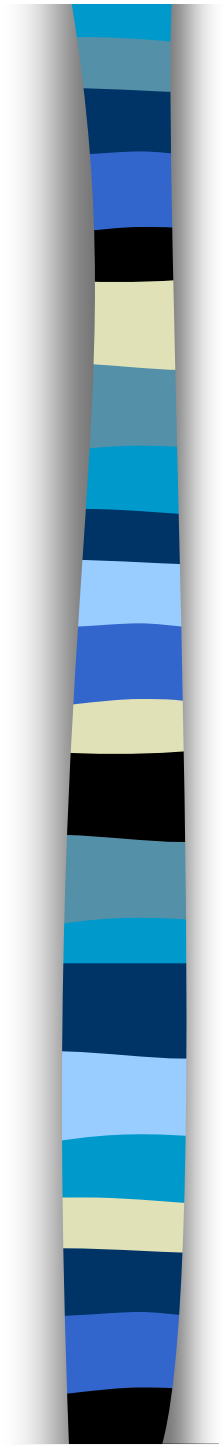


CHAPTER 3

STACKS AND QUEUES

堆疊和佇列

- 堆疊擁有的共同特性就是—先行進入者將較後離去（first in last out, 或簡稱FILO）、或後來進入者可先行離去（last in first out, 或簡稱LIFO）。
- 佇列共同特性在於—先行進入者將率先離去（first in first out, 或簡稱FIFO）。

The Stack ADT

- A stack is an ordered list in which insertions and deletions are made at one end called the top.
- S=(a0,a1,…….an-1),a0 is the bottom element, an-1 is the top element, and a i is on the top of element ai-1,o<i<n.
- Figure3.1 illustrates this sequence of operations.
- Since the last element inserted into a stack is the first element removed, so stack is a Last-in-First-Out(LIFO).

stack: a Last-In-First-Out (LIFO) list

A

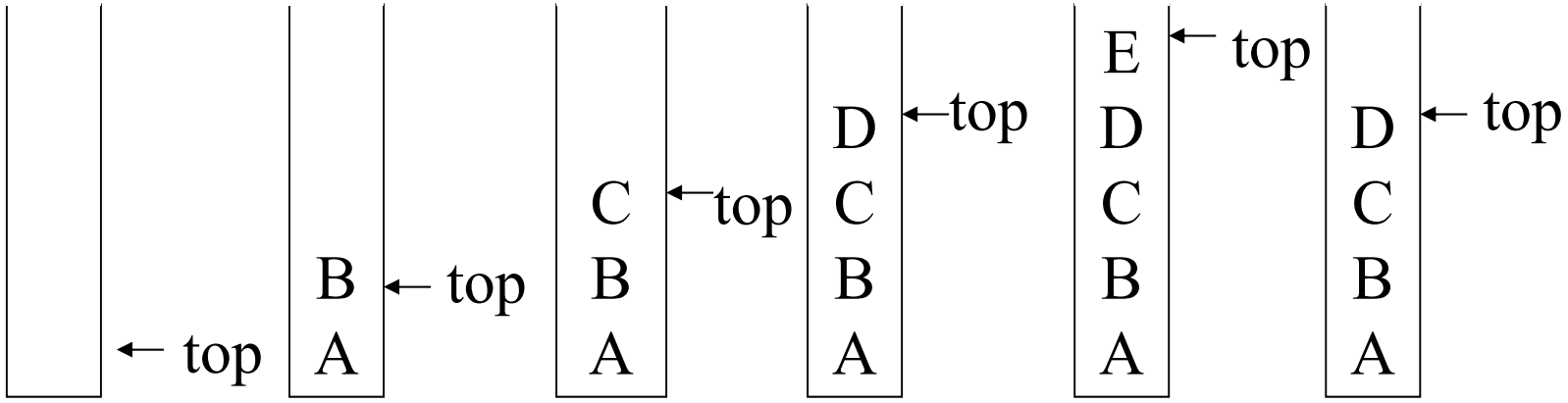
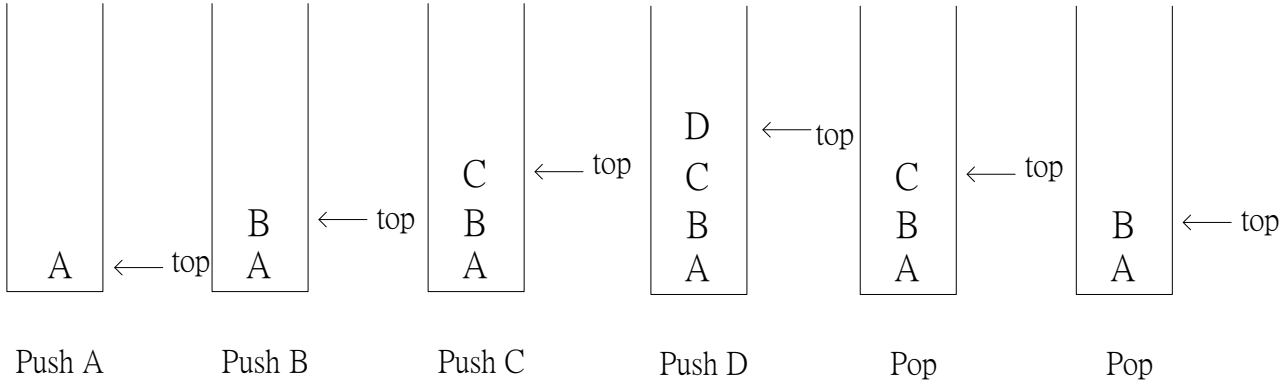
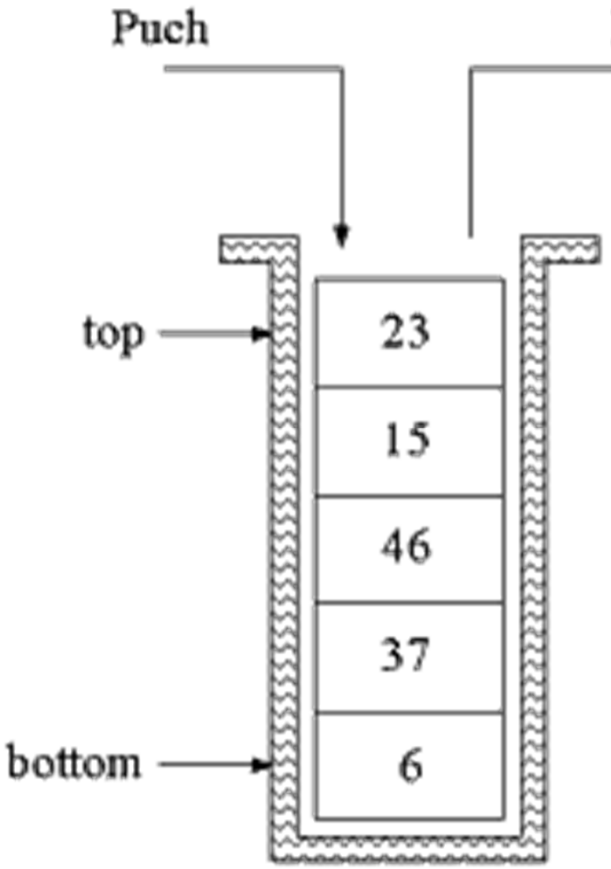


***Figure 3.1:** Inserting and deleting elements in a stack (p.102)



- CREATE(S)：建立空的堆疊。
- PUSH(data, S)：將資料加入堆疊的頂端。
- POP(S)：傳回堆疊頂端的資料，並將該筆資料自堆疊中刪除。
- TOP(S)：傳回堆疊頂端的資料，但不將該筆資料自堆疊中刪除。
- EMPTY(S)：若堆疊內已無任何資料，就傳回真值(True)，否則為假值(False)。
- FULL(S)：若堆疊內已滿，就傳回真值(True)，否則為假值(False)。

構造與相關的操作



Stack operations

1. Create
2. Push(a)
3. Pop(b)
4. IsEmpty
5. IsFull
6. Num of Elements
7. Depth(c)

abstract data type for stack

structure *Stack* is

objects: a finite ordered list with zero or more elements.

functions:

for all *stack* ∈ *Stack*, *item* ∈ *element*, *max_stack_size* ∈ positive integer

Stack CreateS(*max_stack_size*) ::=
create an empty stack whose maximum size is *max_stack_size*

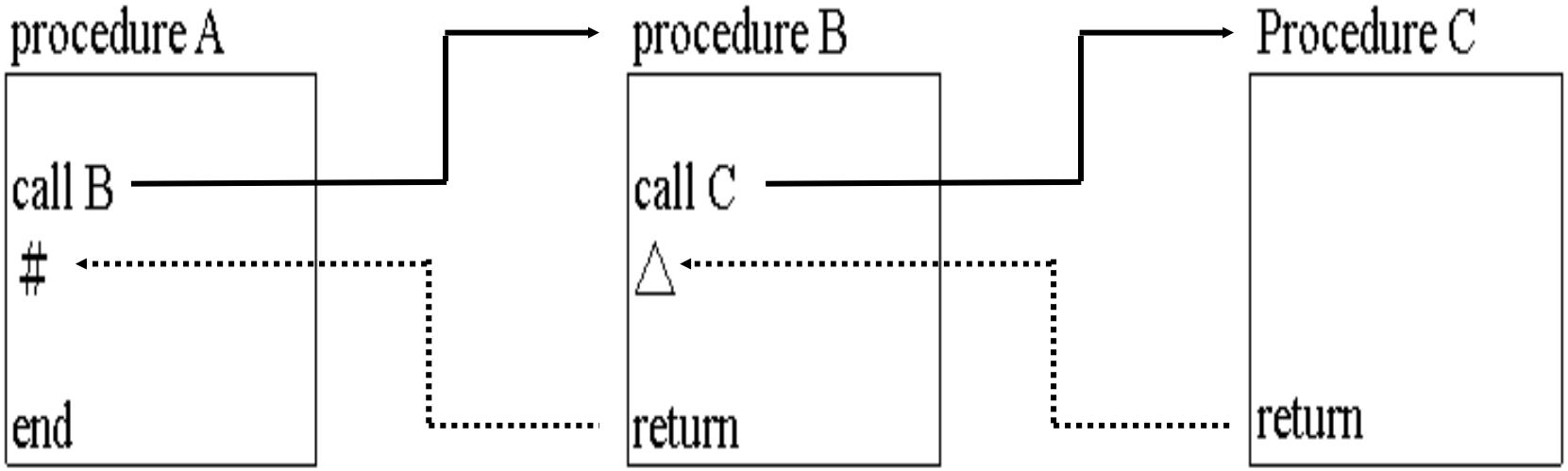
Boolean IsFull(*stack*, *max_stack_size*) ::=
if (number of elements in *stack* == *max_stack_size*)
return TRUE
else return FALSE

Stack Add(*stack*, *item*) ::=
if (IsFull(*stack*)) *stack_full*
else insert *item* into top of *stack* and **return**

Boolean IsEmpty(*stack*) ::=
if(*stack* == CreateS(*max_stack_size*))
return TRUE
else return FALSE
Element Delete(*stack*) ::=
if(IsEmpty(*stack*)) **return**
else remove and return the *item* on the top
of the stack.

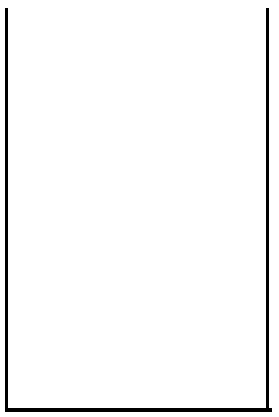
***Structure 3.1:** Abstract data type *Stack* (p.104)

程序呼叫之範例一

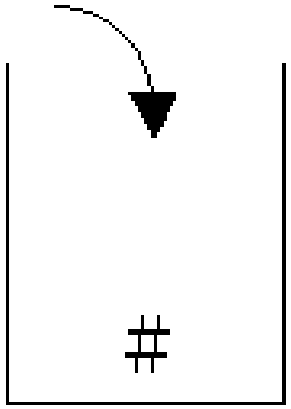


＃和△表示程序呼叫結束後的返回處

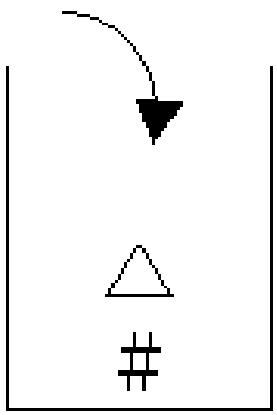
堆疊解決堆疊程序呼叫時流程轉移的過程



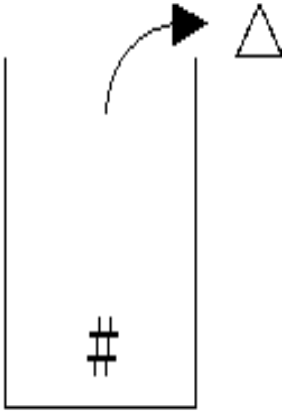
空堆疊



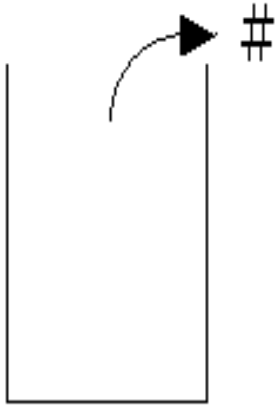
在A程序中呼叫（call)程序B，遂將返回位址#push入堆疊中，流程轉至B。



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。



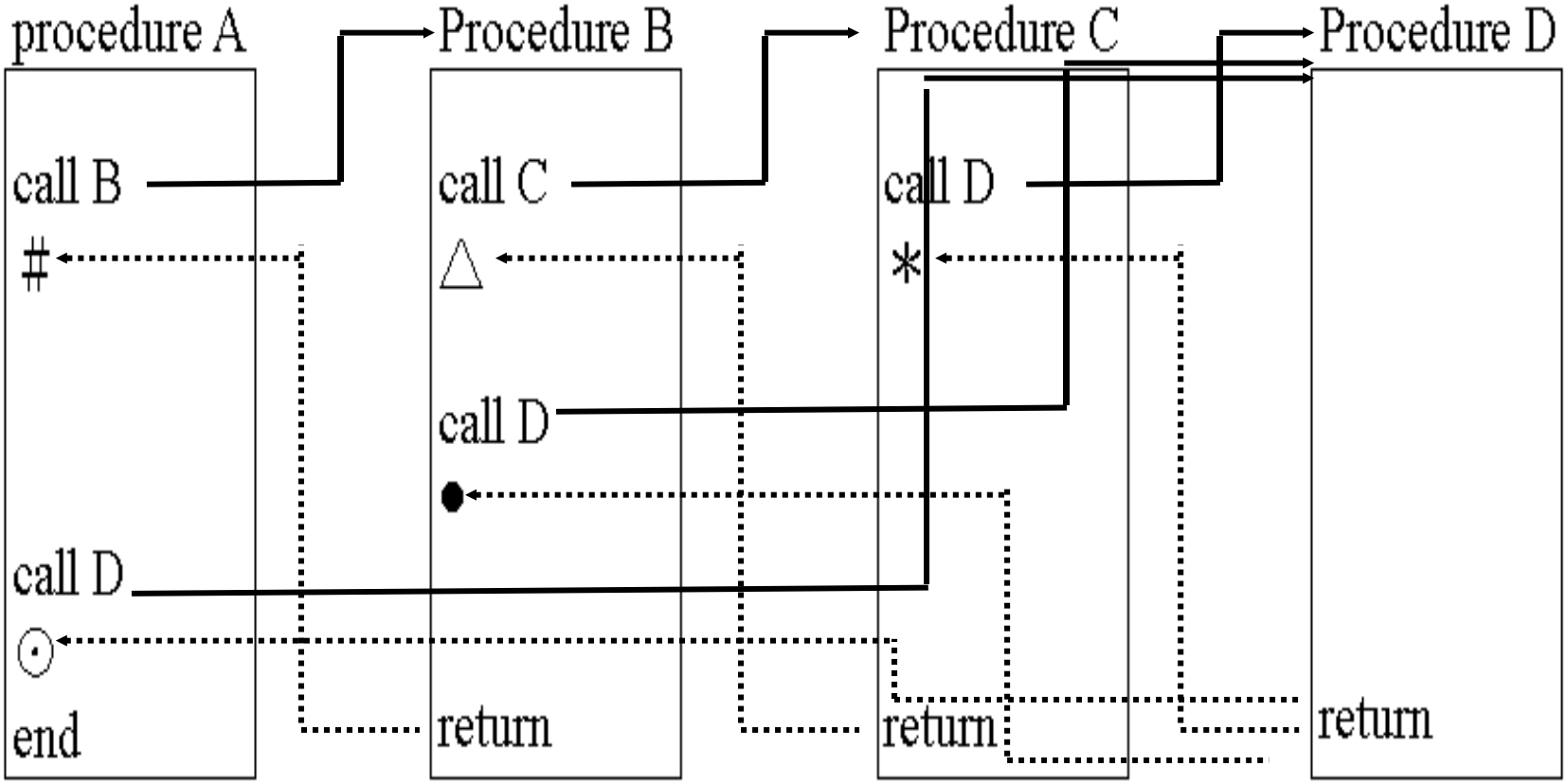
程序C執行結束，pop出堆疊頂端元素△，做為流程轉移的目的地（返回B）。



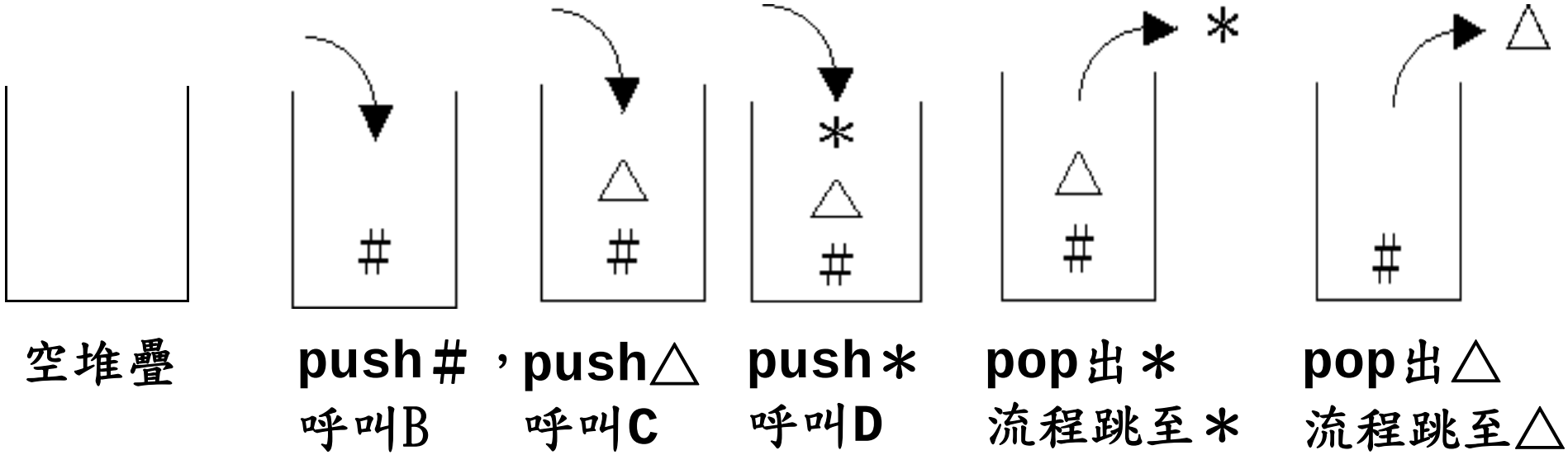
程序B執行結束，pop出堆疊頂端元素#做為流程轉移目的地（返回A）。

堆疊保證了程序呼叫時流程轉移的正確性

程序呼叫之範例二


procedure A
Procedure B
Procedure C
Procedure D
call B
call C
call D
#
△
*
call D
call D
end
return
return
return

