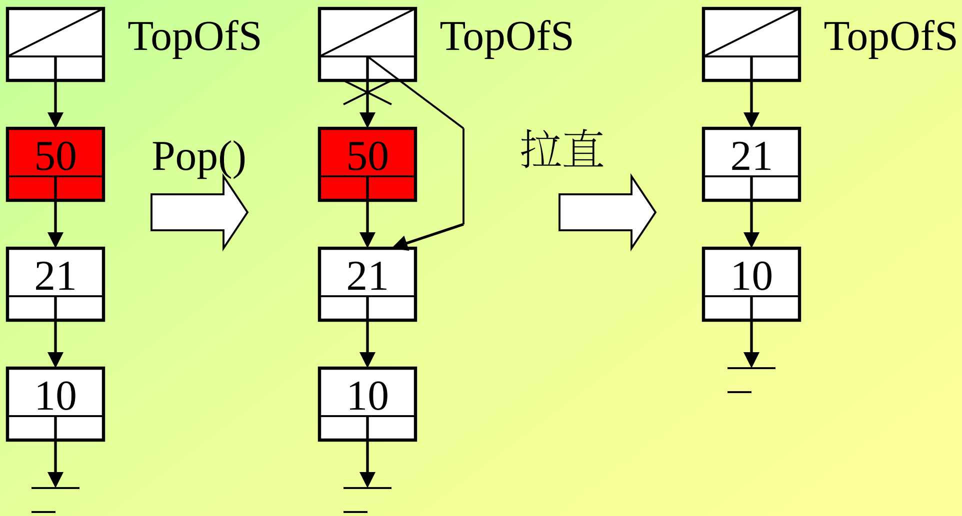
學到這裏，想一想如何使用堆疊將英文字 "data" 翻轉成 "atad"？

4-3 用鏈結串列實作堆疊

Push 相當於在串列的前端插入新節點



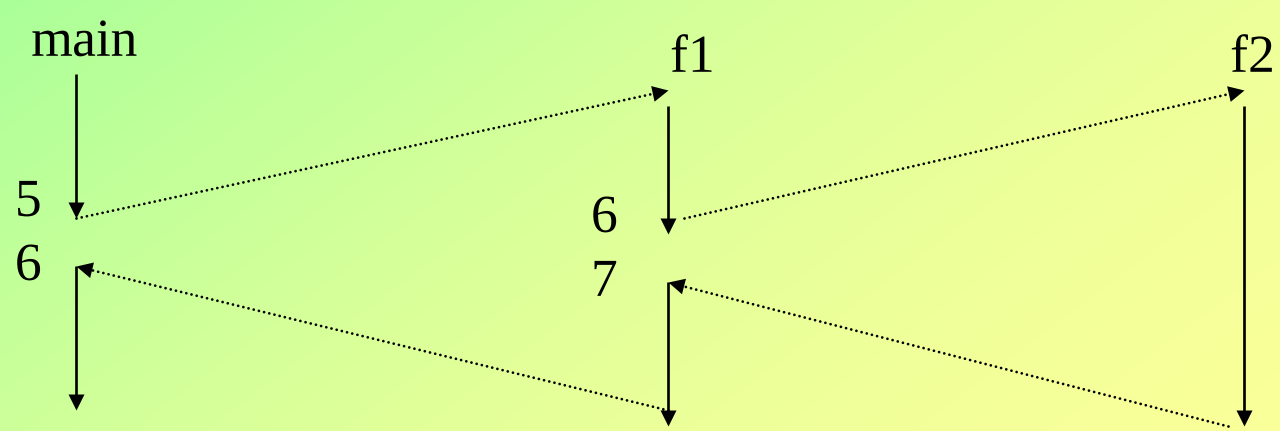
Pop 相當於刪除在串列前端的節點



4-4 堆疊的應用

副程式的呼叫與返回

1. main() 2.{ 3. int i, j ; 4. printf(“main before calling f1\n”); 5. f1(); 6. printf(“main returns from f1\n”); 7. }	1.void f1() 2.{ 3. int m, n ; 4. m = 2 ; 5. printf(“f1 before calling f2\n”); 6. f2(m); 7. printf(“f1 returns from f2\n”); 8. }	1.void f2(int k) 2.{ 3. printf(“f2 no more calling \n”); 4. k++ ; 5.}



運算式的轉換

運算元(變數、文數字)

X = A + B * C – (D + E / F)

等號右邊是算術運算式

運算子(運算符號)

運算式根據運算子的位置可以分為三種：

1. 中序式 (infix)：運算子在運算元的中間，例如　：　A + B
2. 後序式 (postfix)：運算子在運算元的後面，例如 ：A　B +
3. 前序式 (prefix)：運算子在運算元的前面，例如　：　+ A B

