CH3、Stack & Queue

堆疊與佇列

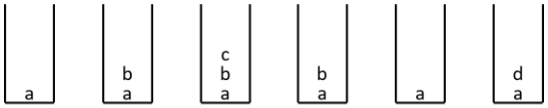
目錄：

Stack 堆疊

Def：

1. 為一有序串列(Order List)
2. 具有 LIFO(Last-In-First-Out)或 FILO(First-In-Last-Out)
3. 有插入(Push)，刪除(Pop)等行為，"皆發生於頂端(Top)"

例：push(a),push(b), push(c), pop,pop,push(d)，問結果？



Output：c, b
Stack 內容：a, d

應用：

1. Subroutine, Recursive 呼叫	7. Graph 的 DFS(追蹤)
2. Infix 轉 Postfix, Prefix	8. 日常取餐盤之行為
3. 後序式計算	9. 將資料反序：*若 Input 大到小，則 Output 小到大*
4. Compiler 文法剖析	10. Stack Computer 機器
5. 迴文檢查(演算法)	11. 處理 Pure-procdure(純程序)
6. Binary Tree 追蹤	12. Maze Problem(迷宮)

Stack 之 ADT

ADT is a spec：

1. of a set of data
2. the set of operation

Object：A finite ordered list with zero or more elements(一有序列帶有 0~多個元素)

Functions：

for all stack ∈ stack
item ∈ element
max_stack_size ∈ positive integer

Include：

1. create
2. isFull
3. add(push)
4. isEmpty
5. delete(pop)

1. create
   `stack creat(mss);`
   => 指建立一空的 Stack，大小為 mss，並將之回傳
2. isFull
   `Boolean isFull(stack, mss)`
   => 接收一 Stack，跟 mss 相比
   若大小相同：Stack full => return true
   若大小不同：return false
3. push
   `stack push(stack, item)`
   => 將 iteme 加入到 stack 之中
   Note: 需 check Stack 是否滿了 ?
   是 => output Stack
   否 => 加入，並回傳 Stack
4. isEmpty
   `Boolean isEmpty(stack)`
   => check Stack 是否為空
   是 => return true
   否 => return false
5. delete(pop)
   `Element delete(stack)`
   => 從 Stack 刪除 Top 端的資料，並回傳
   Not: 欲 check Stack 是否 Empty ? 為空則不能 Delete

另一版本 1~4 相同，But 5 不同

5. delete
   `stack delete(stack)`
   => 刪除 Top 端資料後，回傳 Stack
6. top
   `Element top(stack)`
   => 只查，不 Delete,回傳 Stack 的 Top 元素值

例：

1. pop(push(s, item)) ?
2. top(push(s, item)) ?
3. isEmpty(create(s)) ?
4. pop(create(s)) ?

1. *s*
2. *item*
3. *true*
4. *Stack Empty(Error)*

Stack ADT 之製作

方式：一、Array；二、*Linked list(下章談)*

```
int stack[];
int top=-1,n;        //n 用來記錄大小，stack size
```

程式：

1. create：

```
void create()
{
        top = -1;
        n=size;
        stack = new int[n];
}
```

2. isFull：

```
bool isFull()
{
        retutn top == n-1;
}
```

3. isEmpty：

```
bool isEmpty()
{
        return top==-1;
}
```

4. push：

```
bool push (int item)
{
        if(isFull())
                return false;
        else
        {
                top++;
                stack[top]=item;
                return true;
        }
}
```

5. pop：

```
bool pop(int &item)     //call by reference，參考變數
{
        if(isEmpty())
                return false;
        else
        {
                item=stack[top];
                top--;
                return true;
        }
}
```

Note: int &count=x;   //為 x 取一個暱稱 count，操作 count 及是操作 x(共享記憶體)

Stack Permutation(排列組合)

Def: 給予 n 個 Data,依序執行 push,於過程可產生的合法 Output 組合共 $\frac{1}{n+1} \times C_n^{2n}$ 種*(3 有 5 種；4 有 14 種；5 有 42 種)*