堆疊的操作過程


空堆疊
push＃，push△
呼叫B
呼叫C
push＊
呼叫D
pop出＊
流程跳至＊
pop出△
流程跳至△
push ●
呼叫D
pop出●
流程跳至●
pop出＃
流程跳至＃
push ⊙
呼叫D
pop出⊙
流程跳至⊙

an application of stack: stack frame of function call
(activation record)

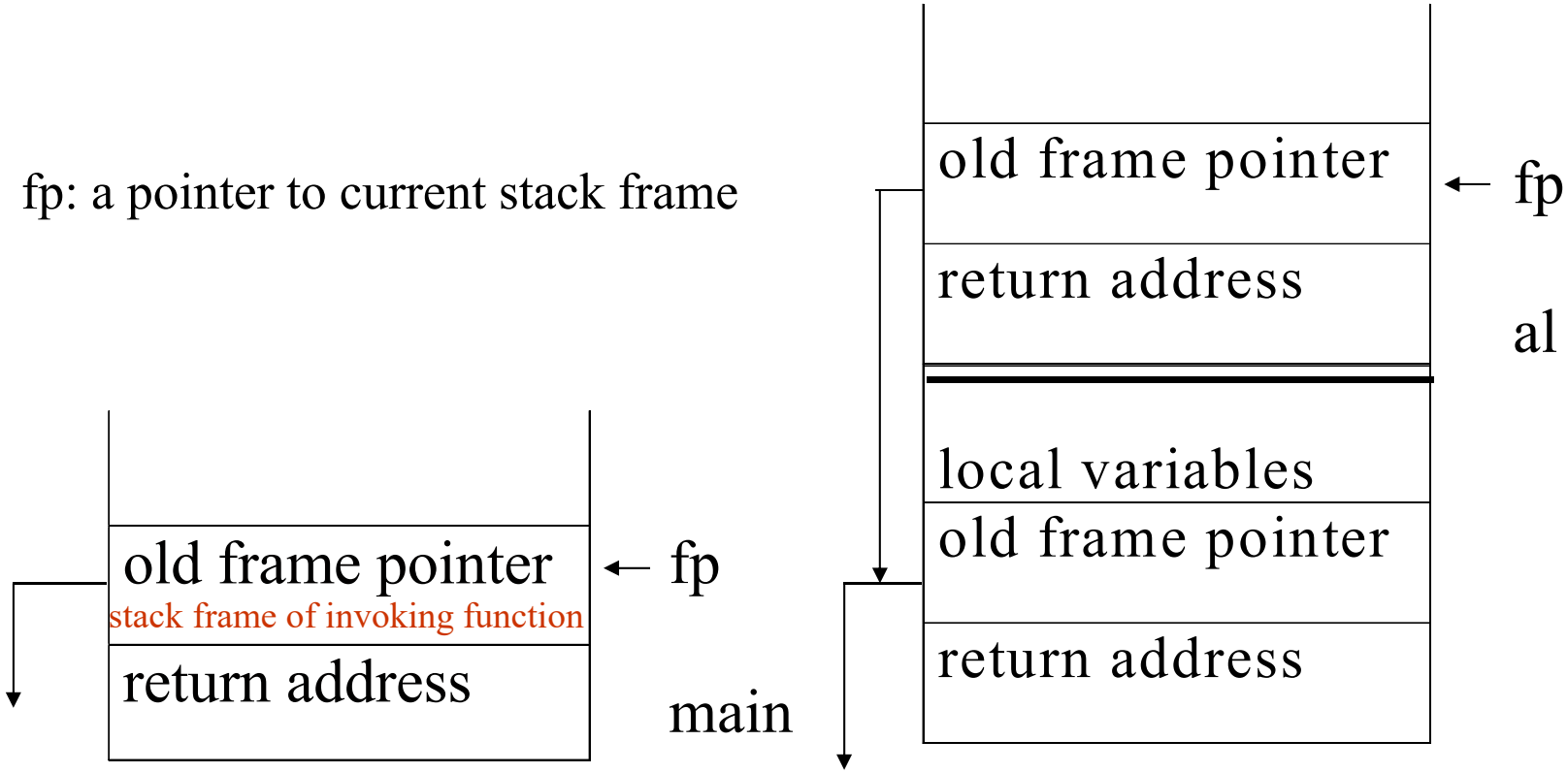


***Figure 3.2:** System stack after function call a1 (p.109)

堆疊的基本運算

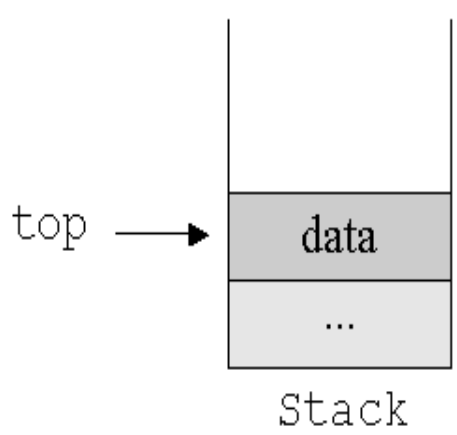
- 程式: 堆疊的宣告

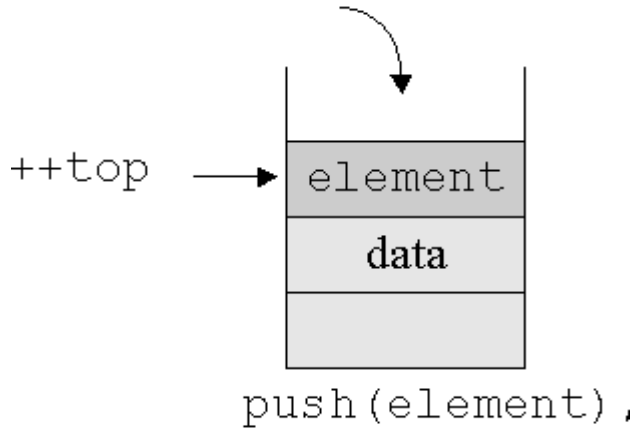
```
1 #define maxsize 5;
2 int stack[maxsize];
3 int top= -1;
```

- 第2行中宣告了大小為5個整數的陣列，將之做為堆疊使用，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。

Push 運算

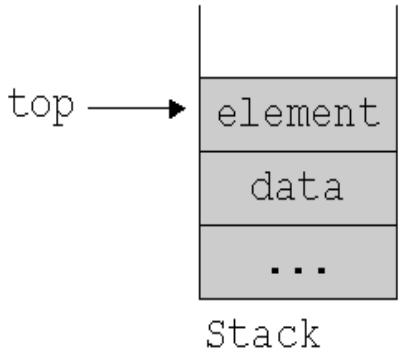
```
1  void push(int element)
2{  if (IsFull( )) StackFull();
3    else Stack[++top]=element;
4}
```

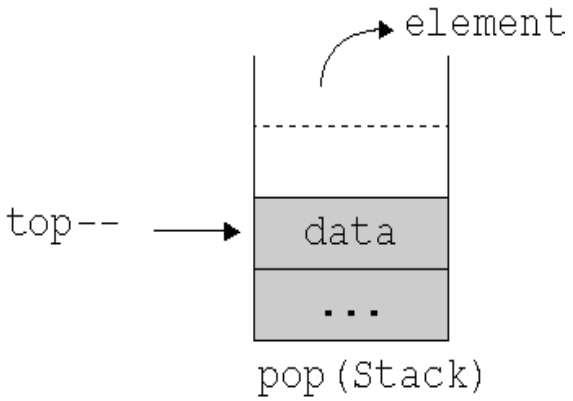




Pop 運算

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





IsFull運算及IsEmpty運算

- 程式IsFull運算

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

陣列Stack中[0]~[maxsize-1]的位置皆放滿元素），來決定是否堆疊已滿？若滿了傳回1，否則傳回0。

- 程式IsEmpty運算

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

Implementation: using array

***Stack* CreateS(max_stack_size)** ::=

```
#define MAX_STACK_SIZE 100 /* maximum stack size */
typedef struct {
        int key;
        /* other fields */
        } element;
element stack[MAX_STACK_SIZE];
int top = -1;
```

***Boolean* IsEmpty(Stack)** ::= top< 0;

***Boolean* IsFull(Stack)** ::= top >= MAX_STACK_SIZE-1;

Add to a stack

```
void add(int *top, element item)
{
 /* add an item to the global stack */
    if (*top >= MAX_STACK_SIZE-1)  {
         stack_full( );
         return;
    }
    stack[++*top] = item;
}
```

***program 3.1:** Add to a stack (p.111)

Delete from a stack

```
element delete(int *top)
{
 /* return the top element from the stack */
    if (*top == -1)
        return stack_empty( );  /* returns and error key */
    return stack[(*top)--];
 }
```

***Program 3.2:** Delete from a stack (p.111)

- C語言程式碼如下：

```
#define  N  100  /* N 為堆疊大小 */
int  stack[N];     /* 陣列stack當作堆疊 */
int  top=  -1;    /* top表頂端之位置 */
```

- 檢查堆疊是否已滿，若已滿則加入失敗。
- 否則將堆疊頂端之指標top 值加1 ，即top上移一格，新資料在加入目前top所指之陣列元素位置中。

- 演算法：虛擬碼

```
Procedure push(int stack[ ] , int element)
{
index top;
if (top ＞＝stack的範圍)
    顯示堆疊已滿的錯誤訊息;
else
{
top ＝ top＋1;
stack[ top ]=element;
 }
}
```

- C語言：程式碼(一)

```
void push (int Stack[ ] , int MaxSize , int top , int
   item )
{
if ( top ＞＝MaxSize－1 )
printf(“堆疊滿了…”) ;
else Stack [+ +top ]＝item ;
}
```

- C語言：程式碼(二)

```
void push(int d)   /*加入資料於堆疊內*/
{
  if(top == N-1)
{
     printf("堆疊滿了\n");     /* 注意堆疊大小 */
       exit(1);                /* 加入失敗，執行結束 */
    }                                    /* end if */
  stack[++top]=d;
}
```

- 檢查堆疊是否空了，若是空堆疊則刪除失敗。
- 否則傳回頂端資料後將top 值減1 ，即top向下移一格。

- 演算法：虛擬碼

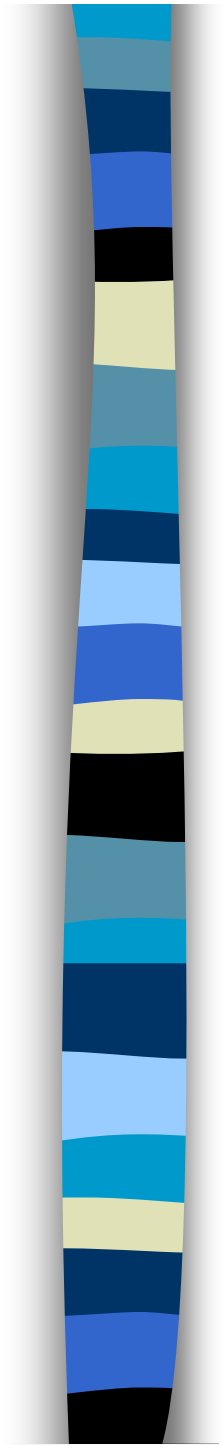
```
Procedure pop(int stack[ ])
{
index top;
if (top 已到堆疊底端)
    顯示錯誤訊息;
else
{
x=堆疊的頂端元素;
top ＝ top－1;
傳回 x;
 }
}
```

- C語言：程式碼(一)

```
void pop (int Stack[ ] , int top )
{
if ( top < 0 )
printf(“堆疊空了，無任何資料可刪…”) ;
else
{
print(“%d 從堆疊被刪除了”  , Stack [ top ] ) ;
x ＝ stack [top - - ];
 }
}
```

- C語言：程式碼(二)

```
int pop()
{
  if(top == -1)                    /* 注意空堆疊情形*/
{ printf("堆疊空了\n");
    exit(1);                       /*刪除失敗，執行結束 */
   }
  return(stack[top--]);
}
```



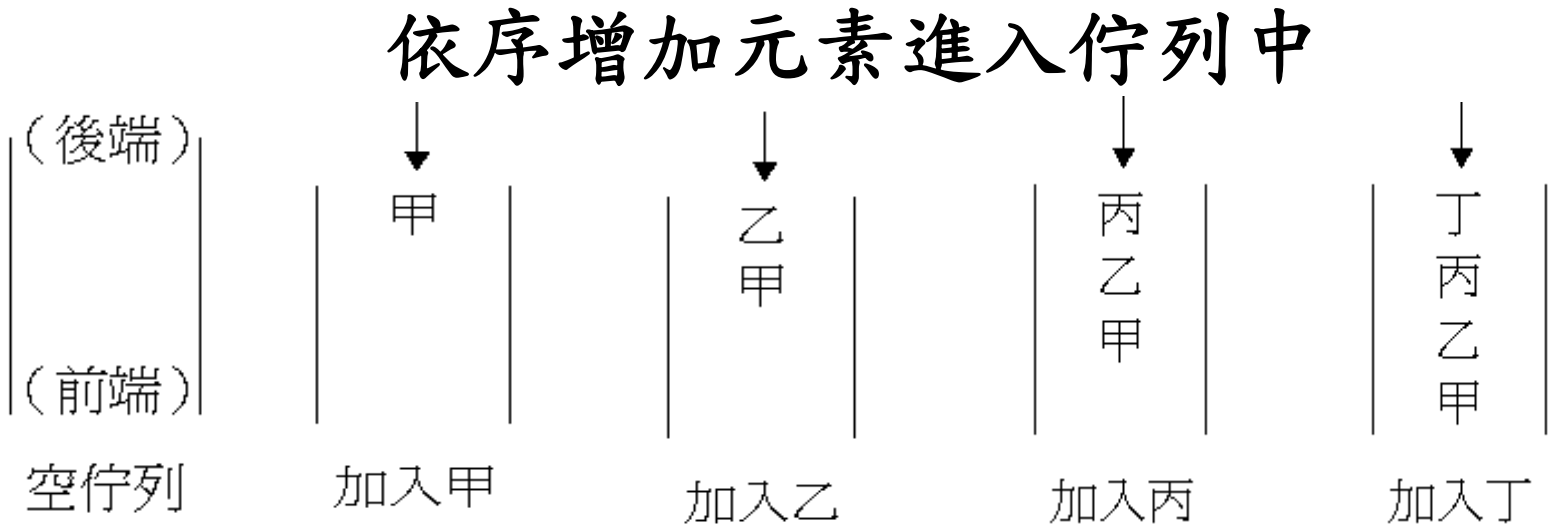
作業

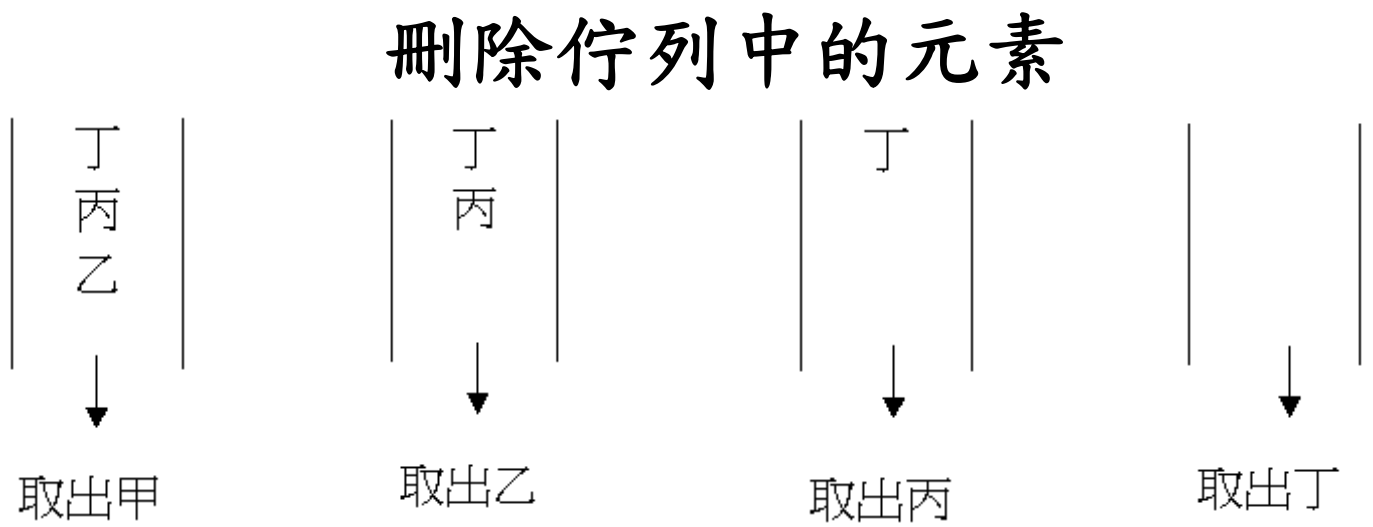
- P112: ex4

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
- 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。

佇列的邏輯圖示及其基本運算





佇列的表示法及基本運作

1. CREATE(Q)：建立空的佇列
2. ADDQ(data, Q)：將資料加入佇列的尾端(rear)
3. DELETEQ(Q)：傳回佇列前端(front)的資料，並將該筆資料自佇列中刪除
4. FRONT(Q)：傳回佇列前端的資料，但不將該筆資料自佇列中刪除
5. EMPTY(Q)：若佇列內已無任何資料，就傳回真值(True)，否則為假值(False)
6. FULL(Q)：若佇列內已滿，就傳回真值(True)，否則為假值(False)

Queue: a First-In-First-Out (FIFO) list

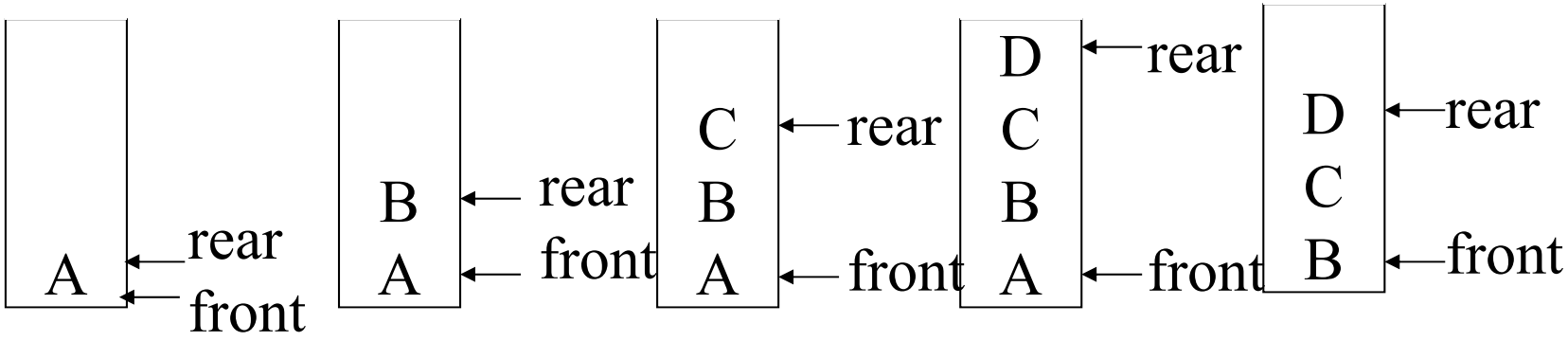


*__Figure 3.4:__ Inserting and deleting elements in a queue (p.114)

Abstract data type of queue

structure *Queue* is

objects: a finite ordered list with zero or more elements.

functions:

for all *queue* ∈ *Queue*, *item* ∈ *element*,

max_ queue_ size ∈ positive integer

Queue CreateQ(*max_queue_size*) ::=

create an empty queue whose maximum size is *max_queue_size*

Boolean IsFullQ(*queue*, *max_queue_size*) ::=

if(number of elements in *queue* == *max_queue_size*)

return *TRUE*

else return *FALSE*

Queue AddQ(*queue*, *item*) ::=

if (IsFullQ(*queue*)) *queue_full*

else insert *item* at rear of *queue* and return *queue*

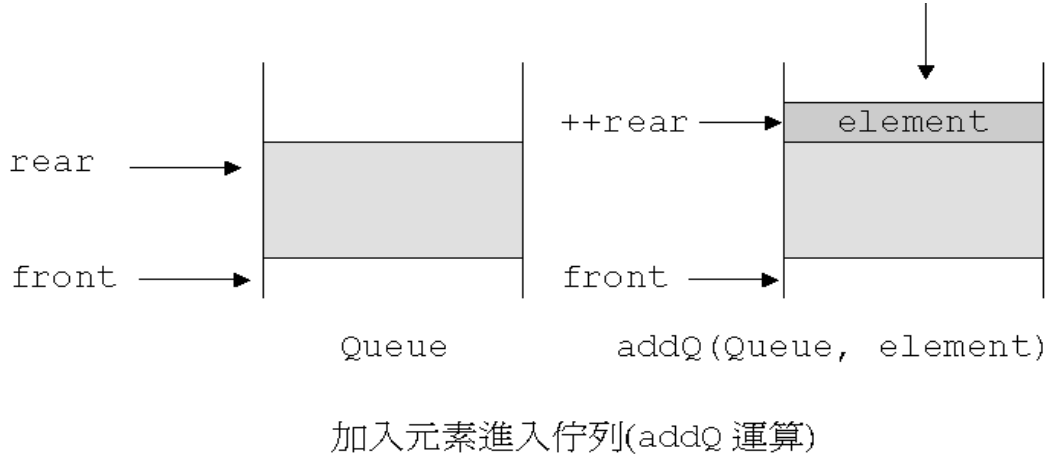
Boolean IsEmptyQ(*queue*) ::=
    if (*queue* ==CreateQ(*max_queue_size*))
    return *TRUE*
    else return *FALSE*
Element DeleteQ(*queue*) ::=
    if (IsEmptyQ(*queue*)) **return**
    else remove and return the *item* at front of queue.