後序式記法又稱為「逆波蘭記法」(Reverse Polish Notation , RPN)，是計算機科學中極為常見的表示法。在許多系統程式中，如作業系統、編譯器等，都會先將中序式轉為後序式再加以運算，因為計算後序式的效率比較好。

中序式轉後序式(括號法)

(A + B) * C – D / E　　　轉後序式

第一步：依照運算子的優先順序，將每個運算子和相關的運算元用括號括起來

((A + B) * C) – D / E　　　(對 * 號加括號)

((A + B) * C) – (D / E)　　(對 / 號加括號)

(((A + B) * C) – (D / E))　(對 - 號加括號)

第二步：對每個運算子找到在它右邊最接近而且未配對的右括號，加上箭頭

(((A + B) * C) – (D / E))

第三步：將運算子移到箭頭所指到的地方，並且去掉所有的括號

A B + C * D E / –

中序式轉前序式

(A + B) * C – D / E　　　轉前序式

第一步：依照運算子的優先順序，將每個運算子和相關的運算元用括號括起來

((A + B) * C) – D / E　　　(對 * 號加括號)

((A + B) * C) – (D / E)　　(對 / 號加括號)

(((A + B) * C) – (D / E))　(對 - 號加括號)

第二步：對每個運算子找到在它左邊最接近而且未配對的左括號，加上箭頭

(((A + B) * C) – (D / E))

第三步：將運算子移到箭頭所指到的地方，並且去掉所有的括號

– * + A B C / D E

例4.3 中序式轉後序式(堆疊法)

運算法原則：

1. 由左而右讀出中序式，一次讀取一個句元 (token，一個運算元或運算子)。
2. 如果是運算元，直接輸出到後序式。
3. 如果是運算子，則必須用到運算子堆疊。
4. 中序式讀完之後，如果運算子堆疊不是空的，則一個一個Pop出來，輸出到後序式。

運算子堆疊使用規則：

1. **左括號 ‘(’** ：一律Push。
2. **右括號 ‘)’** ：一直做Pop輸出至後序式，直到遇見左括號，一同抵銷
3. **其他的運算子**：運算子在堆疊中只能優先序大的壓優先序小的。也就是當運算子在進入堆疊之前，必須和堆疊頂端的運算子比較優先序。如果外面的運算子優先序較大，則Push。否則就一直做Pop，直到遇見優先序較小的運算子或堆疊為空，再把外面的運算子Push。值得注意的是：左括號在堆疊中優先序最小，亦即任何運算子都可以壓他。

處理之句元	處理動作	運算子堆疊	後序式
1. 運算子 ‘(’	①規則3. 必須使用堆疊 ②堆疊規則1. 直接Push	(	
2. 運算元 ‘A’	①規則2. 直接輸出至後序式	(	A
3. 運算子 ‘+’	①規則3. 必須使用堆疊 ②堆疊規則3. ‘+’ 的優先序比 ‘(‘ 大，故Push ‘+’	+ (	A
4. 運算元 ‘B’	①規則2. 直接輸出至後序式	+ (	AB
5. 運算子 ‘)’	①規則3. 必須使用堆疊 ②堆疊規則2. 一直Pop，直到遇見 ‘(‘，兩者一同抵銷		
6. 運算子 ‘*’	①規則3. 必須使用堆疊 ②堆疊規則3. Push，因為堆疊為空	*	
7. 運算元 ‘C’	①規則2. 直接輸出至後序式	*	AB+C
8. 運算子 ‘–’	①規則3. 必須使用堆疊 ②堆疊規則3. ‘–’ 的優先序比 ‘*’ 小，故Pop得到 ‘*’ 並輸出到後序式。接著因堆疊為空故Push ‘–’	–	AB+C*
9. 運算元 ‘D’	①規則2. 直接輸出至後序式	–	AB+C*D
10. 運算子 ‘/’	①規則3. 必須使用堆疊 ②堆疊規則3. ‘/’ 的優先序比 ‘–’ 大，故Push ‘/’	/ –	AB+C*D
11. 運算元 ‘E’	①規則2. 直接輸出至後序式	/ –	AB+C*D E
	①規則4. 中序式已經讀完，因此將堆疊中剩餘的運算子依序Pop到後序式		AB+C*D E/–

(A + B) * C – D / E

運算法原則：

1. 由左而右讀出中序式，一次讀取一個句元 (token)。
2. 如果是運算元，直接輸出到後序式。
3. 如果是運算子，則必須用到運算子堆疊。
4. 中序式讀完之後，如果運算子堆疊不是空的，則一個一個Pop出來，輸出到後序式。

運算子堆疊使用規則：

1. **左括號 ‘(’** ：一律Push。
2. **右括號 ‘)’** ：一直做Pop輸出至後序式，直到遇見左括號，一同抵銷
3. **其他的運算子**：運算子在堆疊中只能優先序大的壓優先序小的。左括號在堆疊中優先序最小，亦即任何運算子都可以壓他。