例 1：給 3 筆 Data a, b, c 列出所有合法之 Stack Permutation

abc：push(a), pop[a], push(b), pop[b], push(c), pop[c]
acb：push(a), pop[a], push(b), push(c), pop[c], pop[b]
bac：push(a), push(b), pop[b], pop[a], push(c), pop[c]
bca：push(a), push(b), pop[b], push(c), pop[c], pop[a]
cab：push(a), push(b), push(c), pop[c], => ERROR
cba：push(a), push(b), push(c), pop[c], pop[b], pop[a]

例：(a, b, c, d)之 Stack Permutation，下列哪些不合法？

1. acbd
2. dcab
3. cdab
4. dbca
5. cbad

只有 1, 5 正確

1. *push(a), pop[a], push(b), push(c), pop[c], pop[b], push(d), pop[d]*
2. *push(a), push(b), push(c), push(d), pop[d], pop[c] => ERROR*
3. *push(a), push(b), push(c), pop[c], push(d), pop[d] => ERROR*
4. *push(a), push(b), push(c), push(d), pop[d] => ERROR*
5. *push(a), push(b), push(c), pop[c], pop[b], pop[a], push(d), pop[d]*

n 個 Data 之 Stack Permutation 和下列問題相同:

1. n 個 Node 可形成之 Binary Tree 種類數(CH5)
2. n 個"("和")"之合法配對數

例：3 個"("及")"

1. *((()))*
2. *()()()*
3. *(())()*
4. *()(())*
5. *(()())*

3. (n+1)個矩陣相乘之可能，乘法配對組合數:

例：4 個矩陣(M1-M4)

1. ((M1*M2)*M3)*M4
2. ((M1*(M2*M3))*M4
3. M1*((M2*M3)*M4)
4. (M1*M2)*(M3*M4)
5. M1*(M2*(M3*M4))

[演算法]：矩陣鏈：$\frac{1}{n+1} \times C_n^{2n}$

4. Y 字火車倒車題型(先進後出)

Infix(中序), Postfix(後序), Prefix(前序式) 介紹

一、Infix:

Def: 一般使用的運算或表示格式

格式	$operand_1$	operator	$operand_2$
例：	3	+	5

優點：User 易懂

缺點：Compiler 對 Infix 計算不易

因為需考慮運算子之 Priority 及 Associative，故需多次 Scan 方可求解

⇨ Performance 下降

例：3+5*8

二、Postfix(後序式 or 後置式)

格式	$operand_1$	$operand_2$	operator
例：	3	5	+

優點：Compiler 處理方便，只需由左到右 Scan 一次，即可求出結果

⇨ Performance 較好

只需一個 Stack

三、Prefix(前序式)

格式	operator	$operand_1$	$Operand_2$
例：	+	3	5

優點：同 Postfix，需 Scan 一次即可求解

Note：右到左 Scan

需 2 個 Stack 來 Support

小結：

前序	+	3	5
中序	3	+	5
後序	3	5	+

中序轉前序 or 後序，方法有 2：

(1) 括號法

(2) Stack 演算法

一、括號法

[中轉後]

1. 將運算式依優先權(Priority)及結合性(Associative)加上完整的括
2. 由右到左將運算子取代最近的右括號
3. Output，省略左括號

[中轉前]

1. 將運算式依優先權(Priority)及結合性(Associative)加上完整的括
2. 由左到右將運算子取代最近的右括號
3. Output，省略右括號

Operator	Priority	Associative
1. (), []	高	右結合
2. –(負號)		右結合
3. **, ↑, $, ^(冪次方)		左結合
4. *, /		左結合
5. +, -		左結合
6. >, <, ==, ≠		左結合
7. !(not), ~, ¬		右結合
8. and, or		左結合
9. =(assign)	低	右結合

Note:

1. Operator 有分為：
   (1) 單元(Unary)：-7, !A
   (2) 雙元(Binary)：3+5, 6*8
2. 做運算先後：
   先看 Priority，(如果 Priority 相同，)再看結合性(Associative)
3. 算術 > 關係 > 邏輯

例 1：A+B*C-D/E

*A+(B*C)-(D/E)*
*((A+(B*C)-(D/E))*
後：ABC+DE/-*
*前：-+A*BC/DE*

例 2：A ↑ B ↑ C

(A ↑ (B ↑ C))
後：ABC ↑ ↑
前： ↑ A ↑ BC

例 3：(A+(B-C)/D)-E/(F*G)

*((A+((B-C)/D))-(E/(F*G)))*
後：ABC-D/+ EFG/-*
*前：-+A/-BCD/E*FG*

[Redefitne 題型]

例 4：Priority 規則：() > + - > * /，且：+- => 右結合；*/ => 左結合，則：A+B-C*D/(F*G+H) 為何？

*(((A+(B-C))*D)/(F*(G+H)))*
*後：ABC-+D*FGH+*/*
前：/+A-BCD/*F+GH*

[反向題]
例 5：Postfix：ABC*E2-/F*+，求 Infix=?

*A+((B*C)/(E-2)*F)*
*(A+(((B*C)/(E-2))*F))*
前：+A/*BC-E2F*

例 6：前：/+AB*-CDE，求 Infix=?

*(A+B)/((C-D)*E)*
*((A+B)/((C-D)*E))*
後：AB+CD-E/*

二、Stack 演算法
中序→後序
概念：
1. 若是 Operand 運算元 => 列印
   若是 Operator 運算子，goto 2.
2. ")" => pop，直到遇到"("為止
3. Otherwise：比較 Priority
   (1) ">" Top 端 => push
   (2) "<="Top 端 => pop，直到 Priority > top 端，才 push

Note:
1. Stack 空，Priority 最低
2. "("在 Stack 外 Priority 最高
   ")"在 Stack 內 Priority 最低
3. 左結合運算子，在 Stack 之外的 Priority，小於 Stack 內的
4. 右結合剛好相反

中序→前序
概念：
1. 若是 Operand 運算元 => push 到 Stack T
   若是 Operator 運算子，goto 2.
2. ")" => pop，直到遇到"("為止
3. Otherwise：比較 Priority
   (1) ">" Top 端 => push
   (2) "<="Top 端 => pop，直到 Priority > top 端，才 push

Note:
1. Stack 空，Priority 最低
2. "("在 Stack 外 Priority 最高
   ")"在 Stack 內 Priority 最低
3. 左結合運算子，在 Stack 之外的 Priority，大於 Stack 內的
4. 右結合剛好相反

Infix 轉 Postfix 之演算法程式：

```
while(infix 尚未由左而右 scan 完) do
    begin
        x=NextToken(infix);
        if(x is operand) then print(x);                          //1.
        else if (x is ")") then                                  //2.
            repeat
                y=pop(s);
                if (y!="()) print(x);
            until(y=="(")
        else                                                     //3.
            switch(compare(x, s.Top))
            {
                case">":    push(s,x);                           //3.1
                case"<=":   repeat                               //3.2
                                y=pop(s);
                                print(y);
                            until(x>s.Top);
                            push(s,x);
            }
            while(Not isEmpty(s)) do
            begin
                y=pop(s);
                print(y);
            end
    end
```

Infix 轉 Prefix 之演算法程式：*(會多一個 Stack T)*

```
while(infix 尚未由右而左 scan 完)do
    begin
        x=NextToken(infix);
        if(x is operand) then push(T,x);                         //1.
        else if (x is ")") then                                  //2.
            repeat
                y=pop(s);
                if (y!=")") push(T,y);
            until(y=="(")
        else                                                     //3.
            switch(compare(x, s.Top))
            {
                case">":    push (s,x);                          //3.1
                case"<=":   repeat                               //3.2
                                y=pop(s);
                                push(T,y);
                            until(x>s.Top);
                            push(s,x);
            }
            while(Not isEmpty(s)) do
            begin
                y=pop(s);
                push(T,y);
            end
            while(Not isEmpty(T)) do
            begin
                y=pop(T);
                print(y);
            end
    end
```