***Structure 3.2:** Abstract data type *Queue (p.115)*

佇列的基本運算

程式: 佇列的宣告

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



加入元素進入佇列(addQ 運算)

程式: addQ運算

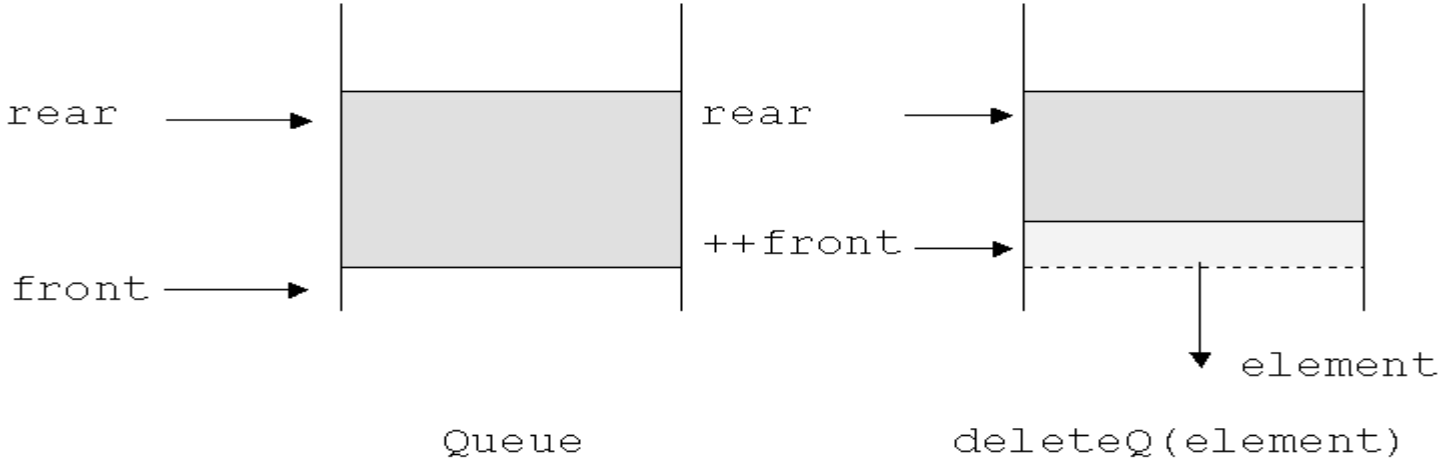
```
1 void addQ(int element)
2 { if (IsQFull()) QueueFull ();
3   else Queue [++rear]= element;
4 }
```

deleteQ 運算

- 程式

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

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

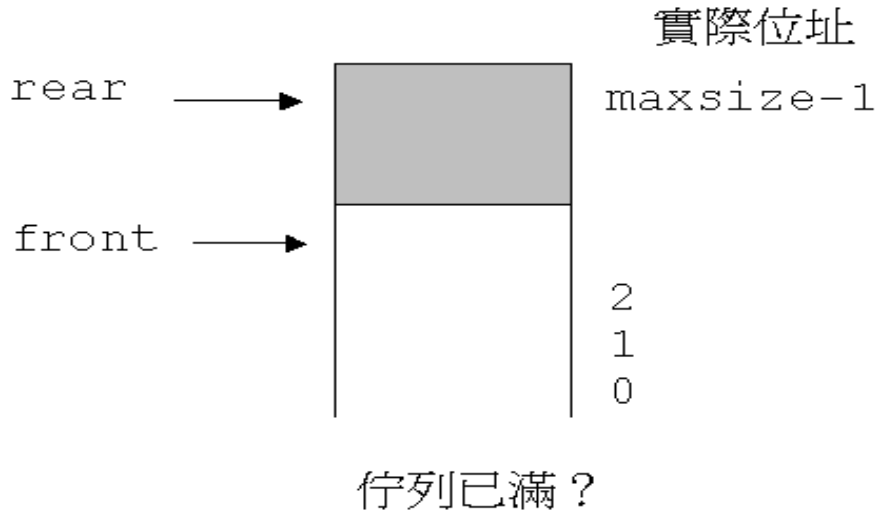


自佇列中刪除元素（deleteQ 運算）

IsQEmpty運算及IsQFull運算

程式： IsQEmpty運算

```
1 int IsQEmpty()
2 {  if (rear == front) return 1;
3    return 0 ;
4 }
```



程式： IsQFull運算

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

Implementation 1: using array

Implementation is using a one-dimensional array and two variables, front and rear.

```
Queue CreateQ(max_queue_size) ::=
# define MAX_QUEUE_SIZE 100/* Maximum queue size */
typedef struct {
            int key;
            /* other fields */
            } element;
element queue[MAX_QUEUE_SIZE];
int rear = -1;
int front = -1;
Boolean IsEmpty(queue) ::= front == rear
Boolean IsFullQ(queue) ::= rear == MAX_QUEUE_SIZE-1
```

Add to a queue

```
void addq(int *rear, element item)
{
/* add an item to the queue */
    if (*rear == MAX_QUEUE_SIZE_1) {
       queue_full( );
       return;
    }
   queue [++*rear] = item;
}
```

***Program 3.3:** Add to a queue (p.108)

Delete from a queue

```
element deleteq(int *front, int rear)
{
/* remove element at the front of the queue */
   if ( *front == rear)
      return queue_empty( );     /* return an error key */
   return queue [++ *front];
}
```

***Program 3.4:** Delete from a queue(p.108)

problem: there may be available space when IsFullQ is true
I.E. movement is required.

Application: Job scheduling

front	rear	Q[0]	Q[1]	Q[2]	Q[3]	Comments
-1	-1					queue is empty
-1	0	J1				Job 1 is added
-1	1	J1	J2			Job 2 is added
-1	2	J1	J2	J3		Job 3 is added
0	2		J2	J3		Job 1 is deleted
1	2			J3		Job 2 is deleted

***Figure 3.5:** Insertion and deletion from a sequential queue (p.117)

佇列之宣告

一般陣列表示佇列

```
#define MaxSize 100          /* 佇列大小 */
int struct queue [MaxSize]；        /* 以陣列表示佇列
*/
int front, rear；
/* front表陣列佇列第一個元素前一格位置，rear表後端之位置
*/
```

以Structure(結構體)表示佇列

```
#define MaxQueue 100        /* 佇列大小 */
typedef struct queue {      /* 以結構體表示佇列*/
int item[MaxQueue] ;        /* 陣列item儲存佇列項目*/
int front, rear; } q ;
/* front表陣列佇列第一個元素前一格位置，rear表後端之位置*/
```

佇列中

一般陣列表示佇列

```
front = 0;          /*front 起始值  */
rear = - 1;         /*rear 起始值  */
```

Structure的佇列

```
q.front = -1;       /* q.front 起始值  */
q.rear = - 1;       /* q.rear 起始值  */
```

加入元素

演算法：虛擬碼

```
Procedure Qadd(int queue[ ] , element )
{
index  rear ;
if (rear ＞佇列的最後位置)
   顯示佇列已滿的錯誤訊息;
else
{
rear ＝ rear＋1;
queue[ rear ] = element
傳回 queele [rear ];
 }
}
```

C語言：程式碼(一)

```
void QAdd (int queue [ ] , int MaxSize , int rear , int x )
{
if ( rear >= MaxSize - 1)
printf(“佇列已滿了…”) ;
else queue [+ + rear ]＝x ;
}
```

C語言：程式碼(二)

```
void queue (int data)
/* 此函數將data放入queue 之後端 */
{
 if (q.rear == MaxQueue-1)
   { printf (“ queue is full \n”);
     exit(1);
    }
  else
    q.item[++q.rear] = data;
//將q.rear +1後，再將新元素data 放在q中
}
```

刪除元素

演算法：虛擬碼

```
Procedure QDel(int queue[ ] )
{
index  front , rear ;
if (front ＞= rear)
    顯示佇列已空的錯誤訊息;
else
{
front ＝ front＋1;
傳回 queue [front ];
 }
}
```

C語言：程式碼(一)

```
void QDel (int queue [ ] , int front , int rear )
{
if ( front >= rear )
printf(“佇列已空了，無資料可刪…”) ;
else
 {
front + +;
printf(“ %d 已從佇列中刪除了 ”  , queue [front ] ;
}
}
```

C語言：程式碼(二)

```
int dequeue (void)
 /* 此函數自queue 之前端刪除元素 */
{
 if (q.front == q.rear)
/* q.front == q.rear 表 queue 為空 */
    { printf (“ queue is empty \n”);
      exit(1);
    }
  else
 return ( q.item[++q.front] );
 /* 將q.rear +1後，此時q.front 指向 queue 之第一個元素，
    再傳回此值 */
}
```

環狀佇列

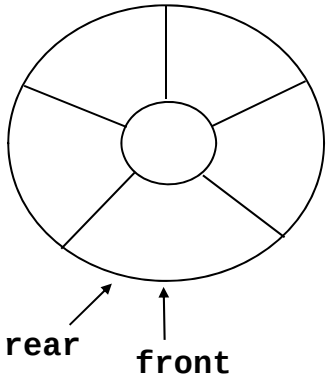
- 線性的陣列因陣列元素位置與註標之對應關係，使其不方便模擬佇列的增加、刪除的消長現象，而用過的空位無法再行利用；
- 不過若將線性的陣列折繞成環狀（circular）陣列，就可再次使用到空出的空間。

假設$n = 5$，利用mod(i) 將循序的數列i轉變為呈環狀計數的數列：

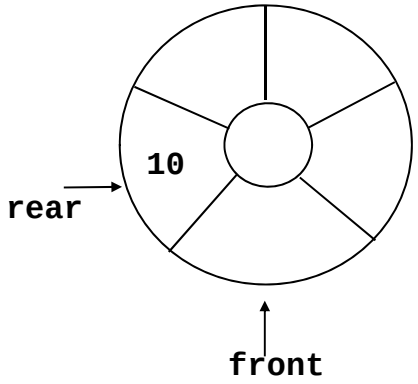
i	0	1	2	3	4	5	6	7	8	9	10	11	12	…
mod(i)	0	1	2	3	4	0	1	2	3	4	0	1	2	

於是原程式的++rear或++front，須分別改為++rear%n和++front%n，其中n為佇列的大小即可有環狀計的註標。

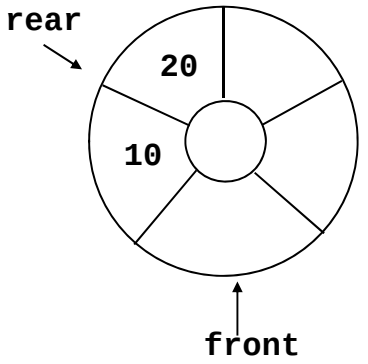
範例



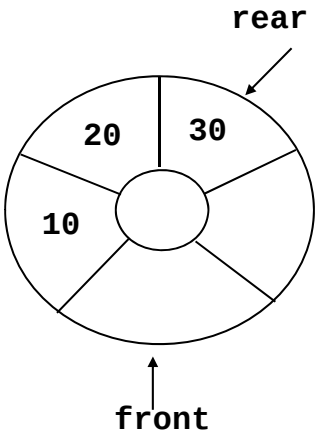
(a)空佇列



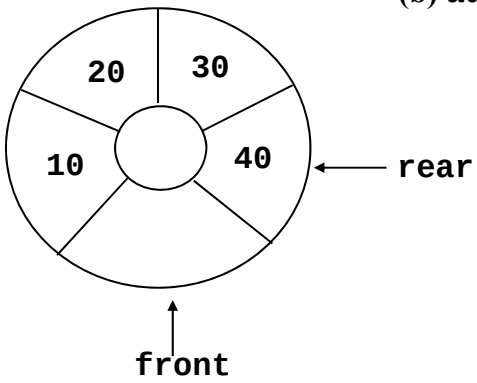
(b) addQ(10)



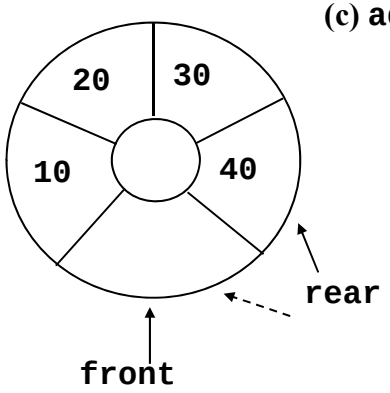
(c) addQ(20)



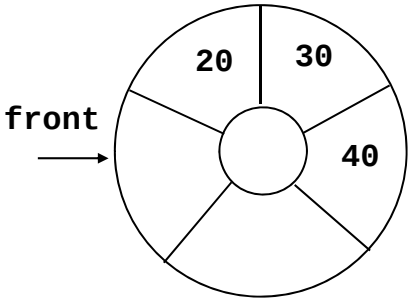
(d) addQ(30)



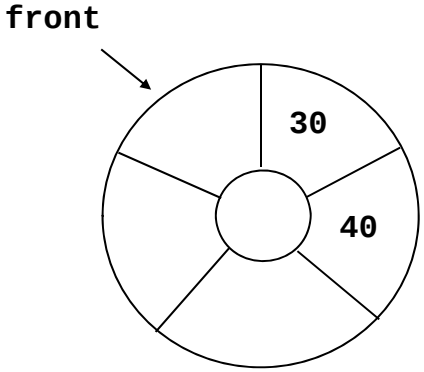
(e) addQ(40)



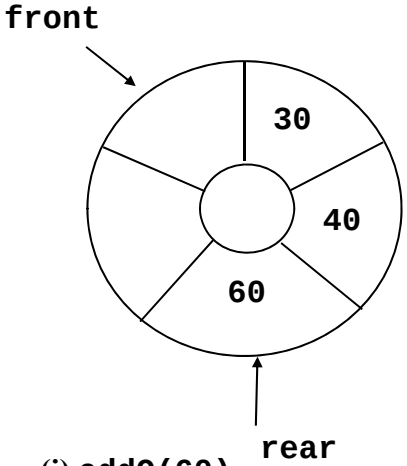
(f) addQ(50)，但佇列已滿



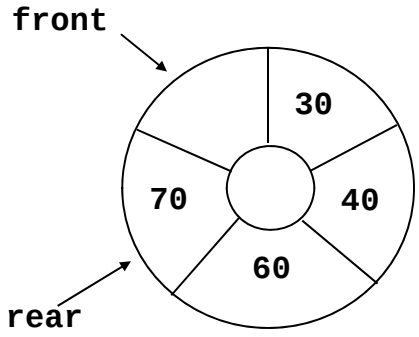
(g) deleteQ()



(h) deleteQ()



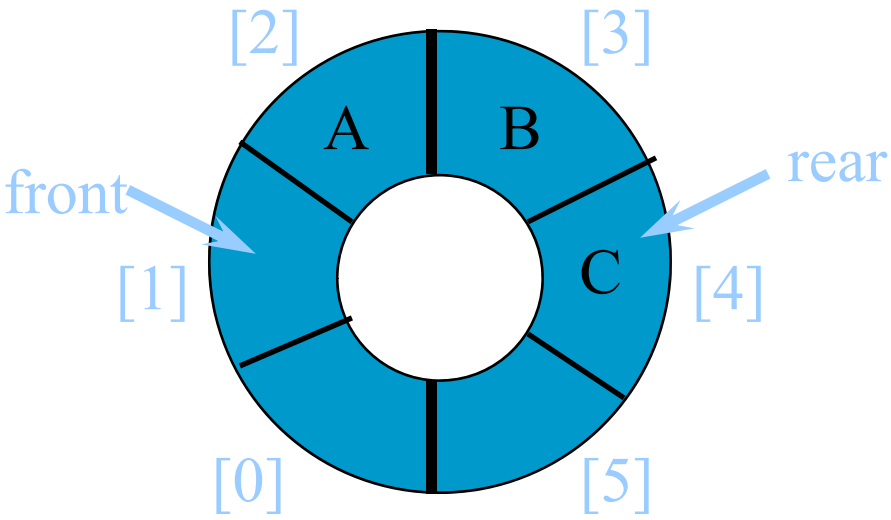
(i) addQ(60)

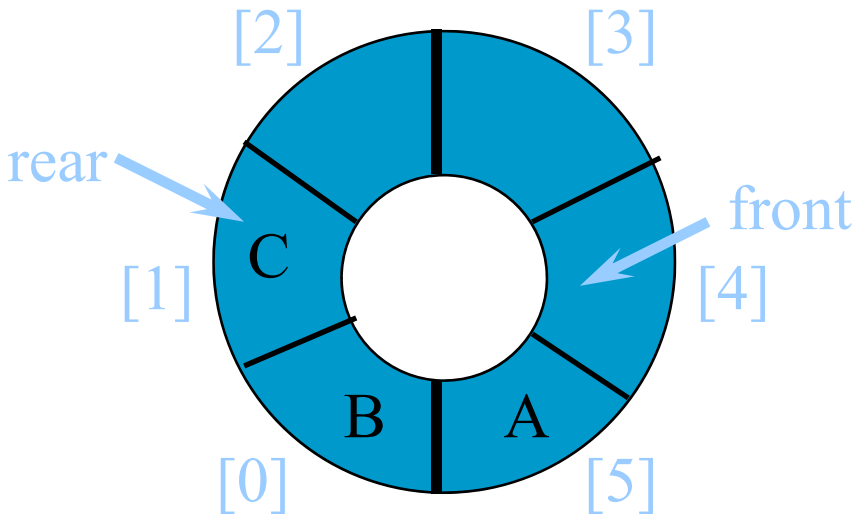


(j) addQ(70)