後序式求值

後序式A B + C * D E / –（假設運算元的值分別是2, 5, 4, 6, 3）：中序式 (A + B) * C – D / E=26

處理句元	處理動作	運算元堆疊
1. 運算元 'A'	規則2. 直接Push	A (=2)
2. 運算元 'B'	規則2. 直接Push	B (=5) A (=2)
3. 運算子 '+'	規則3. 作兩次Pop，先Pop出來的B為第二運算元，後Pop出來的A為第一運算元，執行A+B，設為X，Push(X)	X (=7)
4. 運算元 'C'	規則2. 直接Push	C (=4) X (=7)
5. 運算子 '*'	規則3. 作兩次Pop，先Pop出來的C為第二運算元，後Pop出來的X為第一運算元，執行X*C，設為Y，Push(Y)	Y(=28)
6. 運算元 'D'	規則2. 直接Push	D (=6) Y (=28)
7. 運算元 'E'	規則2. 直接Push	E (=3) D (=6) Y (=28)
8. 運算子 '/'	規則3. 作兩次Pop，先Pop出來的E為第二運算元，後Pop出來的D為第一運算元，執行D/E，設為Z，Push(Z)	Z(=2) Y (=28)
9. 運算子 '–'	規則3.作兩次Pop，先Pop出來的Z為第二運算元，後Pop出來的Y為第一運算元，執行Y–Z，設為W，Push(W)	W(=26)
	規則4. 後序式已經讀完，堆疊只剩一個元素，即為結果W(=26)	

規則：

1. 由左而右讀出後序式，一次讀取一個句元 (token)。
2. 如果是**運算元**，一律Push入運算元堆疊。
3. 如果是**運算子**，則作兩次Pop，第一次Pop出來的單元是第二運算元，第二次Pop出來的則是第一運算元。第一運算元和第二運算元根據運算子作適當的運算，結果再Push回運算元堆疊。
4. 最後的結果會在堆疊的頂端。

4-5 佇列(queue)

◎ 佇列：等待接受服務或是取得資源的隊伍

◎ 新加入者從隊伍的**後端** (rear) 進入佇列。買完票者從隊伍的**前端** (front) 離開佇列（當然排在隊伍前端的人才有資格先買票）

◎ 先離開佇列的，一定是隊伍中先進來的。這種**先進先出** (first in first out, **FIFO**) 的現象，正是佇列最重要的特性

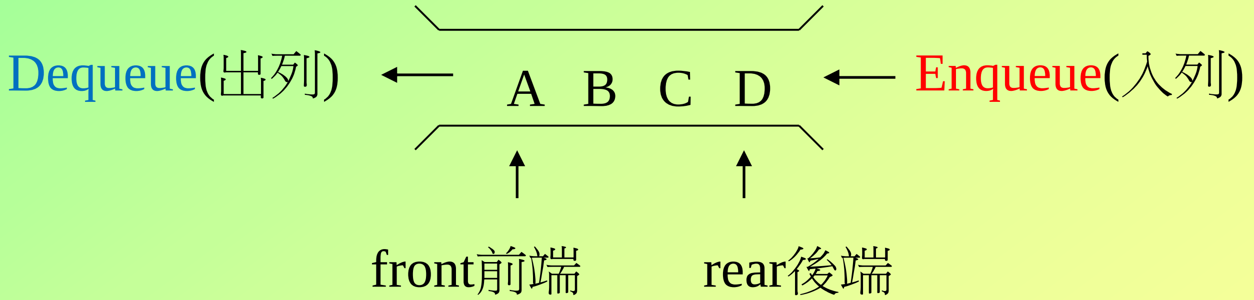
有幾種運算可以作用在佇列這種抽象結構上面：

1. Enqueue (Add)：把新項目由後端 (rear) 加入佇列（加入隊伍）

2. **Dequeue** (Delete)：從佇列的前端 (front) 刪除一個項目（離開隊伍）

3. IsFull ：測試佇列是否已滿，已滿為真 (TRUE)，未滿為偽 (FALSE)

4. IsEmpty：測試佇列是否為空，空時為真，非空時為偽



4-6 用陣列結構建置佇列

〈空佇列 , rear = front = -1〉

Enqueue(A)

[0]	[1]	[2]	[3]	[4]	[5]	[6]
A						

front = -1 , rear = 0 (rear++)

Enqueue(B)

[0]	[1]	[2]	[3]	[4]	[5]	[6]
A	B					

front = -1 , rear = 1 (rear++)

Dequeue()

[0]	[1]	[2]	[3]	[4]	[5]	[6]
	B					

front = 0 , rear = 1 (front++)

Enqueue(C~G)