例：將中序：A/(B-C*D)轉為後序(含過程)

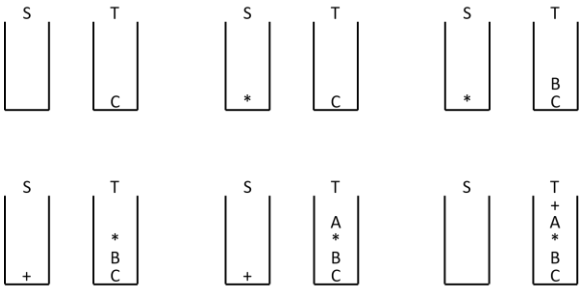
1. *"A" => print"A"*
2. *"/" => / > stack empty => (s, /)*
3. *"(" => (> stack top "/"*
4. *"B" => print"B"*
5. *"-" => - > stack top "(" => (s, ()*
6. *"C" => print"C"*
7. *"*" => * > stack top "-" => (s, -)*
8. *"D" => print"D"*
9. *")" pop[*], pop[-], pop[(], stop*
10. *Scan 完畢：pop all element in stack print "/"*

=> ABCD-/*

例：將中序：A+B*C 轉為前序(含過程)

1. *"C" => push(T, C)*
2. *"*" => * > Stack Empty => push(S, *)*
3. *"B" => push(T, B)*
4. *"+" => + < Stack Top * => pop, push(T, *), + > Stack Empty => push(S, +)*
5. *"A" => push(T, A)*
6. *Scan 完，將 Stack S 剩於 pop 到 T*
7. *Pop all elemenet in T and print*

*=> +A*BC*



Evaluation

Postfix Evaluation 演算法

```
while(postfix 尚未由左而右 scan 完) do
{
        x=NextToken(postfix);
        if(x is operand) then push(s,x);
        else
        {
                1.pop 適當個數之運算元;
                2.執行得結果 r;
                3.push(s,r);
        }
}
pop(s);                 //即為結果
```

例 1：Postfix => 623*/4*

1. 寫出計算過程
2. 系統至少要給幾格的 Stack？

1. 過程：

(1) "6" => push 6
(2) "2" => push 2 6
(3) "3" => push 3 2 6
(4) "" => pop[3], pop[2] => 2*3 = 6 => push 6 6*
(5) "/" => pop[6], pop[6] => 6/6 = 1 => push 1
(6) "4" => push 4 1
(7) "" => pop[4], pop[1] => 4*1 = 4 => push 4*
(8) Scan 完，pop Stack 即為結果

2. 至少給 3 格，看過程最多需多少格

例 2：中序式：A+B*C-(D/E/F)轉成 Postfix，問此 Postfix 計算時至少需多少 Stack size？

Postfix：ABC+DE/F/-*
=> 最少 3 格

Prefix Evaluation 演算法

```
while(prefix 尚未由右而左 Scan 完) do
{
        x=NextToken(pretfix);
        if(x is operand) then push(s,x);
        else
        {
                1.pop 適當個數之運算元;
                2.執行得結果 r;
                3.push(s,r);
        }
}
pop(s);                 //即為結果
```

例 1：Pretfix => *+23/63

1. 寫出計算過程
2. 系統至少要給幾格的 Stack？

1. 過程：

(1) "3" => push 3
(2) "6" => push 6 3
(3) "/" => pop[6], pop[3] => 6/3=2 => push 2
(4) "3" => push 3 2
(5) "2" => push 2 3 2
(6) "+" => pop[2], pop[3] => 2+3=5 => push 5 2
(7) "" => pop[5], pop[2] => 5*2 =10 => push 10*
(8) Scan 完，pop Stack 即為結果