環狀佇列

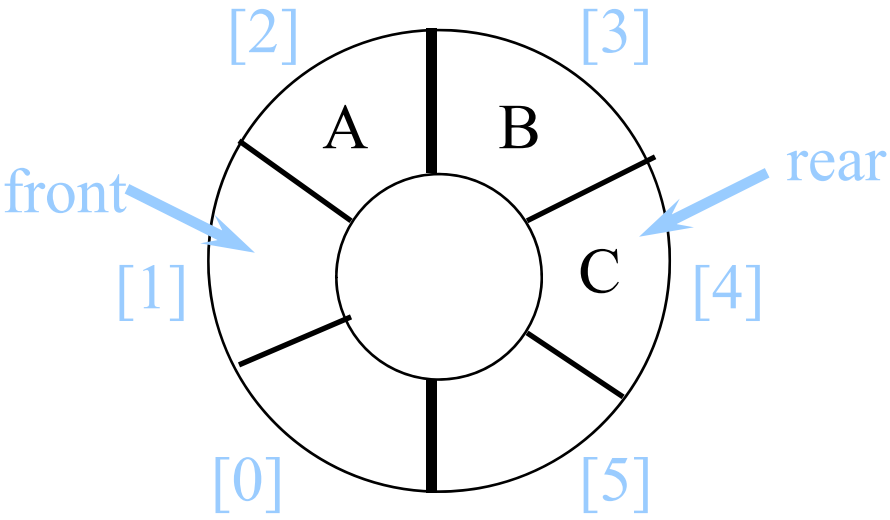
- 使用整數變數 front及rear.
- front永遠指到佇列中第一個元素的前面一位
- rear 指到最後一個元素





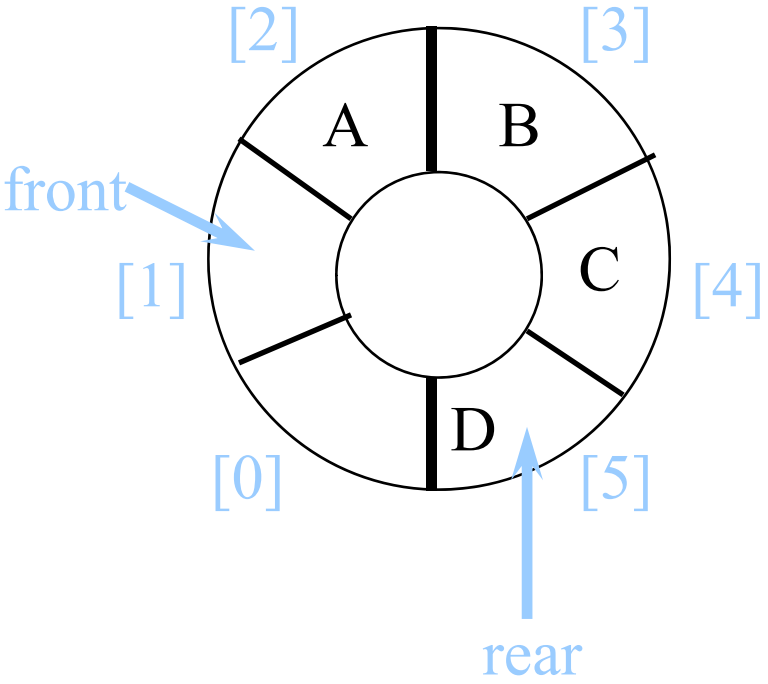
加入新元素

- rear 順時鐘方向加一



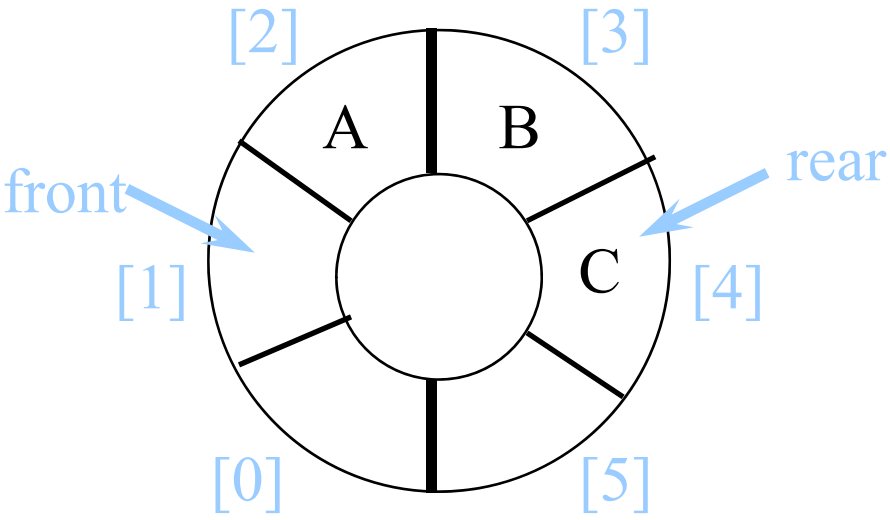
加入新元素

- rear 順時鐘方向加一
- 然後加到 queue[rear].



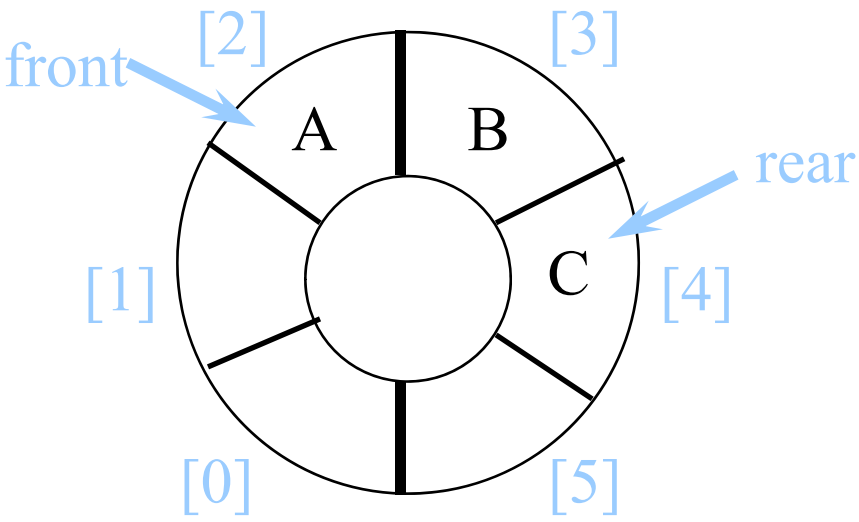
刪除一個元素

- front順時鐘方向加一



刪除一個元素

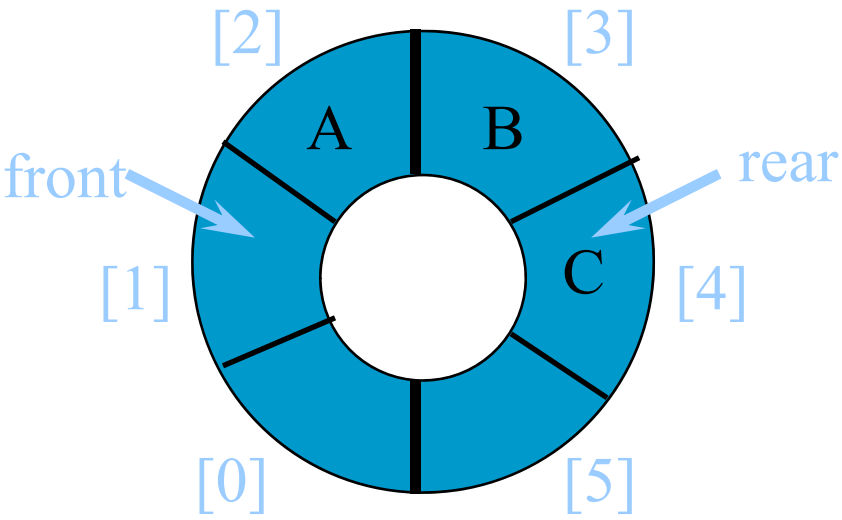
- front順時鐘方向加一
- 取出queue[front]的值



順時鐘方向加一

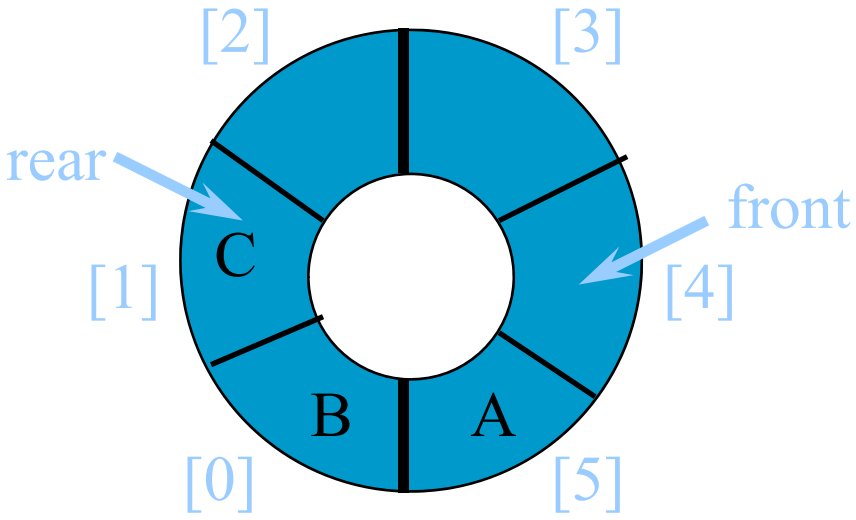
- rear++;

if (rear = = queue.length) rear = 0;

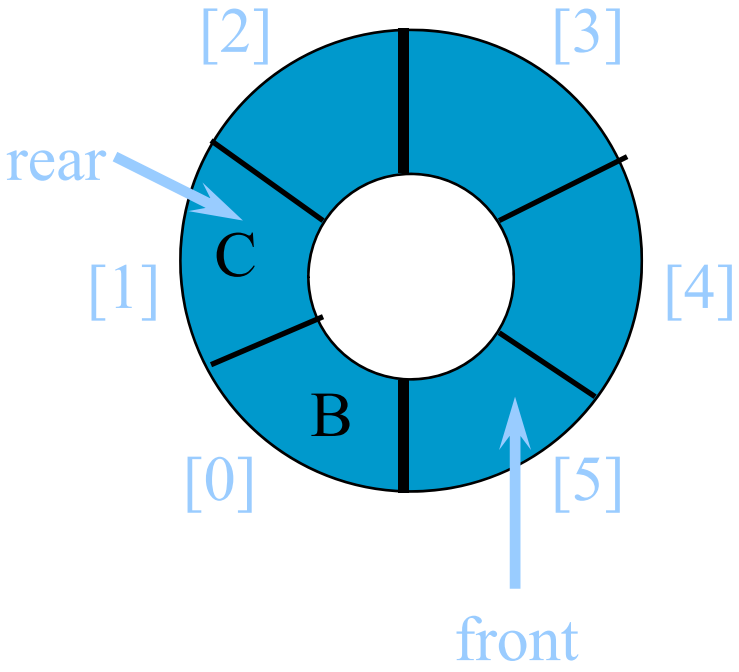


rear = (rear + 1) % queue.length;

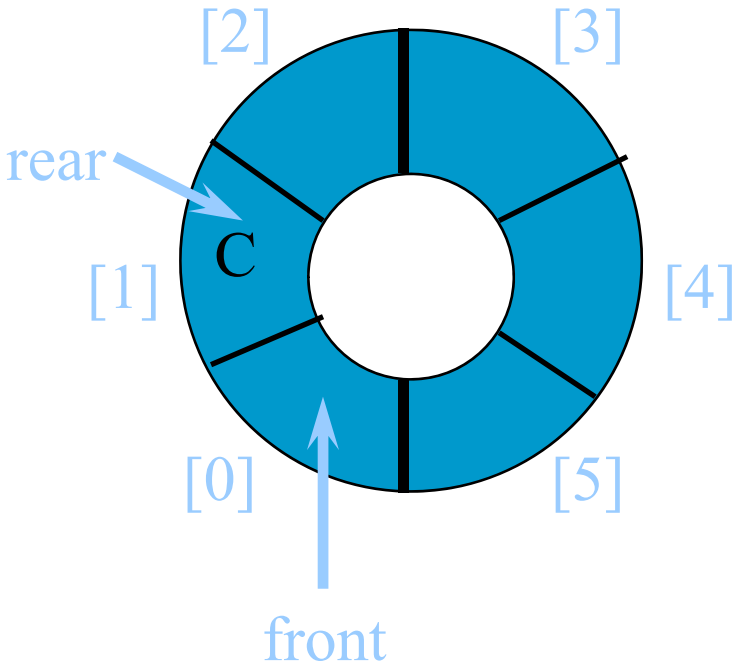
佇列變空



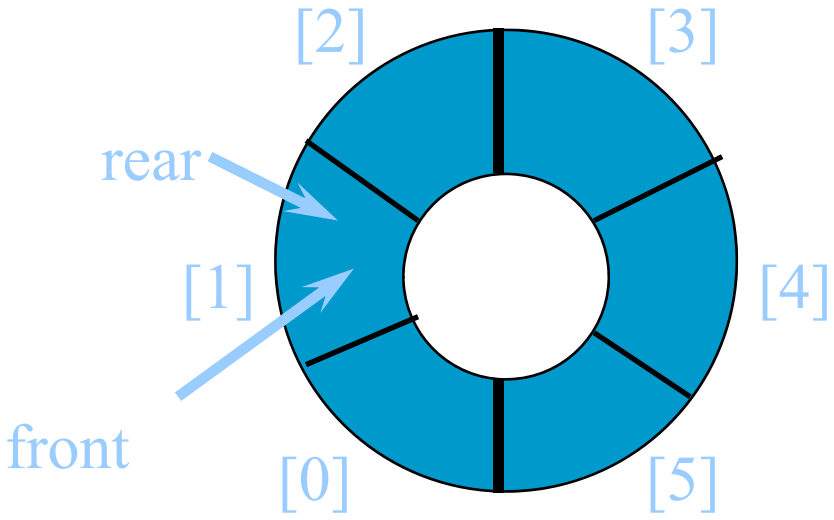
佇列變空



佇列變空

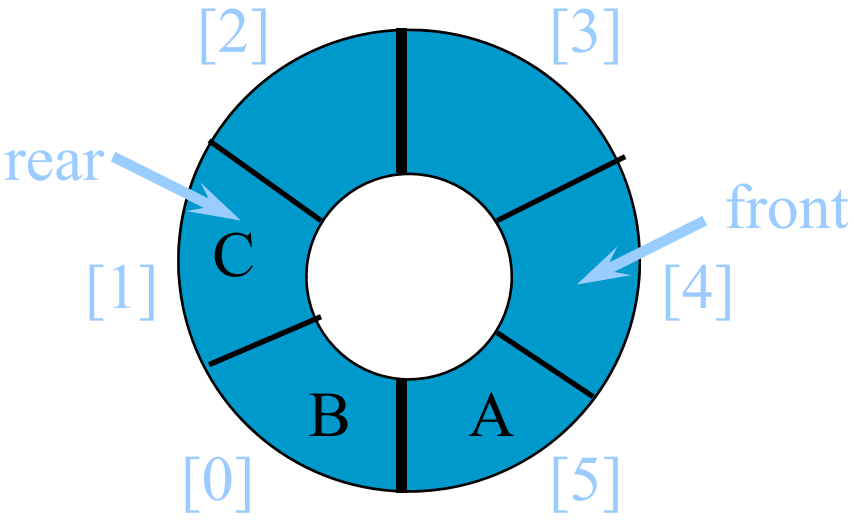


佇列變空

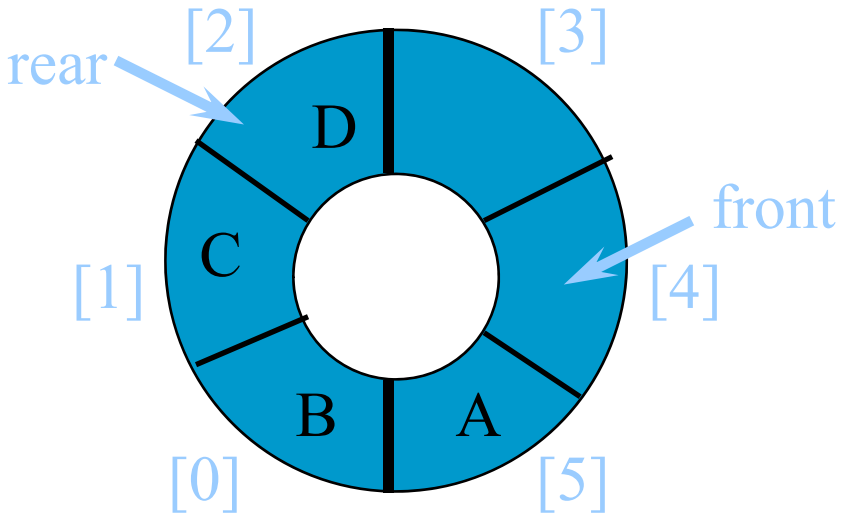


- 佇列變空，則front = rear.
- 佇列剛好建立時是空的
- 因此起始狀況為 front = rear = 0.

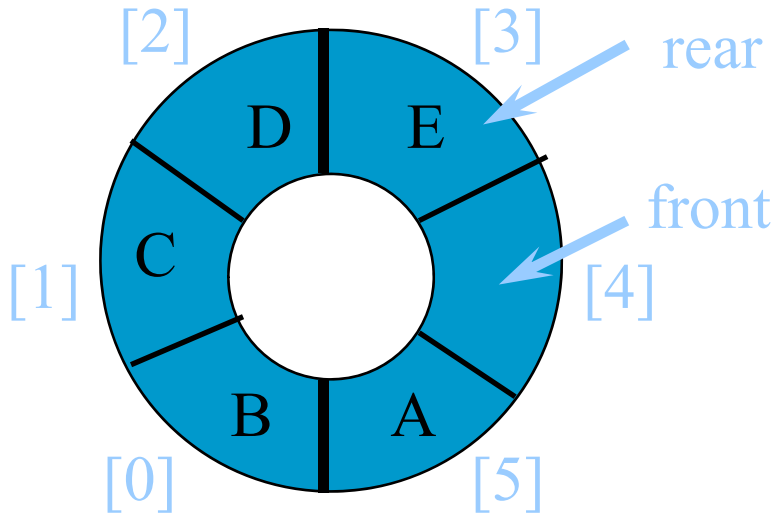
佇列溢滿



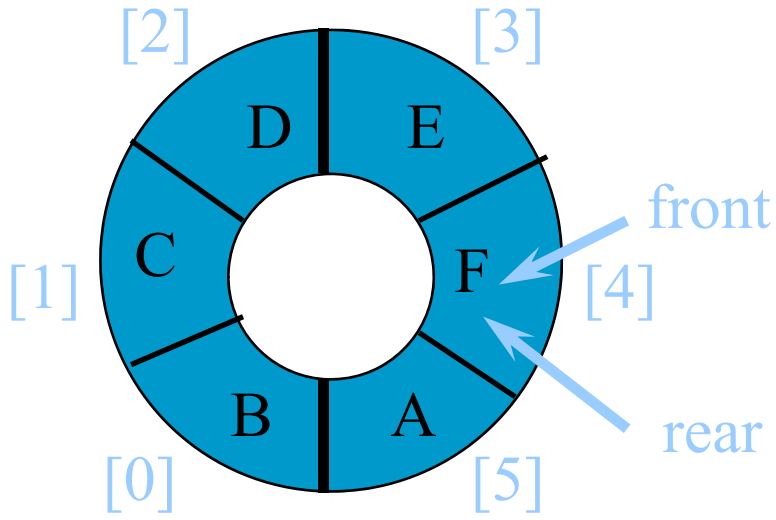
佇列溢滿



佇列溢滿



佇列溢滿



- 在佇列是滿的時候，也會形成front==rear的狀況
- 我們無法分辨這個佇列到底是空的或是滿的

佇列溢滿=佇列空空!!!???