[0]	[1]	[2]	[3]	[4]	[5]	[6]
	B	C	D	E	F	G

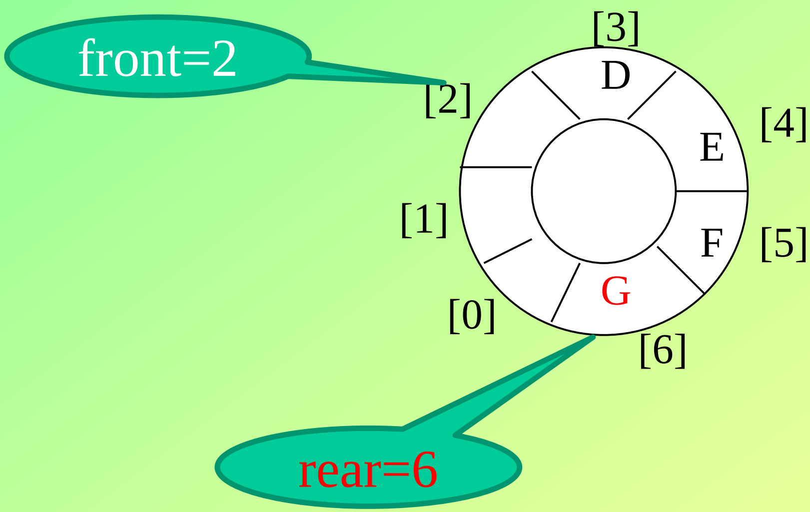
front = 0 , rear = 6

Enqueue(H) 失敗，因 rear 加 1 會超出陣列範圍

陣列前面其實還有空的位置卻不能利用的情形非常不合理。
解決的方法之一：每次Dequeue之後，就把Queue中每個項目都往左挪一格

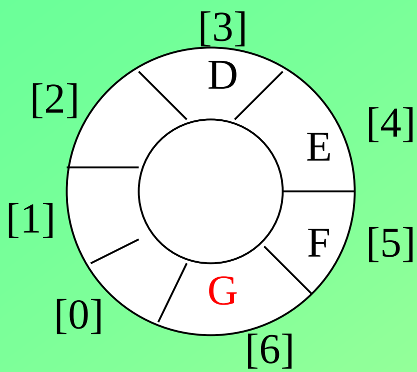
解決的方法之二：環狀佇列—解決空位無法使用的問題

1. rear指向排尾，front 指向排頭的前一個空位。
2. 當rear = front時為空佇列(初設 rear = front = 0) 。
3. 當 (rear + 1) % MAX_ITEM = front時
（當rear順時針推進一格會碰到front時）為滿佇列。
4. Enqueue時，rear 順時針推進一格
5. Dequeue時，front 順時針推進一格



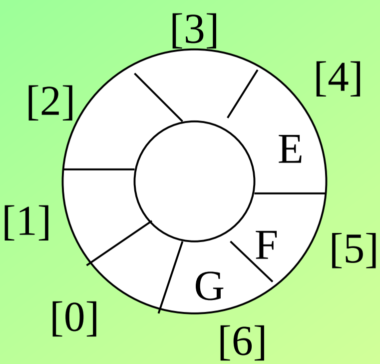
MAX_ITEM = 7 (7個位置)

front = 2，排頭的前一個空位
rear = 6，排尾



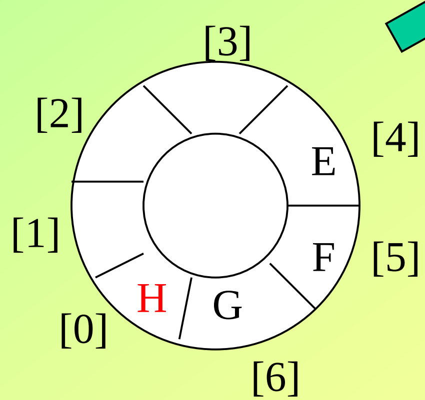
front = 2
rear = 6

Dequeue ( )



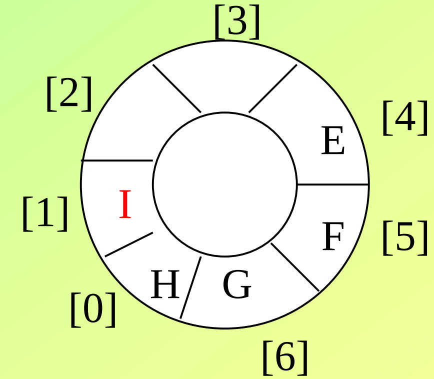
front = 3 -> (2+1)%7 = 3
rear = 6

Enqueue (H)