2. 至少給 3 格，看過程最多需多少格

Stack 用於 Compiler(剖析)

[常見題型]

1. 判斷敘述是否合手$\{a^n b^n | n \geq 1\}$：

   n 代表 1~多個；a^n 指 a 出現 n 次、b^n 指 b 出現 n 次

   例：

   正確：ab、aabb、aaabbb

   錯誤：abab、aabbb、aaabb

   步驟：

   (1) 左→右 Scan：”a”則 gush(S, a)；”b”則 pop(S)

   if 之後 Scan 到”a”，Error

   if 無法 pop(S)，Error

   (2) Scan 完畢檢查：Stack 為空？若空則正確；若非空則 Error

2. 迴文測試(Palindrome)

   Check 字串是否符合：{w$w’|w’為 w 的反序}

   例：

   正確：abc$cba、ab$ba

   錯誤：abc$cb

   程式：

```
bool isPalindrome(char *s)
{
        int i=n/2;
        for(i=0;i<n;i++)
        {
                if(s[i]!=s[n-i-1])
                        return false;
        }
        return true;
}
```

3. 判別字串”a”, “b”出現次數是否相同？

   例：

   正確：abba、ababab

   錯誤：aabbb

   步驟：

   (1) 左→右 Scan：”a”則 c_1++；”b”則 c_2++

   (2) Scan 完畢檢查：若 $c_1 == c_2$，則正確；否則錯誤

Queue(佇列)

Def：為一有序串列，具有下列特性：

1. 具 FIFO(Fisrt-In-First-Out)
2. 插入元素在尾端(Rear)，刪除元素在前端(Front)：插入、刪除發生於不同端(Stack 之插入、刪除於同一端)

Queue 應用

1. OS 之 Scheduling Queue
2. IO Device 會用 Queue 接收 IO Request
3. 用於 Simulation(模擬)系統之效能評估
4. IO 時的 Buffer 常用 Queue
5. 圖形的追蹤：BFS(Queue)、DFS(Stack)
6. Binary Tree 的"Level Order Traversal"
7. Priority Queue(優先權：以 Priority 高的先 Output，不是 FIFO)
8. 日常排隊的模式(FIFO)

Queue 的 ADT

Object：A finite ordered list withzero or more elemnts.
(為一有限的有序串列，由 0~多個元素組成)

Function：

for all Queue ∈ Queue
item ∈ Element
max_queue_size ∈ Positive Integer

Include:

1. create
   `queue creat(mss);`
   => 指建立一空的 Queue，大小為 mss，並將之回傳
2. isFull
   `Boolean isFull(queue, mss)`
   => 接收一 Queue，跟 mss 相比
   若大小相同：Queue full => return true
   若大小不同：return false
3. add
   `queue add(queue, item)`
   => 將 iteme 加入到 Queue 之中
   Note: 需 check Queue 是否滿了？
   是 => output Queue
   否 => 加入，並回傳 Queue

4. isEmpty

   Boolean isEmpty(queue)

   => check Queue 是否為空

   是 => return true

   否 => return false

5. delete

   Element delete(queue)

   => 從 Queue 刪除 front 端的資料，並回傳

   Not: 欲 check Queue 是否 Empty？為空則不能 Delete

Queue ADT 之實作方式

一、 Array

[法一] Linear Array

[法二] Circular Array(n-1 格)

[法三] Circular Array(n 格)

二、 利用 Linked List：[法一]、[法二](之後談)

一、Array

[法一] Linear Array

```
int front, rear;
int queue[];
```

程式：

1. create：

```
void create(int size)
{
        front=rear=-1;
        n=size;
        queue=new int[n];
}
```