- 解決辦法
  - 限定任何時間時，佇列中最多只能有MaxSize-1個元素
  - 也就是front所指的地方不能存放東西

程式:環狀佇列

```
1 #define maxsize 10;
2 int CQueue[maxsize];
3 int front = 0;
4 int rear = 0;
5
6 void addCQ(int element)
7 {  if (IsCQFull())  CQueueFull ();
8    else  CQueue [++rear%maxsize] = element;
9 }
10
```

front所指的位置為特意保留的空位；
Queue[(front+1)% maxsize]為環狀佇列Queue的前端元素，Queue[rear % maxsize]為環狀佇列Queue的後端元素

程式:　環狀佇列(續)

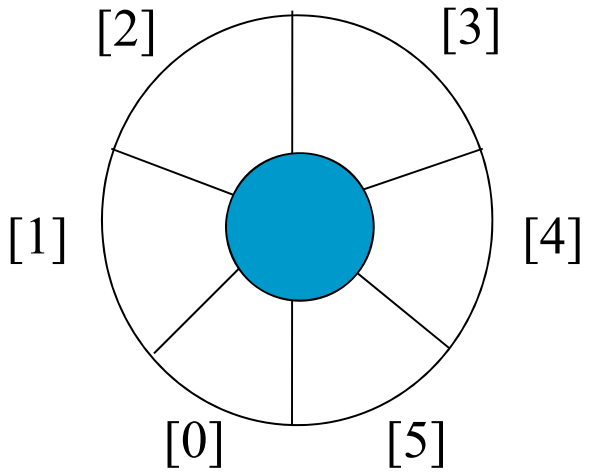
```
11      int deleteCQ()
12      {       if  (IsCQEmpty())
13              {       CQueueEmpty();
14                      return 0;
15              }
16              else  return  CQueue[++front%maxsize];
17      }
18
19      int IsCQEmpty()
20      {       if  (rear == front) return 1;
21              return 0 ;
22      }
23
24      int IsQFull()
25      {       if  ((rear+1)%n == front) return 1;
26              return 0;
27      }
```

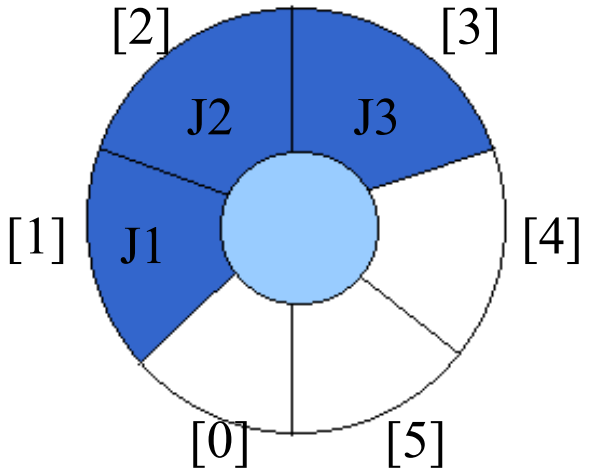
Implementation 2: regard an array as a circular queue

front: one position counterclockwise from the first element

rear: current end

EMPTY QUEUE





front = 0
rear = 0

front = 0
rear = 3

***Figure 3.6:** Empty and nonempty circular queues (p.117)

Problem: one space is left when queue is full

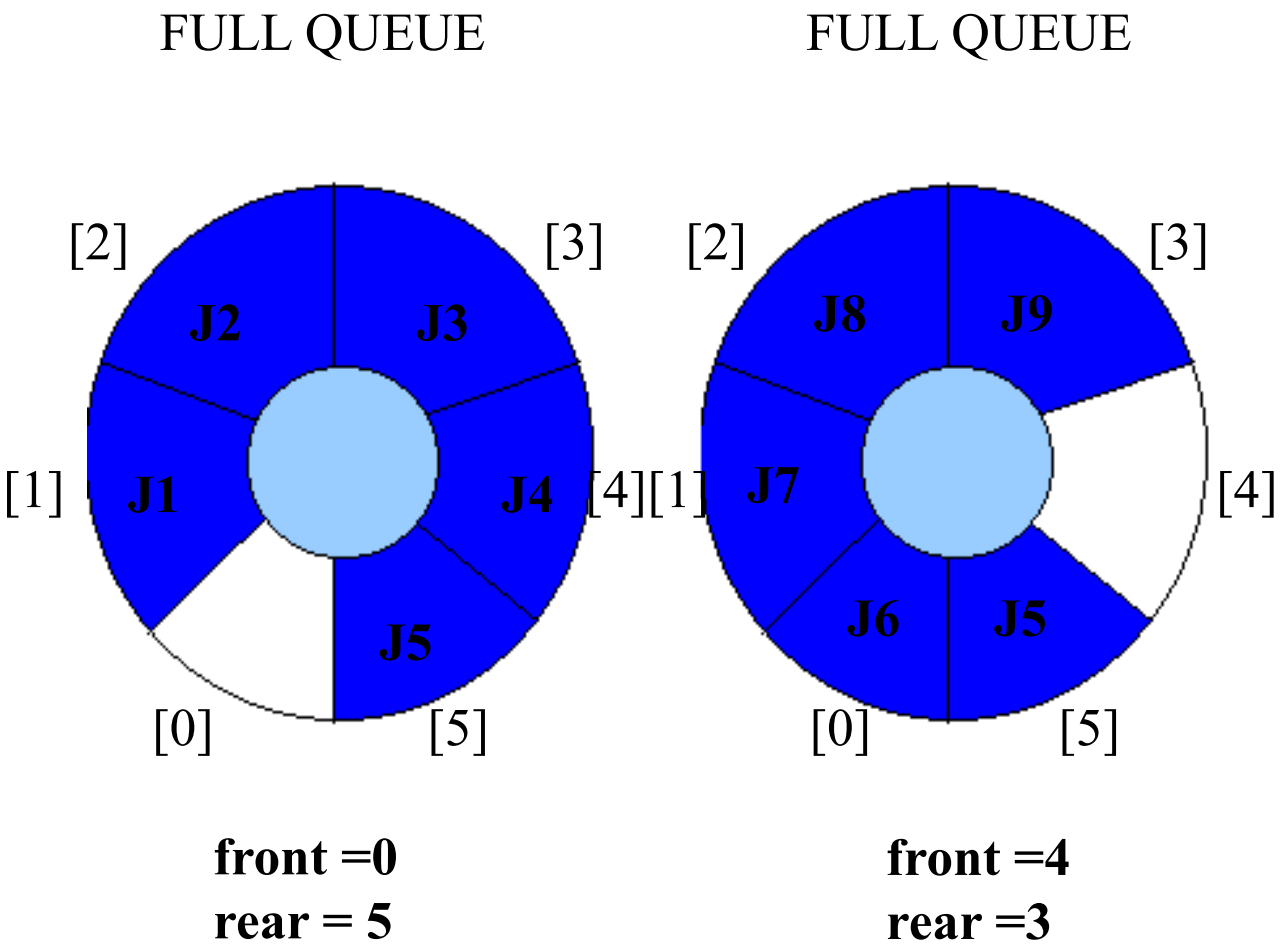


***Figure 3.6:** Full circular queues and then we remove the item (p.117)

Add to a circular queue

```
void addq(int front, int *rear, element item)
{
/* add an item to the queue */
    *rear = (*rear +1) % MAX_QUEUE_SIZE;
     if (front == *rear) /* reset rear and print error */
     return;
   }
     queue[*rear] = item;
}
```

***Program 3.5:** Add to a circular queue (p.116)

Delete from a circular queue

```
element deleteq(int* front, int rear)
{
  element item;
  /* remove front element from the queue and put it in item */
     if (*front == rear)
        return queue_empty( );
                  /* queue_empty returns an error key */
    *front = (*front+1) % MAX_QUEUE_SIZE;
    return queue[*front];
}
```

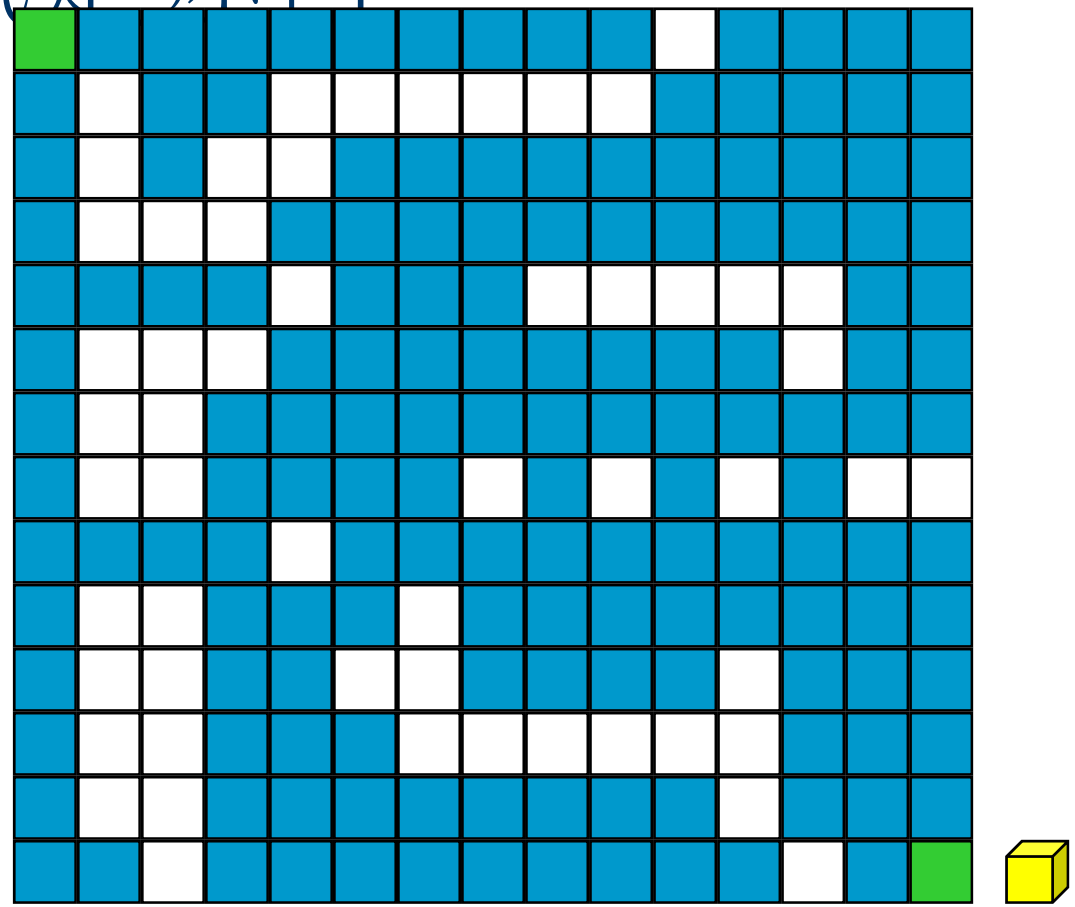
***Program 3.6:** Delete from a circular queue (p.116)