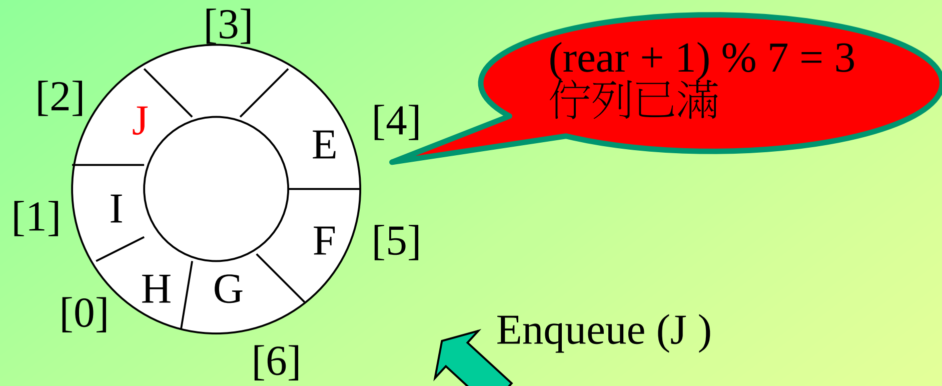
front=3
rear = 0 -> (6+1)%7 = 0

Enqueue (I)



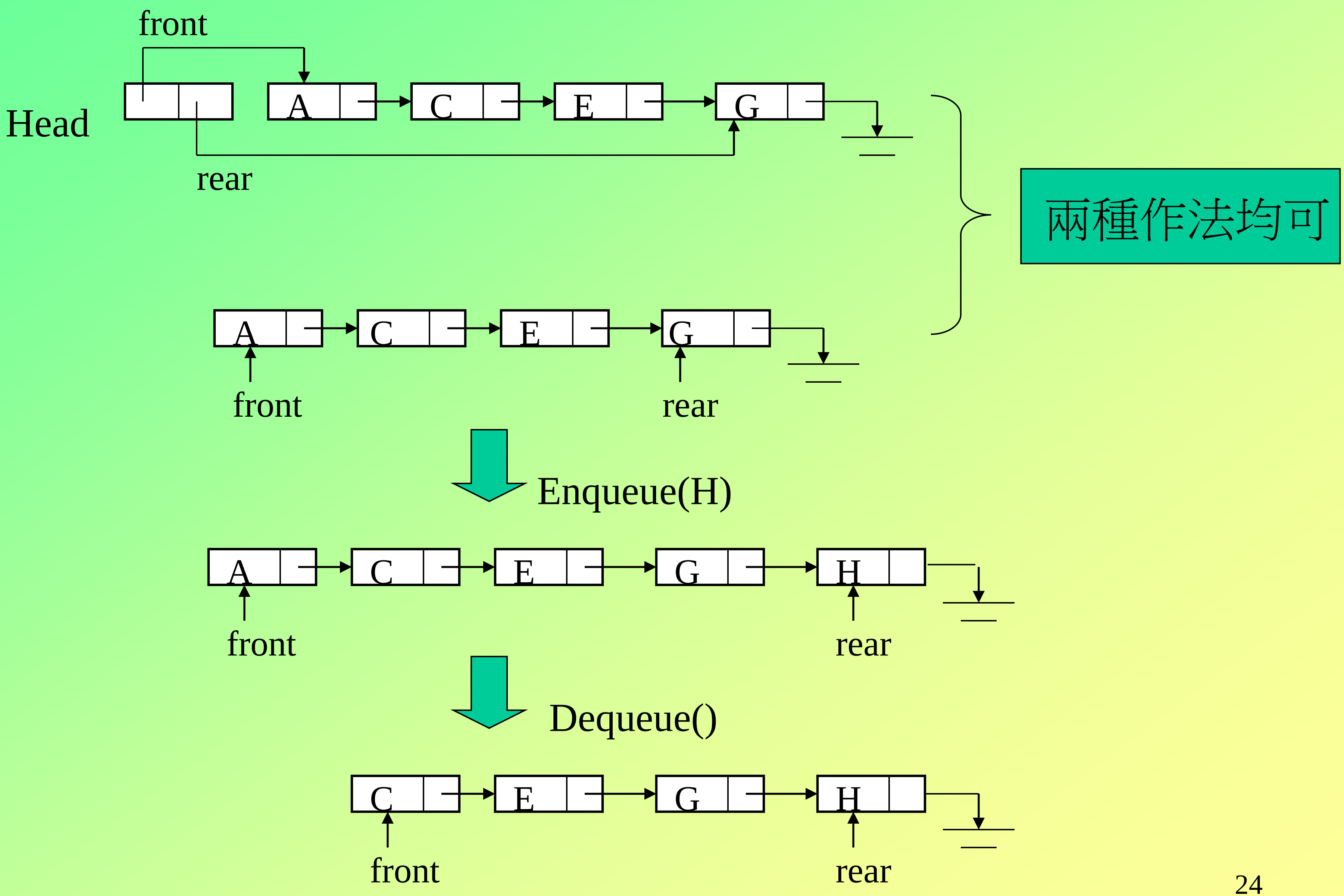
front=3
rear = 1 ->
(0+1)%7 = 1

Enqueue (J)