2. isFull：

```
bool isFull()
{
        retutn rear == n-1;
}
```

3. isEmpty：

```
bool isEmpty()
{
        return front==rear;
}
```

4. add：

```
bool add(int item)
{
        if(isFull())
                return false;
        else
        {
                rear=rear+1;
                queue[rear]=item;        //此二行可合併成：queue[++rear]=item;
                return true;
        }
}
```

5. delete：

```
bool delete(int &item)
{
        if(isEmpty())
                return false;//Queue empty
        else
        {
                front=front+1;
                item=queue[front];       //此二行可合併：item=queue[++front];
                return true;
        }
}
```

Note: 當 rear=n-1(但 front≠-1)時，不代表 Queue 真的滿了

_	_	f	*	*	*	r

解法：一一往回挪

程式：

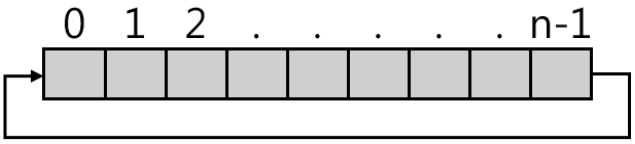
```
if(front!=-1)
{
        for(int i=front+1;i<=rear;i++)          //1.
        {
                queue[i-(front+1)]=queue[ii];
        }
        rear=rear-(front+1);                    //2.
        front=-1;                               //3.
}
```

⇨ O(n)

此法問題：當滿了時，需檢查是否真的滿了，若否則需調整 => O(n)，故以”Circular Queue”來改善

[法二] Circular Queue(n-1 格)

概念：



[資結]

```
int front, rear, n;
int CQ[];
```

程式：

1. create：

```
void create(int size)
{
        front=rear=0;
        n=size;
        CQ=new int[n];
}
```

2. isFull：

```
bool isFull()
{
        retutn rear == n-1;
}
```

3. isEmpty：

```
bool isEmpty()
{
        return front==rear;
}
```

4. add：

```
bool add(int item)
{
        int newrear=(rear+)%n;
        if(newrear==front)
                return false;//Queue Full
        else
        {
                rear=newrear;
                CQ[rear]=item;
                return true;
        }
}
```

5. delete：

```
bool delete(int &item)
{
	if(front==rear)
		return false;
	else
	{
		front=(front+1)%n;
		item=CQ[front];
		return true;
	}
}
```

結論：

1. n 格，只能用 n-1 格
2. 改善 Linear Queue 的 add O(n)→O(1)
3. 欲充分使用 n 格，需額外加上一 tag：若 False 表 Empty、若 True 表 Full

[法三]：因為可多利用一格，但仍需多一個 Tag，故現今較不重要

1. add：

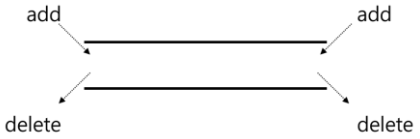
```
bool add(int item)
{
	if(rear==front && tag==true)
		return false;
	else
	{
		rear=(rear+1)%n;
		CQ[rear]=item;
		if(rear==front)	tag=true;
		return true;
	}
}
```

2. delete：

```
bool delete(int &item)
{
	if(front==rear && tag==false)
		return false;
	else
	{
		front=(front+1)%n;
		item=CQ[front];
		if(front==rear)	tag=false;
		return true;
	}
}
```

Queue 的種類

1. 一般 Queue，前端刪除、尾端加入(FIFO)
2. Priority Queue(非 FIFO)
   (1) 插入任意元素
   (2) 刪除最大/最小(擇一)元素：採用 Heap(最大使用 Max-Heap、最小使用 Min-Heap)
3. Double-Ended Queue(雙邊佇列)
   Queu 的兩端(Rear & Front)皆可做 add 與 delete



   分為 2 種：
   (1) Input-Restricted：
      插入：固定端；刪除：任意端
   (2) Output-Restricted：
      插入：任意端；刪除：固定端
4. Double-Ended Priority Queue(雙邊優先佇列)
   提供：
   (1) 插入任意值
   (2) 刪除最大元素
   (3) 刪除最小元素(刪除最大、最小元素之指令為併存)
   採用"Min-Max Heap"或"Deap"