作業

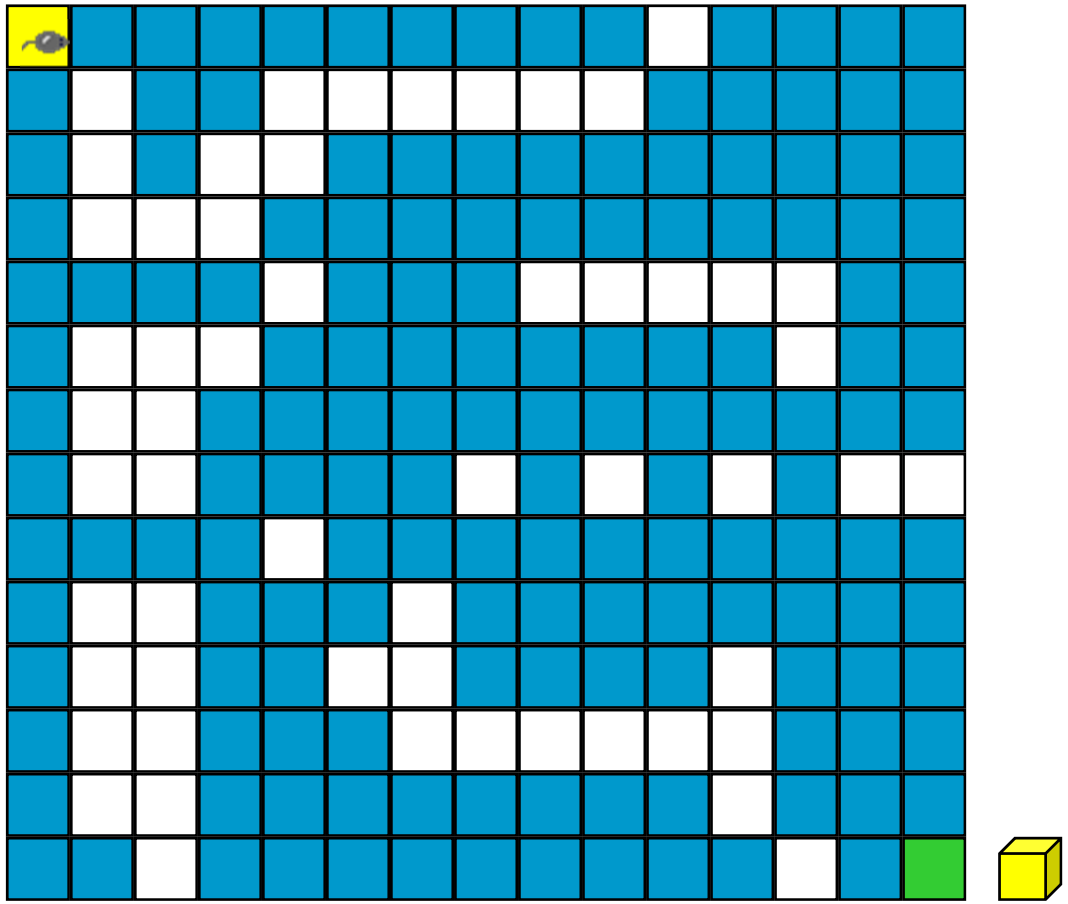
- P119: ex2

跳到下一節

老鼠走迷宫

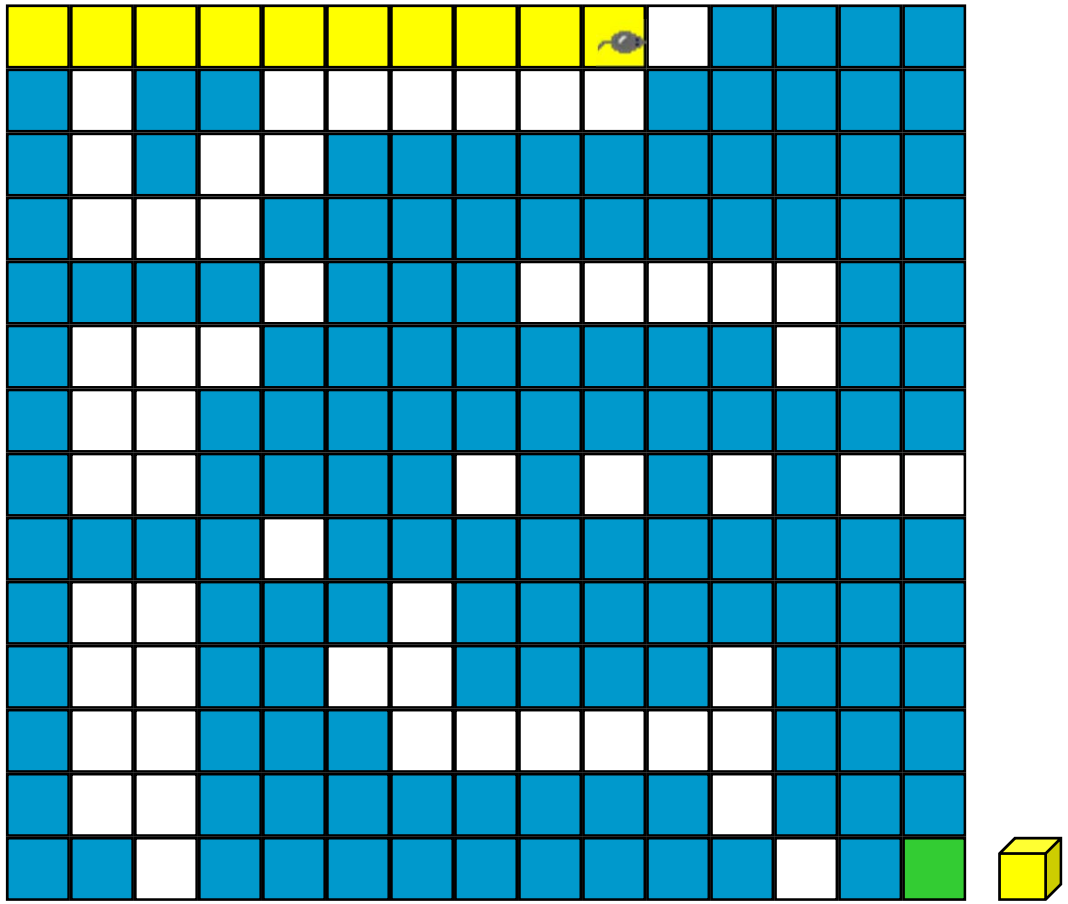


老鼠走迷宮



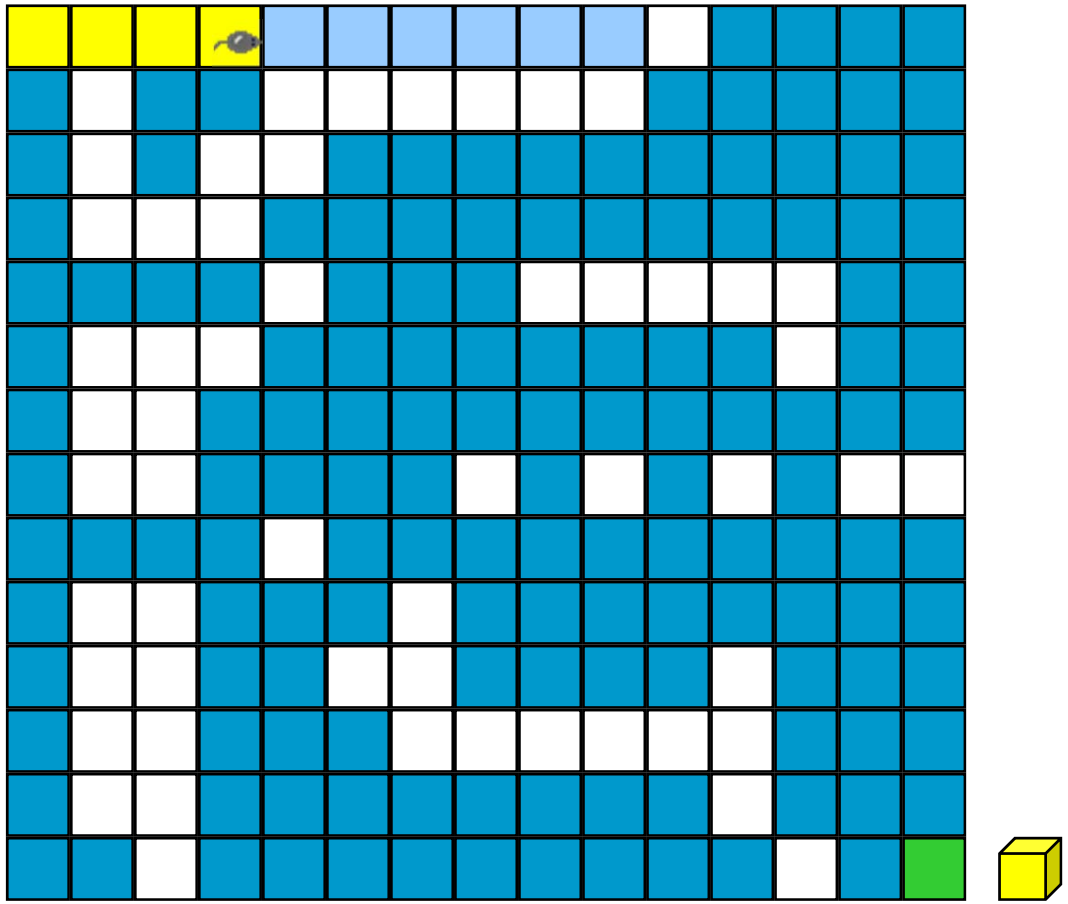
- 移動順序：右，下，左，上
- 走過的位置堵住以免重複

老鼠走迷宫



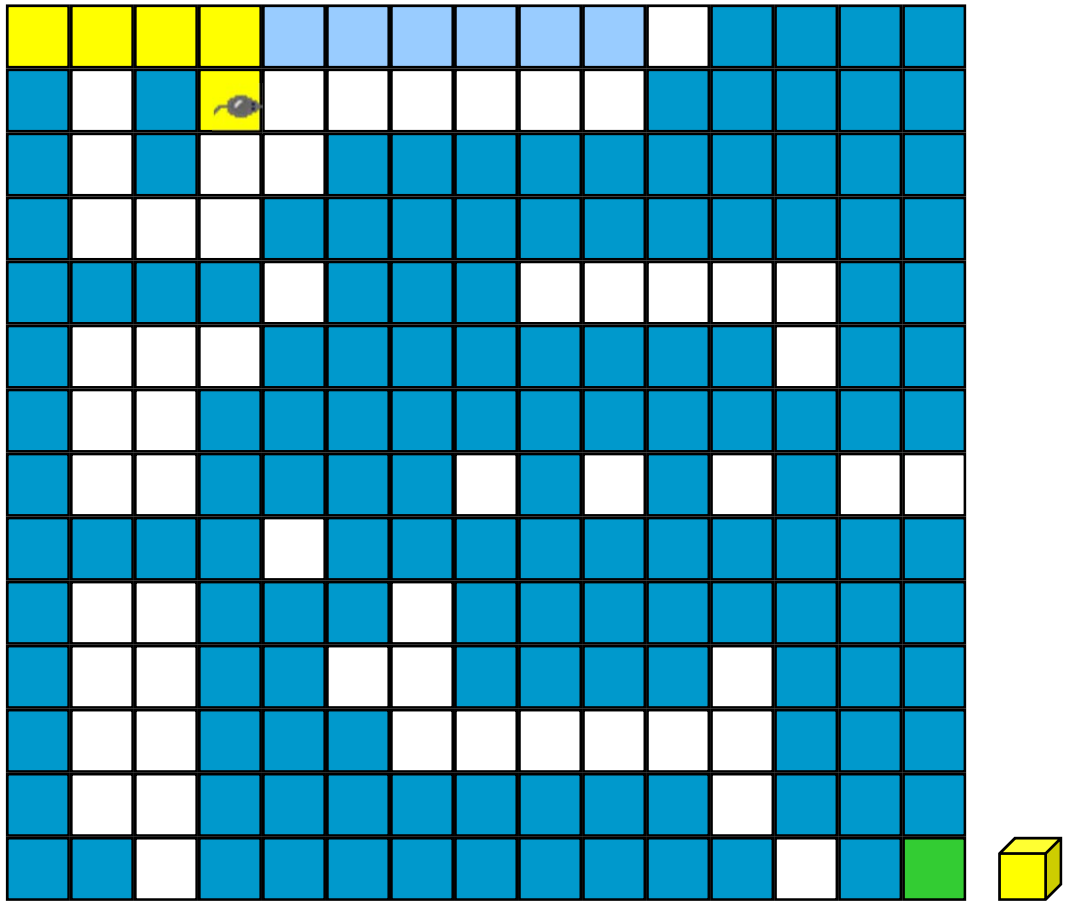
- 往回走直到有路可走

老鼠走迷宮



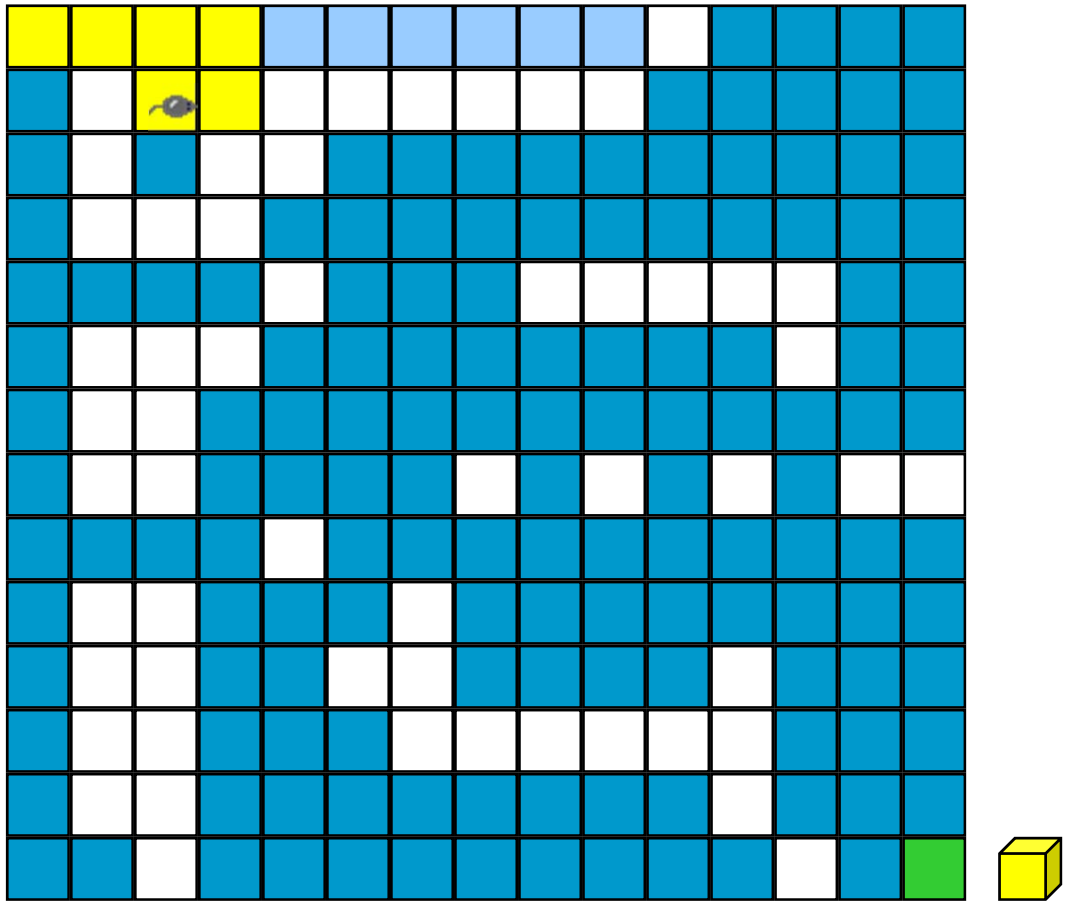
- 往下移動.

老鼠走迷宮



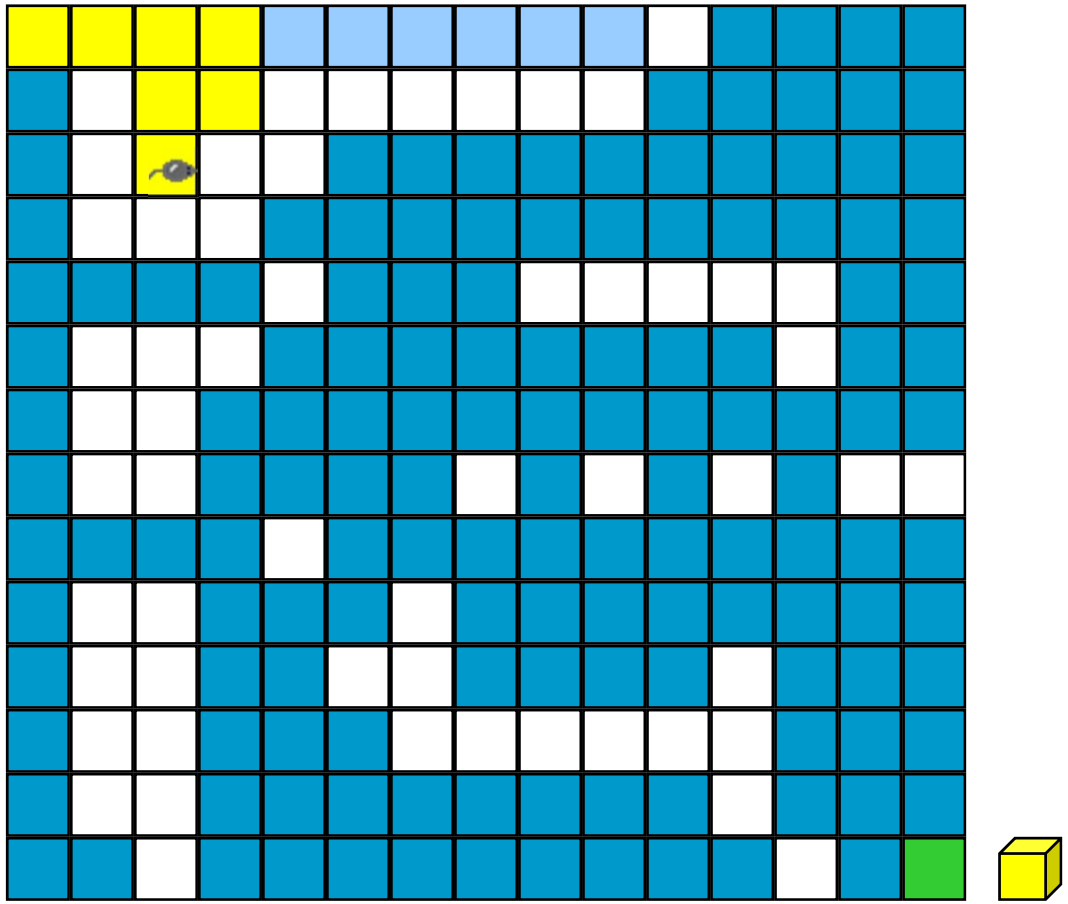
- 往左移動.

老鼠走迷宮



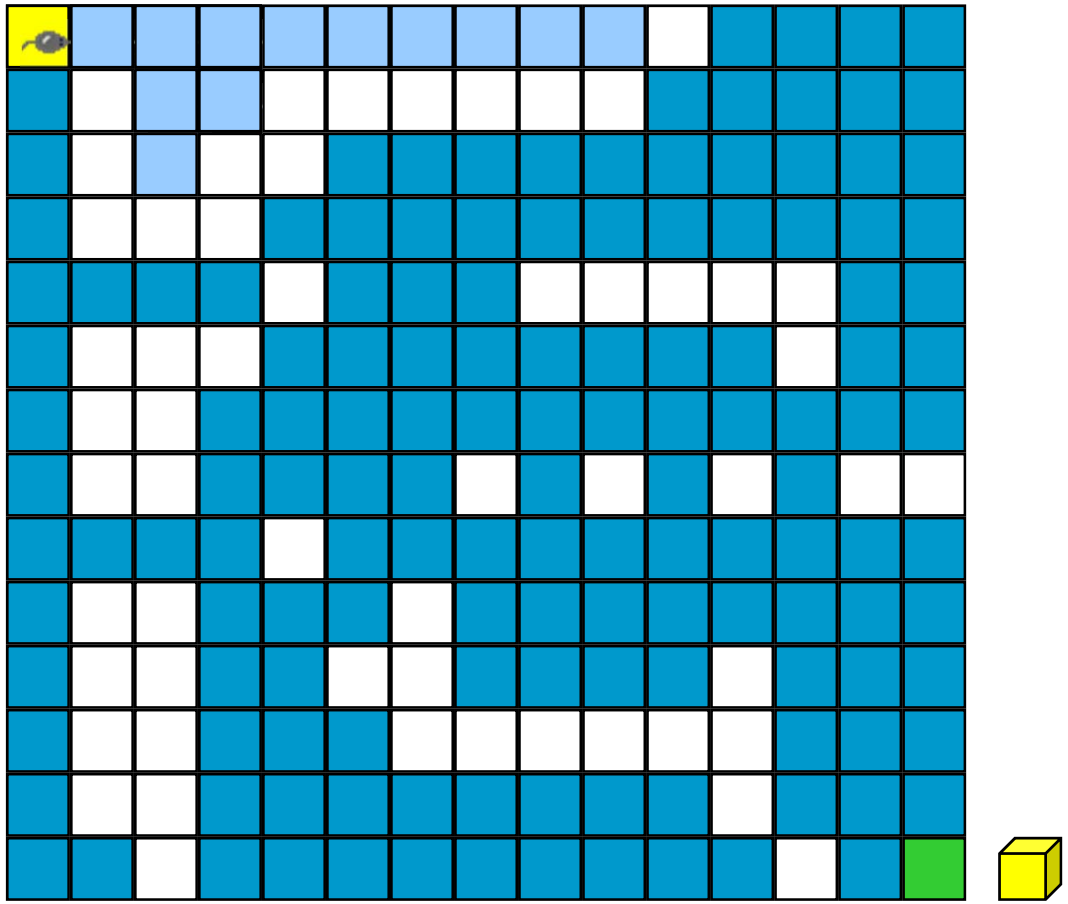
- 往下移動.

老鼠走迷宮



- 往回走，直到可以往前移動的方塊

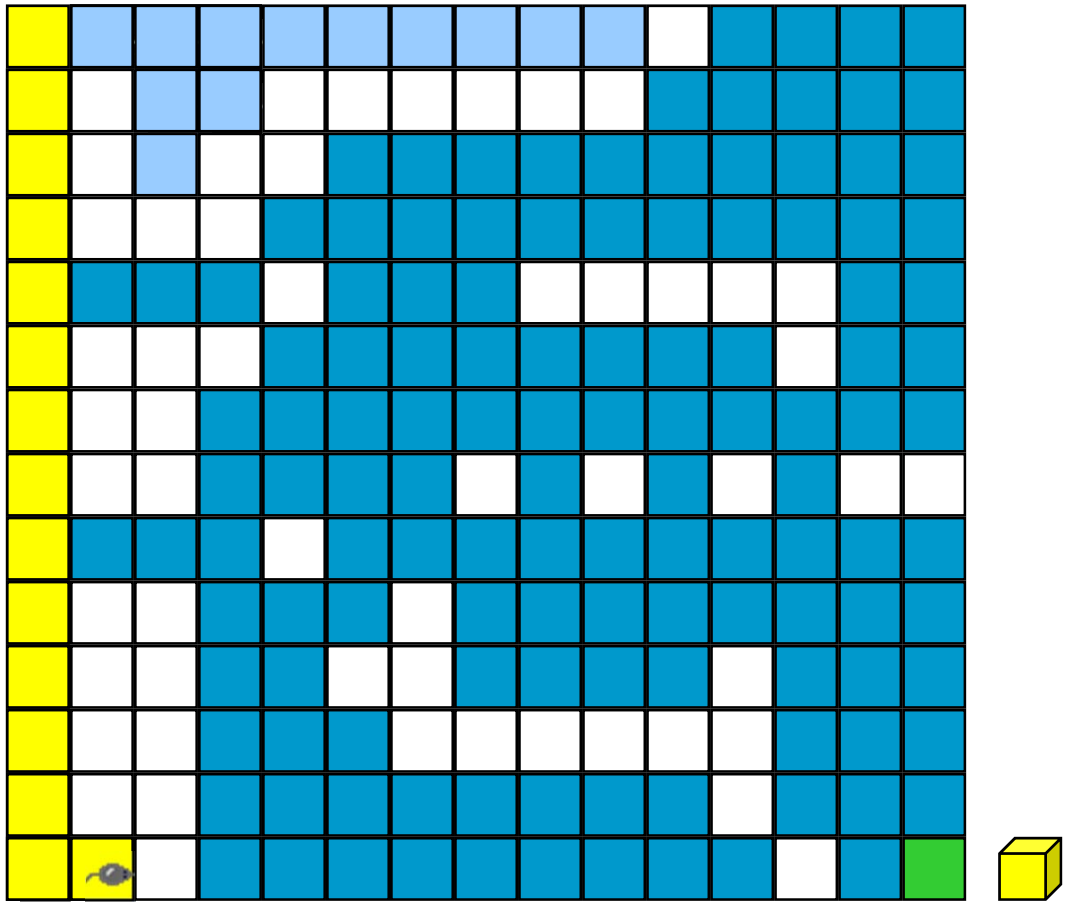
老鼠走迷宮



- 往回走，直到可以往前移動的方塊

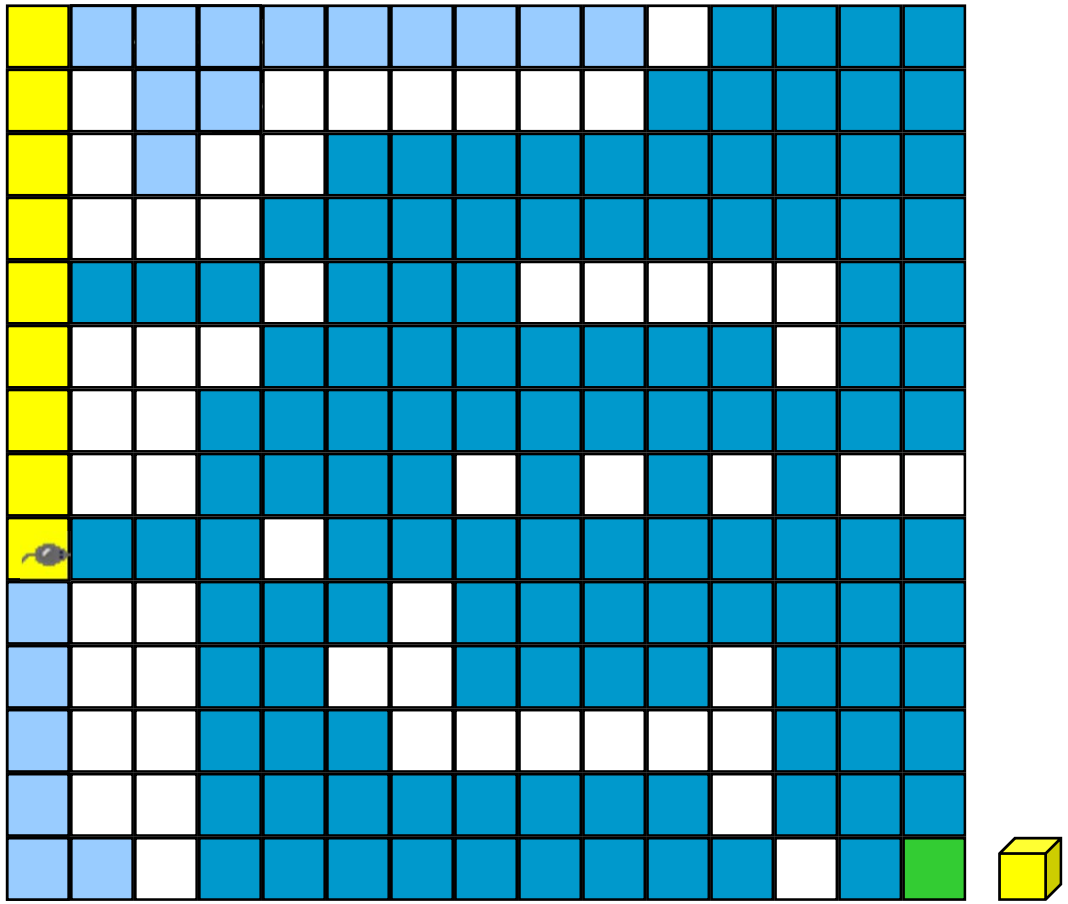
- 往下移動.