front=3
rear = 2 ->
(1+1)%7 = 2

4-7 用鏈結串列實作佇列



4-8 遞迴　　　階乘的遞迴呼叫

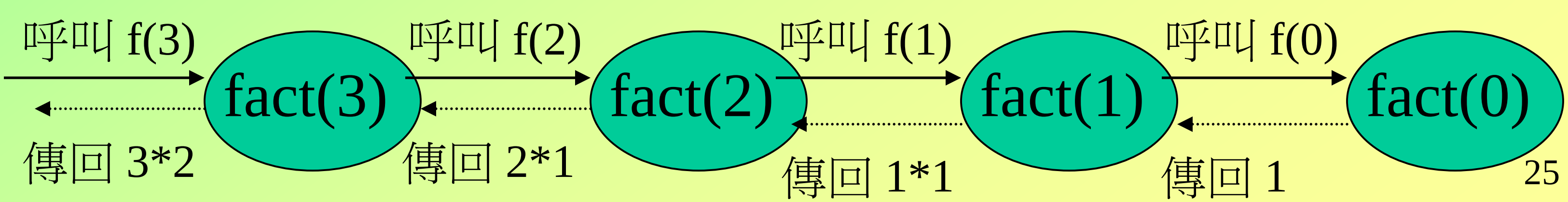
◎定義

$$
n! = \begin{cases} 1, & \text{if } n=0 \\ n * (n-1)!, & \text{if } n>0 \end{cases}
$$

◎C語言函式

```
int  fact( int n)
{           int         f;
            if  (n==0) return 1; //終止條件
            f = fact(n-1);         //遞迴呼叫
            return ( n * f );
}
```

◎呼叫 fact(3)的圖示



Fibonacci數列的遞迴函式

◎定義

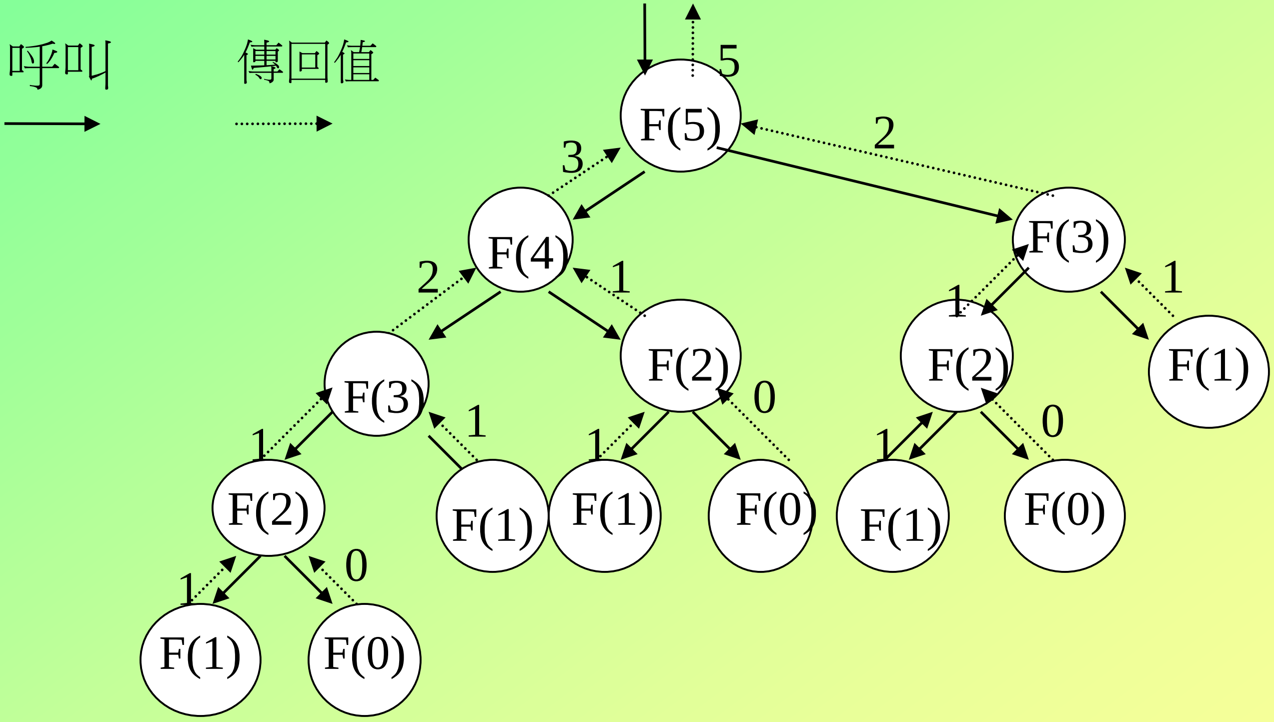
$$Fib(n) = \begin{cases} 0, & \text{若 } n = 0 \\ 1, & \text{若 } n = 1 \\ Fib(n-1) + Fib(n-2), & \text{若 } n >= 2 \end{cases}$$