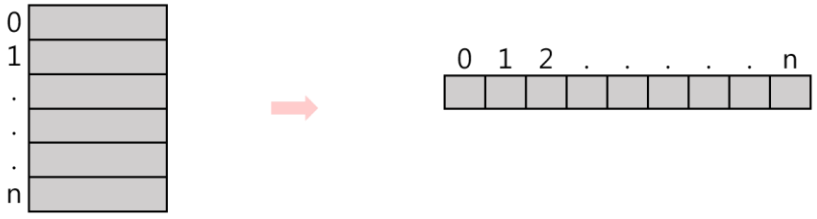
[補充]Stack 的相互製作

一、利用 Stack 作 Queue
二、利用 Queue 作 Stack

一、利用 Stack 作 Queue

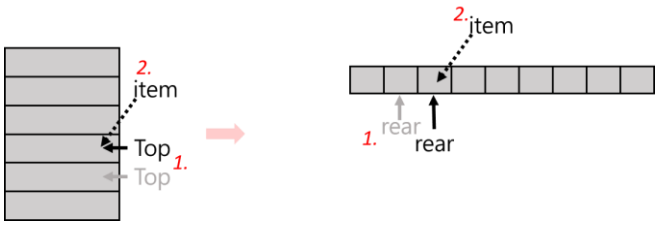
1. create

```
createQ(n)
{
        createS(n);
}
```



2. add

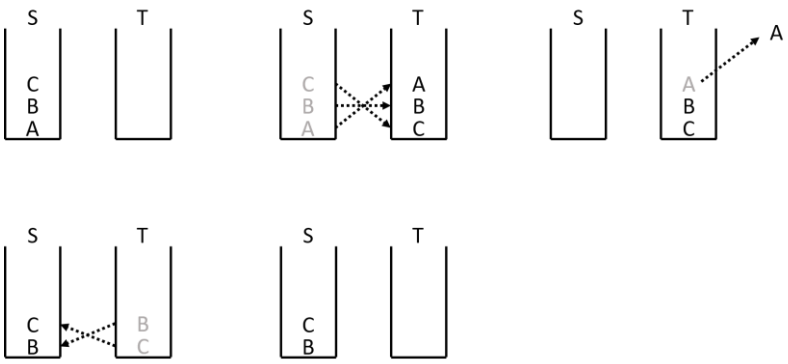
```
addQ(Q, item)
{
        if(isFull(S))
                then Queue Full;
        else push(S, item);
}
```



Note 以 Queue 模擬 Stack

3. delete

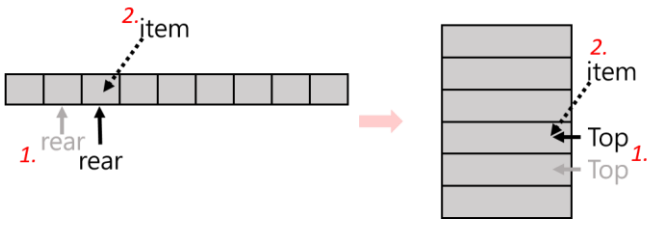
```
deleteQ(Q)
{
        if(isEmpty(S))
                then Queue Empty;
        else
        {
                T: stack;
                while(Not isEmpty(S)) do
                {
                        y=pop(S);
                        push(T, y);
                }
                item=pop(T);
                while(Not isEmpty(S))
                {
                        y=pop(T);
                        push(S, y);
                }
        }
}
```