老鼠走迷宮



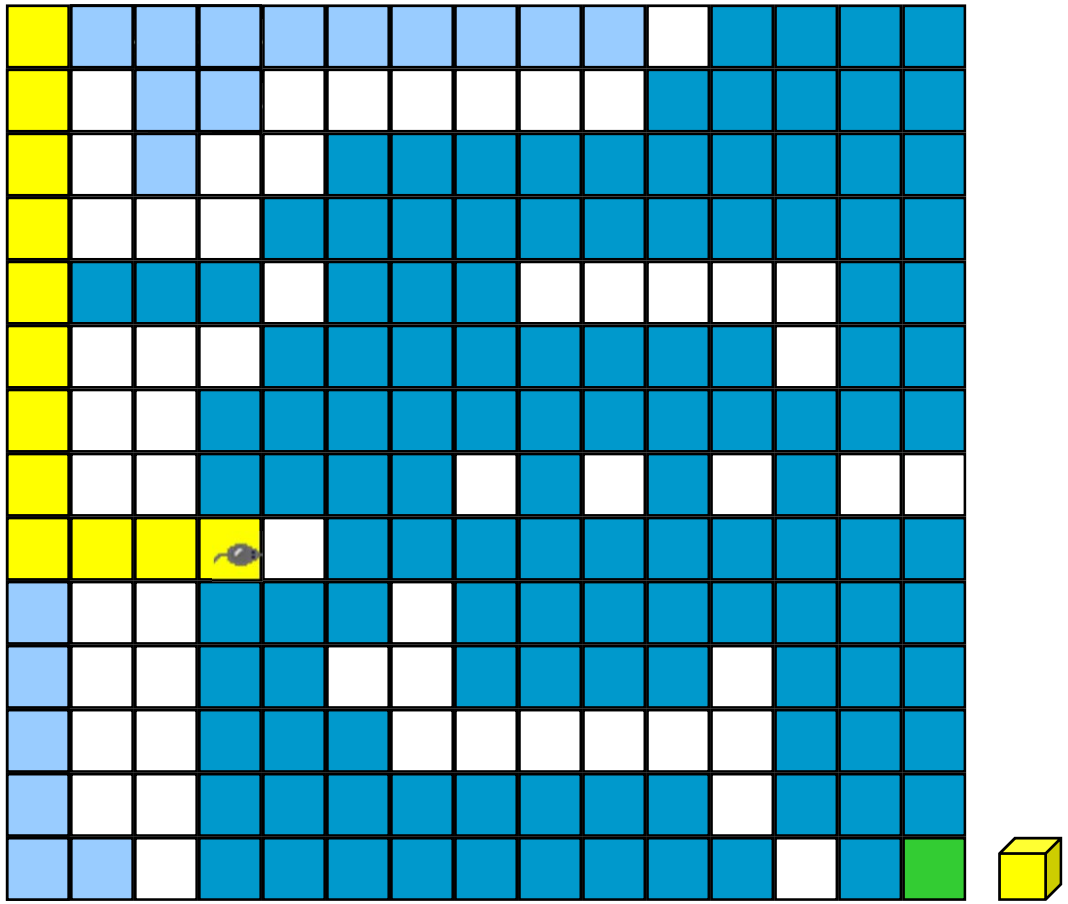
- 往右移動.
- 回溯(Backtrack).

老鼠走迷宮



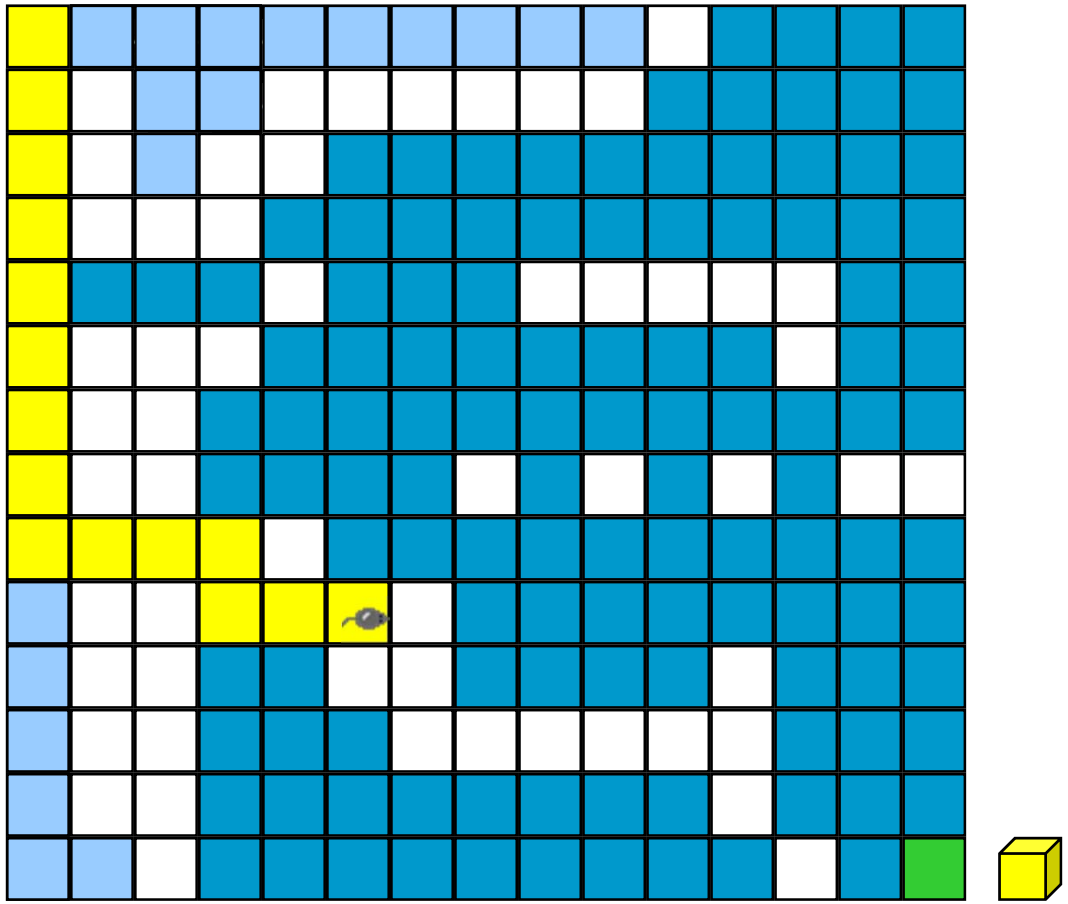
- 往右移動.

老鼠走迷宮



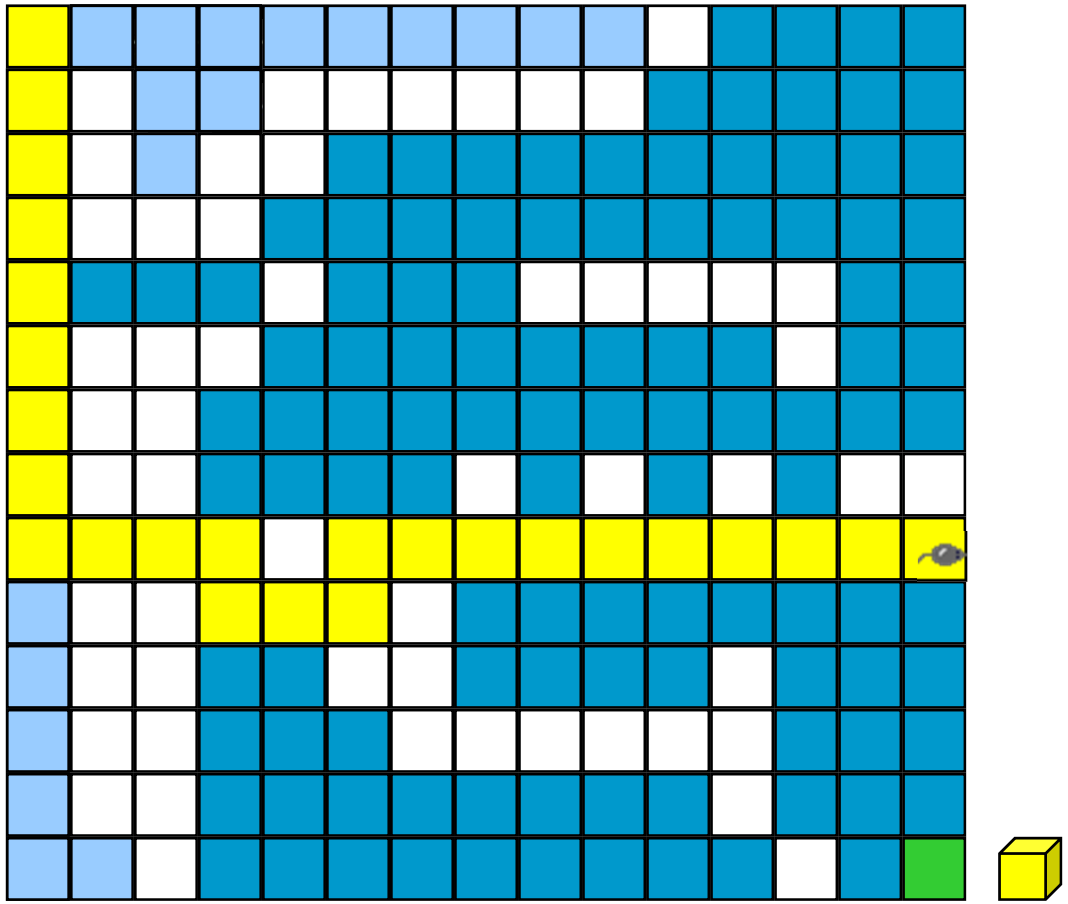
- 往下一格然後往右移動.

老鼠走迷宮



- 往上一格然後往右移動.

老鼠走迷宮



- 往下移動到出口並且吃起司.
- 用堆疊處理從迷宮的入口處到目前位置的路徑.

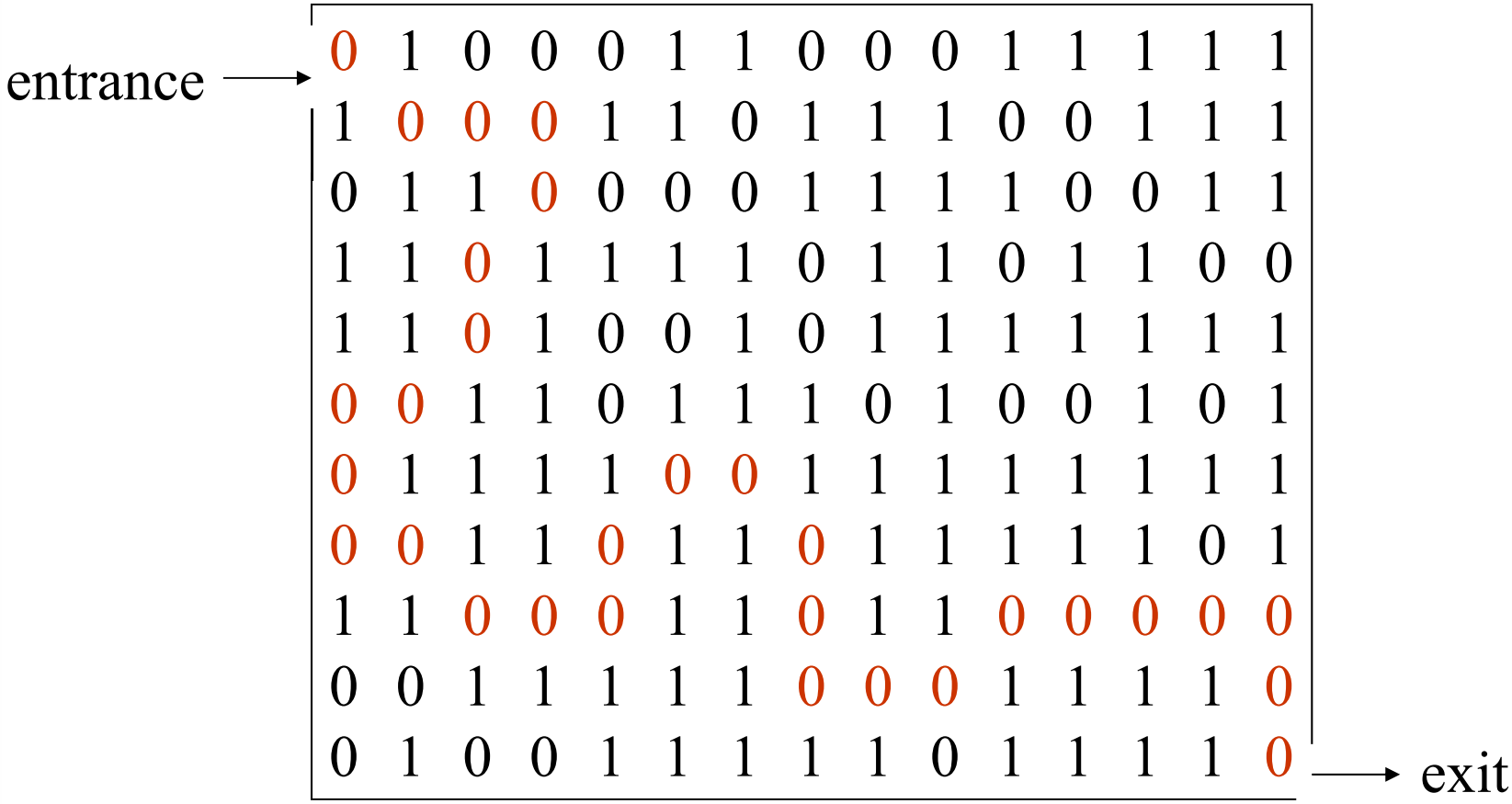
3.5 A Mazing Problem (老鼠走迷宮)

- 老鼠走迷宮是個有趣的心理實驗，把老鼠放入迷宮中，老鼠可靠單純的嚐試錯誤 (trial and error) 法，即找到出口。這種嚐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，我們或稱之為「窮舉」 (enumeration) 的策略。
- 我們可用電腦程式模擬老鼠走迷宮的過程，採用的策略即為嚐試錯誤法，但是曾經走錯的路我們不應再次嚐試，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得多。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 。

老鼠走迷宮(續)

- 首先應先設想必需的資料結構：
- 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？）
- 各位應養成思考問題時，同時溶入資料結構考量的能力。

A Mazing Problem



1: blocked path 0: through path

***Figure 3.8:** An example maze(p.113)

a possible representation

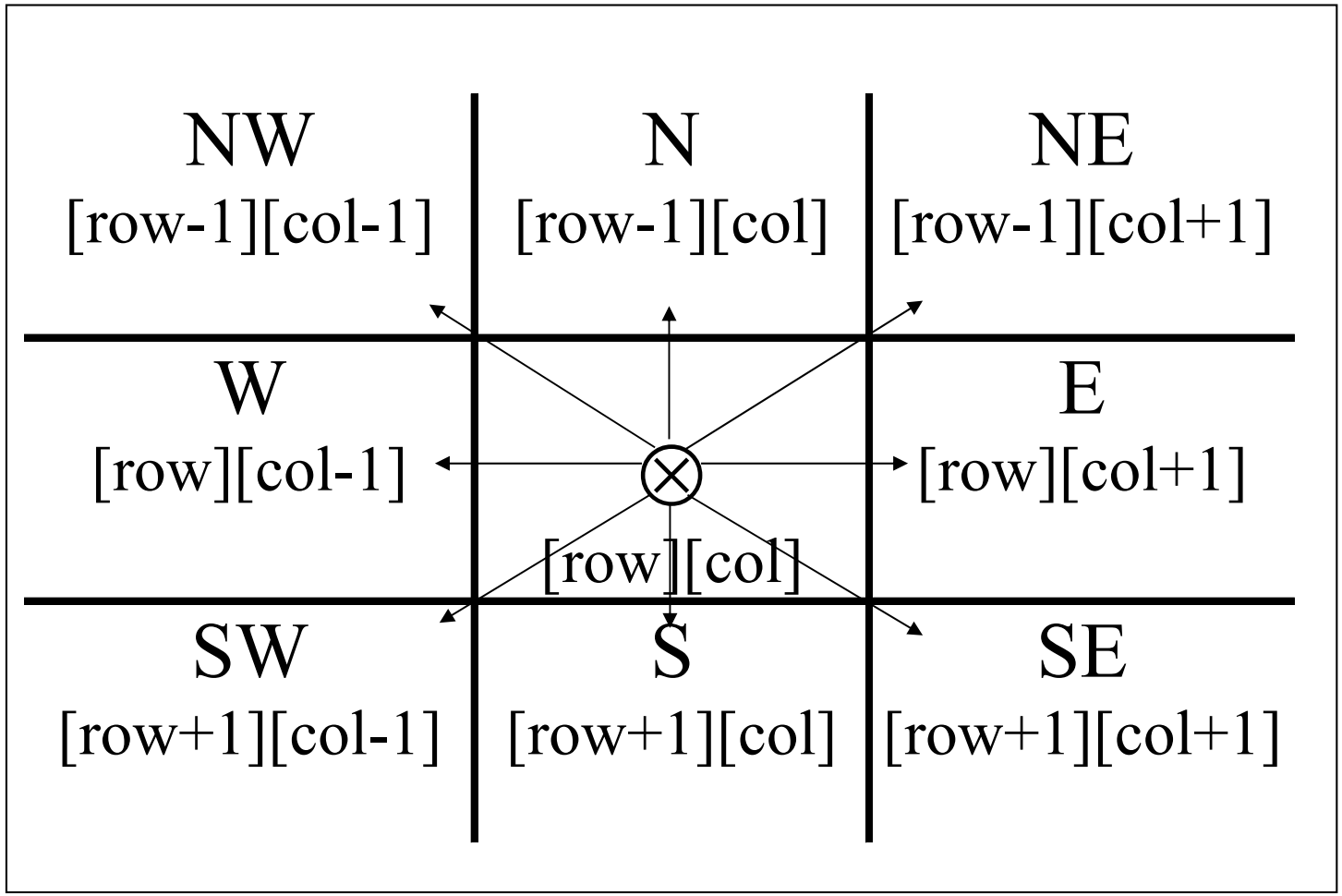


***Figure 3.9:** Allowable moves (p.113)

a possible implementation

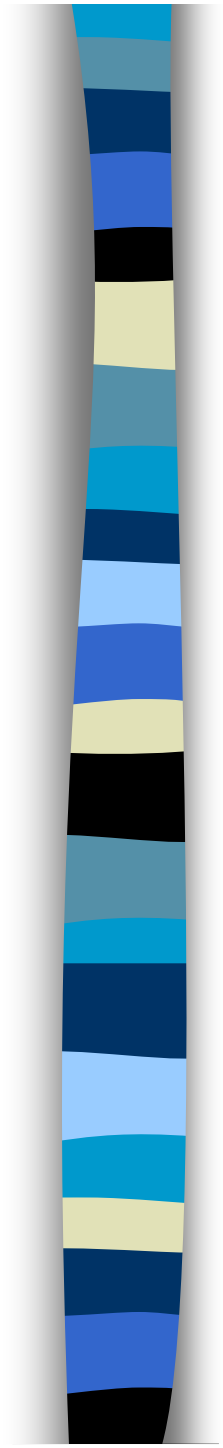
```
typedef struct {
        short int vert;
        short int horiz;
        } offsets;
offsets move[8]; /*array of moves for each direction*/
```

```
next_row = row + move[dir].vert;
next_col  = col  +  move[dir].horiz;
```

Name	Dir	move[dir].vert	move[dir].horiz
N	0	-1	0
NE	1	-1	1
E	2	0	1
SE	3	1	1
S	4	1	0
SW	5	1	-1
W	6	0	-1
NW	7	-1	-1

Use stack to keep pass history

```
#define MAX_STACK_SIZE 100
        /*maximum stack size*/
typedef struct {
        short int row;
        short int col;
        short int dir;
        } element;
element stack[MAX_STACK_SIZE];
```



Initialize a stack to the maze's entrance coordinates and direction to north;
while (stack is not empty){
  /* move to position at top of stack */
<row, col, dir> = delete from top of stack;
while (there are more moves from current position) {
    <next_row, next_col > = coordinates of next move;
    dir = direction of move;
    if ((next_row == EXIT_ROW)&& (next_col == EXIT_COL))
      success;
    if (maze[next_row][next_col] == 0 &&
      mark[next_row][next_col] == 0) {

```
        /* legal move and haven't been there */
            mark[next_row][next_col] = 1;
            /* save current position and direction */
            add <row, col, dir> to the top of the stack;
            row = next_row;
            col  = next_col;
            dir  = north;
        }
    }
}
printf("No path found\n");
```

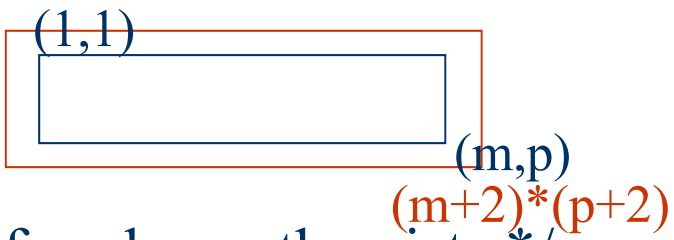
***Program 3.11:** Initial maze algorithm (p.126)

The size of a stack?