0	1	1	2	3	5	8	13
Fib(0)	Fib(1)	Fib(2)	Fib(3)	Fib(4)	Fib(5)	Fib(6)	Fib(7)

◎C語言函式

```
int  Fib ( int n )
{          int  i, j ;
           if ( n == 0 )
                 return ( 0 ) ; //終止條件：n=0
           if ( n == 1 )
                 return ( 1 ) ; //終止條件：n=1
           i = Fib(n-1);
           j = Fib(n-2);
           return ( i + j );
}
```

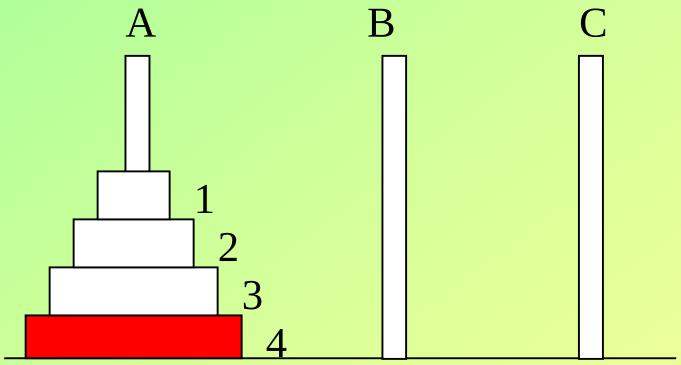
◎呼叫Fib(5)的圖示



河內塔 (Hanoi Tower)

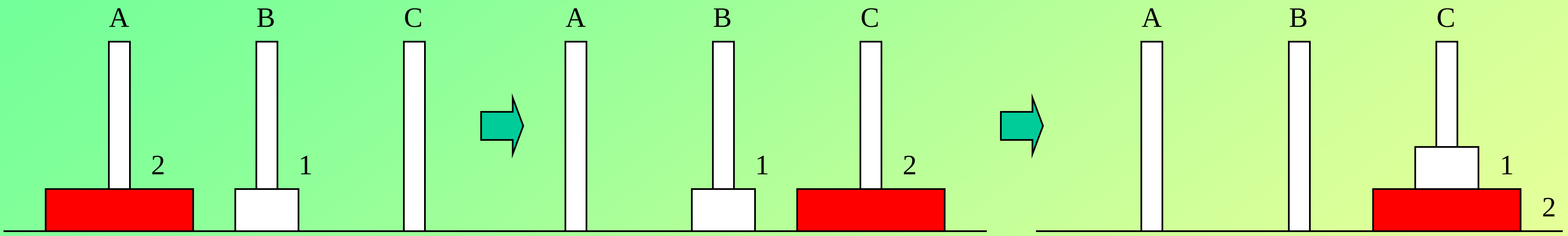
有A、B、C三個塔柱，A塔柱上插有 n 個大小各不相同，由小到大編號為1, 2, 3,…, n 的圓盤，亦即編號越大直徑越大。
這個遊戲的目的，是要把這 n 個盤子，由A塔柱搬至塔C柱，並且必須遵守下列規則：