二、利用 Queue 作 Stack

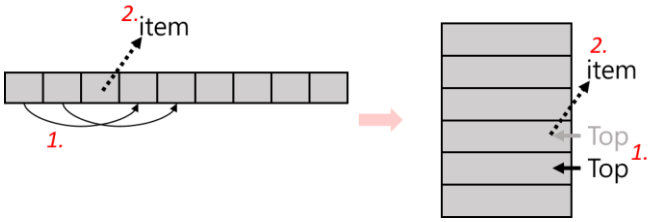
1. push

```
push(S, item)
{
        if(isFull(Q))
                then Stack Full;
        else
                AddQ(Q, item);
}
```



2. pop

```
pop(S)
{
        if(isEmpty(Q))
                then Stack Empty;
        else
        {
                n=Rear-Front;
                for i=1 to (n-1) do        //1.
                {
                        y=deleteQ(Q);
                        addQ(Q, y);
                }
                item=deleteQ(Q);        //2.
        }
}
```



Multiple Stack 製作

作法：利用一個一維陣列實作出：

一、2 個 Stack

二、n 個 Stack(n>2)

Note：Stack1 會有 Top1、Stack2 會有 Top2，只要 Top1≠Top2，則還沒有 Full

一、2 個 Stack

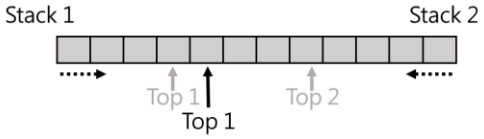
[資結]

```
A: array[1: n];
Top1: int=0;
Top2: int=n+1;
```

程式：

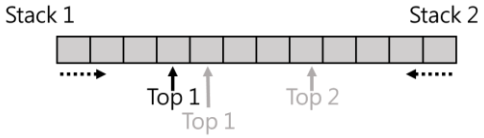
1. push

```
push(i, item)       //將 item 放入 Stacki
{
        if(i==1)    //加入到第 1 個 Stack
        {
                if(Top1+1==Top2)
                        then Stack Full;
                else
                        A[++Top1]=item;
        }
        else   //加入到第 2 個 Stack
        {
                if(Top2-1==Top1)
                        then Stack Full;
                else
                        A[--Top2]=item;
        }
}
```



2. pop

```
pop(i)                      //針對 ith Stack pop
{
        if(i==1)    //pop 第 1 個 Stack
        {
                if(Top1==0)
                        then Stack1 Empty;
                else
                        item=A[Top1--];
        }
        else        //pop 第 2 個 Stack
        {
                if(Top2==n-1)
                        then Stack2 Empty;
                else
                        item=A[Top2++];
        }
}
```



二、一個 Array 製作 n 個 Stack

說明：令 array[1: m]

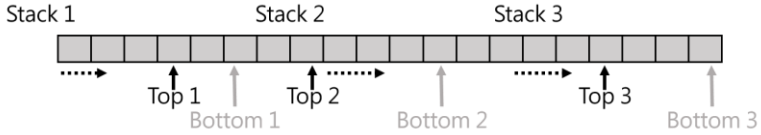
1. 起初每個 Stack 分到相同空間，約[m/n]
2. 每個 Stack，除了 Top 需再加一個 Bottom 指標

製作：

```
Top[1: n] of integer
Bottom[1: n+1] of integer
```

Note：各 Stack i 之 Top&Bottom 初始值：

```
Top[i]=Bottom[i]=(i-1)*[m/n];
Bottom[n+1]=m;
```



判別：

```
Top[i]=Bottom[i];        //Stack Empty
Top[i]=Bottom[i+1];      //Stack Full
```

程式：

1. push

```
push(i, item)
{
        if(Top[i]==Bottom[i+1])
                then Stack i Full;
        else
        {
                Top[i]=Top[i]+1;
                A[Top[i]]=item;
        }
}
```

2. pop

```
pop(i)        //item
{
        if(Top[i]==Bottom[i])
                then Stack i Empty;
        else
        {
                item=A[Top[i]];
                Top[i]=Top[i]-1;
        }
}
```

Note：當 Stack i 滿了，不見得代表整個 Array 皆滿

⇨ *Solution：挪動元素所在位置：O(n)(效能差)*