1. 一次只能搬動一個盤子
2. 任何時間直徑較大的盤子都不能壓在較小盤子的上面
3. 盤子能放在任何塔柱上。

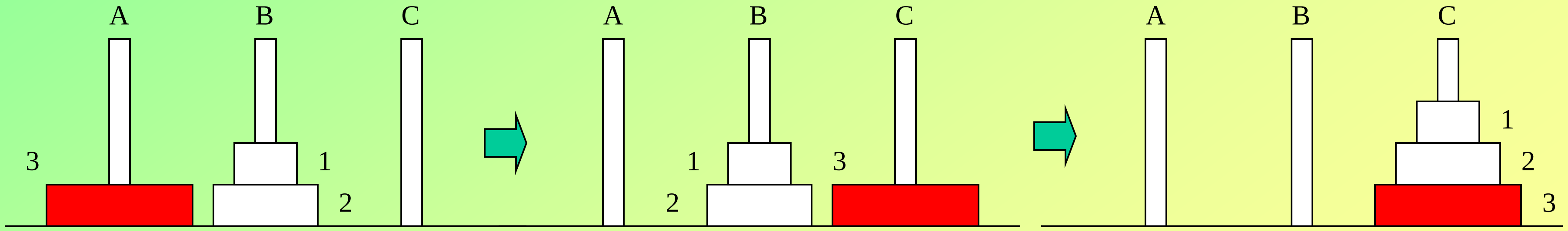


※當 n=1 時，只需一次搬移：祇要將編號 1 的盤子，從塔柱A直接搬至塔柱C即可

※當 n=2 時，必須執行 3 次搬移：

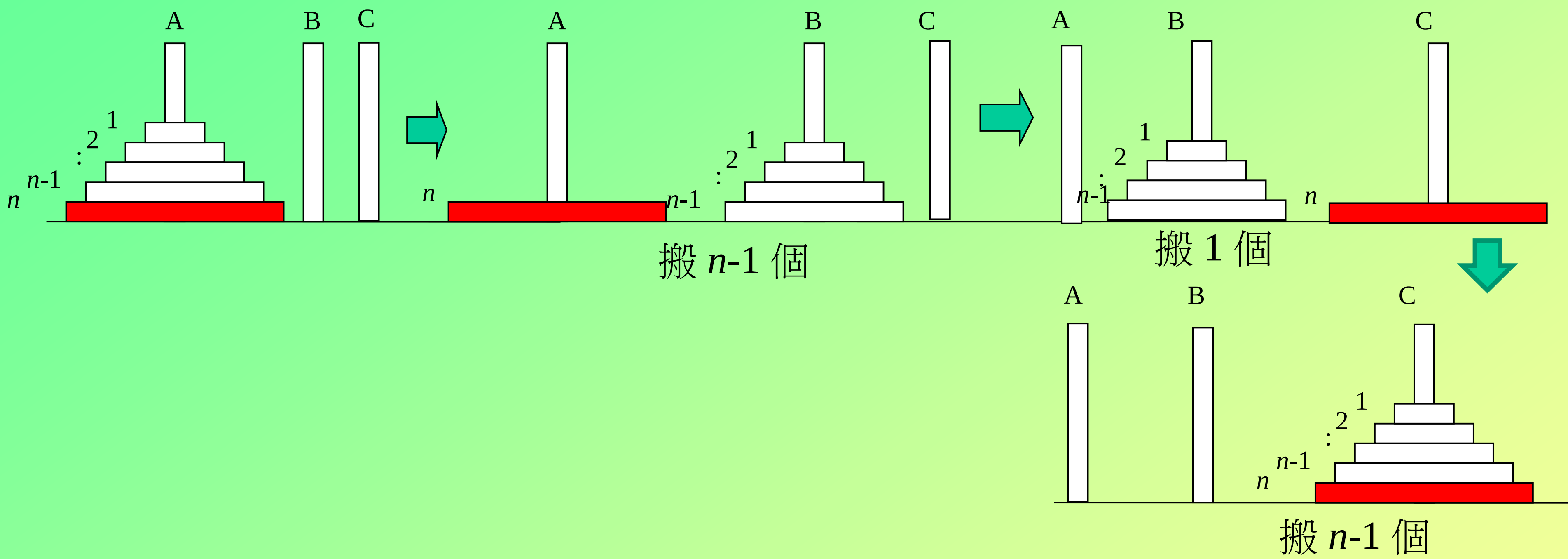


※當 n=3 時，必須執行 7 次搬移：



前 3 次 第 4 次 後 3 次

※當 n 個盤子需要搬移時



$$
\begin{aligned}
A_n &= A_{n-1} + 1 + A_{n-1},\ (A_1 = 1) \\
&= 2 \times A_{n-1} + 1 \\
&= 2 \times (2 \times A_{n-2} + 1) + 1 \quad (A_{n-1} = 2 \times A_{n-2} + 1) \\
&= 2^2 \times A_{n-2} + 2 + 1 \\
&\ \vdots \\
&= 2^k \times A_{n-k} + 2^{k-1} + 2^{k-2} + \ldots + 2 + 1
\end{aligned}
$$

遞迴關係式

當 $k = n\text{-}1$ 時

$$
\begin{aligned}
A_n &= 2^{n-1} \times A_1 + 2^{n-2} + 2^{n-3} + \ldots + 2 + 1 \\
&= 2^{n-1} + 2^{n-2} + 2^{n-3} + \ldots + 2 + 1 \\
&= 2^n - 1
\end{aligned}
$$

二項式係數

◎組合 $$C(n,k) = \frac{n!}{k!(n-k)!}$$

因為階乘的增長非常快速，n值只要超過13，$n!$ 就超出「長整數」(long int) 所能表示的範圍。直接用定義來計算C(n,k) 並不可行，因此才有以遞迴關係式求解的途徑產生（但是遞迴的方法並不是最快的）。

◎遞迴定義

$$C(n,k) = \begin{cases} 1, & \text{if } k=0 \text{ or } k=n \\ C(n-1, k-1) + C(n-1,k), & \text{if } n>k>0 \end{cases}$$

◎遞迴函式

```
int     C(int n, int k)
        {
                if ( k == 0 || k == n ) return 1;     //終止條件
                return  ( C(n-1,k-1) + C(n-1,k) );//遞迴呼叫
        }
```