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}_{m*p}$$

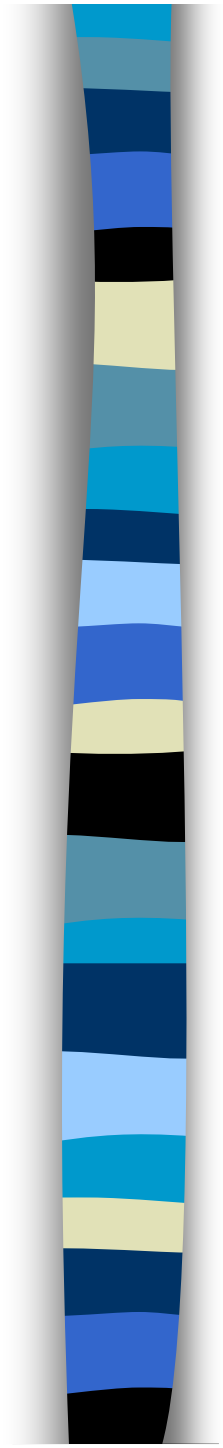
mp --> ⌈ m/2⌉ p, mp --> ⌈ p/2⌉ m

***Figure 3.11:** Simple maze with a long path (p.127)




```
void path (void)
{
/* output a path through the  maze if such a path exists */
   int i, row, col, next_row, next_col, dir, found = FALSE;
   element position;
   mark[1][1] = 1; top =0;
   stack[0].row = 1; stack[0].col = 1; stack[0].dir = 1;
   while (top > -1 && !found) {
      position = delete(&top);
      row = position.row; col = position.col;
      dir  = position.dir;
      while (dir < 8 && !found) {
            /*move in direction dir  */
            next_row = row + move[dir].vert;
            next_col  = col + move[dir].horiz;
```

	0	
7	**N**	**1**
6 W		**E 2**
5	**S**	**3**
	4	



```
        if (next_row==EXIT_ROW && next_col==EXIT_COL)
            found = TRUE;
        else if ( !maze[next_row][next_col] &&
                    !mark[next_row][next_col] {
            mark[next_row][next_col] = 1;
            position.row = row; position.col = col;
            position.dir = ++dir;
            add(&top, position);
            row = next_row; col = next_col; dir = 0;
         }
         else ++dir;
    }
}
```

```
    if (found) {
        printf("The path is :\n");
        printf("row col\n");
        for (i = 0; i <= top; i++)
            printf(" %2d%5d", stack[i].row, stack[i].col);
        printf("%2d%5d\n", row, col);
        printf("%2d%5d\n", EXIT_ROW, EXIT_COL);
     }
    else printf("The maze does not have a path\n");
}
```

***Program 3.12:** Maze search function (p.128)

3.6 運算式的轉換和求值 (Evaluation of Expressions)

- 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。

運算式的種類、計算與轉換-說明

- 算術運算式是使用「運算子」（Operators）和「運算元」（Operands）所組成，如下所示：

  A+B

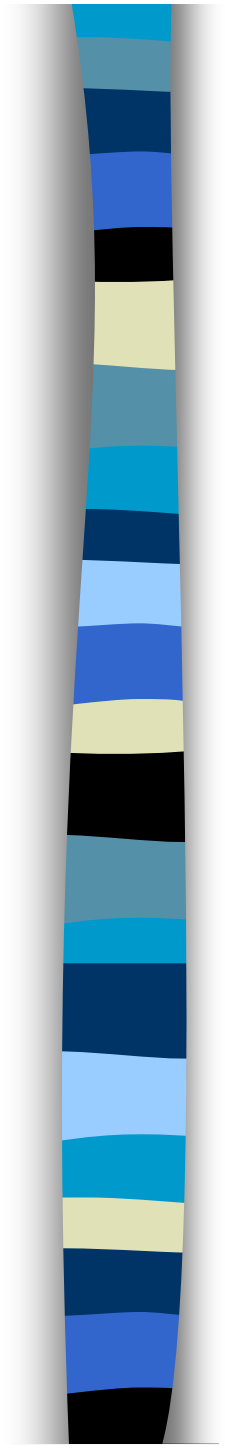
  A*B+C

- 上述運算式的A、B和C是運算元，＋和*是運算子。

運算式的種類、計算與轉換-種類

運算式種類依據運算子位在運算式中的位置，可以分為三種，如下所示：

- **中序表示法（Infix）**：運算式中的運算子是位在兩個運算元之間，例如：A+B和A*B+C。
- **前序表示法（Prefix）**：運算子位在兩個運算元之前，例如：+AB和+*ABC。
- **後序表示法（Postfix）**：運算子位在兩個運算元之後，例如：AB+和AB*C+。



user **compiler**

Infix	Postfix
2+3*4	234*+
a*b+5	ab*5+
(1+2)*7	12+7*
a*b/c	ab*c/
(a/(b-c+d))*(e-a)*c	abc-d+/ea-*c*
a/b-c+d*e-a*c	ab/c-de*+ac*-

***Figure 3.13:** Infix and postfix notation (p.120)

Postfix: no parentheses, no precedence

運算式的種類、計算與轉換-優先順序

- 中序表示法在執行運算式的計算和轉換前，需要注意運算子的「優先順序」（Priority），運算子的優先順序，如下表所示：

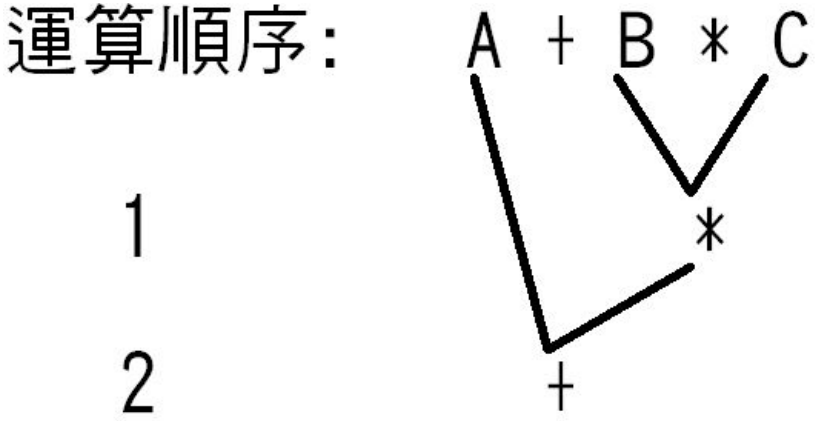
運算子	優先順序
括號( )	高 ↓ 低
負號-	
乘* 除/ 餘數%	
加+ 減-	

運算式的種類、計算與轉換-中序計算

- 運算式在先計算B*C後才和A相加，因為運算子*的優先順序大於+。如果中序運算式不考慮運算子優先順序，同一個中序運算式可能產生不同的運算結果，如下所示：

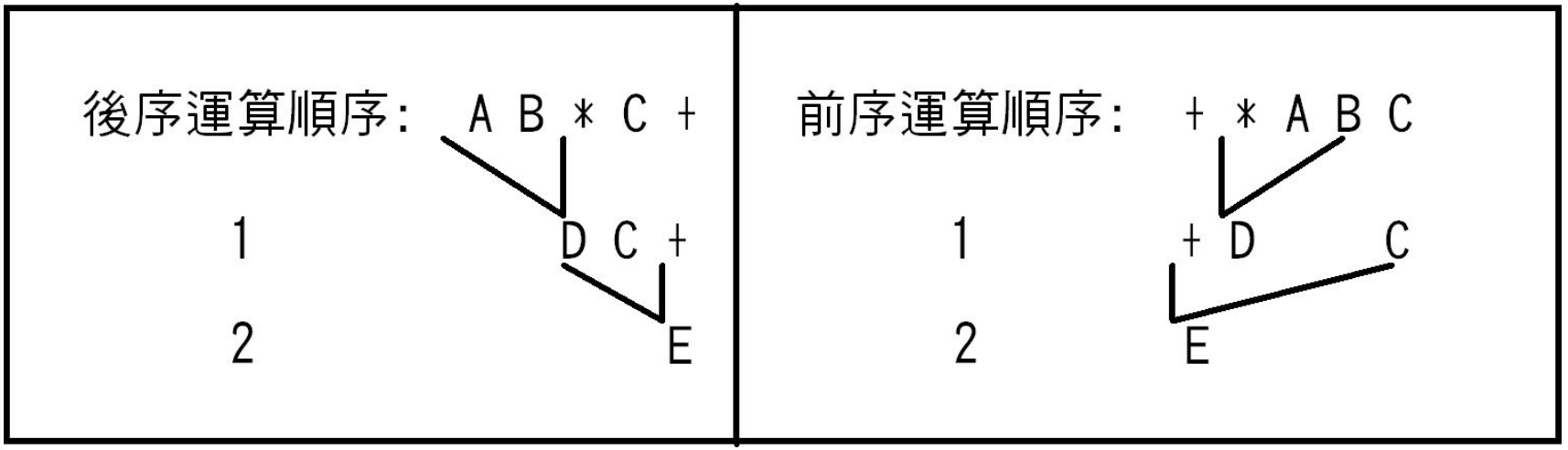
  A+B*C：先算A+B，再和C相乘

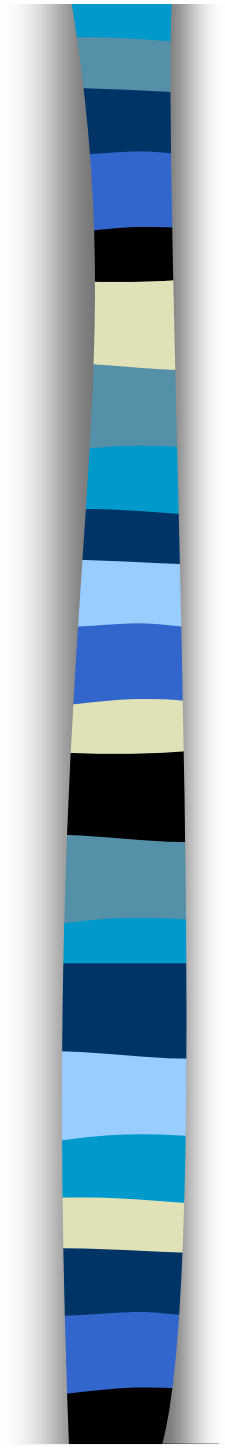
  A+B*C：先算B*C再加上A



運算式的種類、計算與轉換-前序與後序計算

- 前序和後序表示法，就不需要考慮運算子的優先順序，如下所示：





Token	Operator	Precedence[1]	Associativity
( ) [ ] -> .	function call array element struct or union member	17	left-to-right
-- ++	increment, decrement[2]	16	left-to-right
-- ++ ! - - + & * sizeof	decrement, increment[3] logical not one's complement unary minus or plus address or indirection size (in bytes)	15	right-to-left
(type)	type cast	14	right-to-left
* / %	mutiplicative	13	Left-to-right

+ -	binary add or subtract	12	left-to-right
<< >>	shift	11	left-to-right
> >= < <=	relational	10	left-to-right
== !=	equality	9	left-to-right
&	bitwise and	8	left-to-right
^	bitwise exclusive or	7	left-to-right
|	bitwise or	6	left-to-right
&&	logical and	5	left-to-right
▣	logical or	4	left-to-right

?:	conditional	3	right-to-left
= += -= /= *= %= <<= >>= &= ^= ⊠	assignment	2	right-to-left
,	comma	1	left-to-right

1.The precedence column is taken from Harbison and Steele.
2.Postfix form
3.prefix form

***Figure 3.12:** Precedence hierarchy for C (p.130)

運算式的種類、計算與轉換-轉換(方法1)

- 運算式轉換分為中序轉前序和中序轉後序表示法，其轉換步驟十分相似，其差異只在運算子是位在運算元前或後。例如：中序運算式，如下：

  A*(B+C)

- 上述運算式轉換成前序和後序表示法的步驟，以運算子優先順序來進行處理，如下表所示：

步驟	說明	前序表示法	後序表示法
1	轉換加法	A*(+BC)	A*(BC+)
2	轉換乘法	*A(+BC)	A(BC+)*
3	刪除括號	*A+BC	ABC+*

運算式的種類、計算與轉換-轉換(方法2)

- 另一種方法是先替中序運算式加上完整括號來確認運算的優先順序，如下所示：

  中序運算式：　　A+B*(C+D)-E

  加上括號的中序運算式：　((A+(B*(C+D)))-E)

- 上述是加上括號的中序運算式，現在只需從最中間的括號開始，將運算子移到右括號的位置且刪除右括號，直到刪除所有右括號為止，如下所示：

  將運算子搬移到右括號：　((A(B(CD+*+E-

  刪除所有的左括號：　　ABCD+*+E-

運算式的種類、計算與轉換-轉換(範例)

- 一些中序運算式轉換成前序和後序運算式的範例，如下表所示：

中序表示法	前序表示法	後序表示法
A+B	+AB	AB+
(A+B) /C	/+ABC	AB+C/
(A+B) * (C+D)	*+AB+CD	AB+CD+*

3.6.2 後序運算式的計算-說明

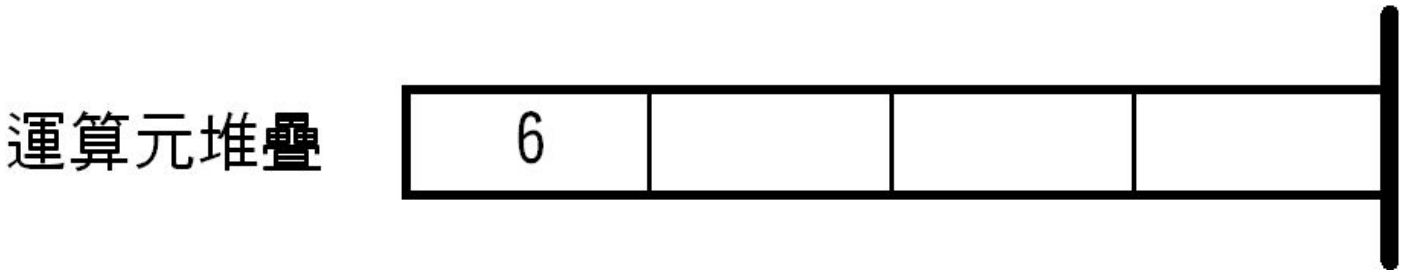
- 後序運算式的計算和前序運算式類似，都屬於無括號和優先順序的運算式計算，例如：一個後序運算式如下所示：

  67*45++

- 上述運算式的運算子是位在運算元之後，換句話說，當讀取運算式的運算元或運算子時，只需一個堆疊存放運算元即可，如果是運算子馬上取出堆疊的運算元，然後將計算結果存回到堆疊。

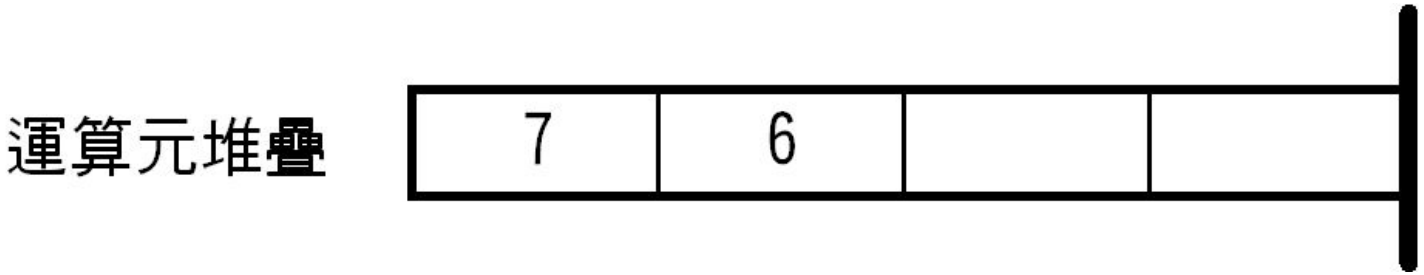
後序運算式的計算-過程1

- 後序運算式：67*45++的計算過程是在主迴圈依序讀取運算式的運算元或運算子，第1個讀入的字元是'6'，因為是運算元，所以存入運算元堆疊，如下圖所示：



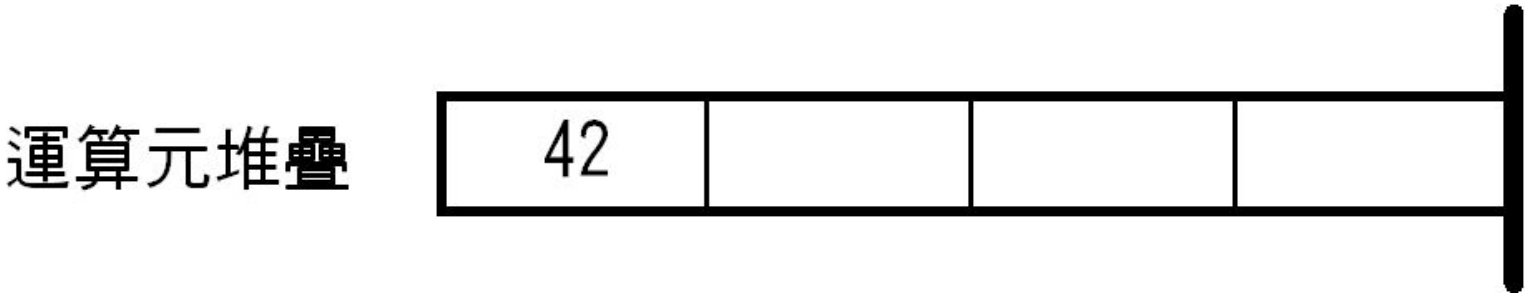
後序運算式的計算-過程2

- 接著讀入的字元'7'是運算元，再存入運算元堆疊，如下圖所示：



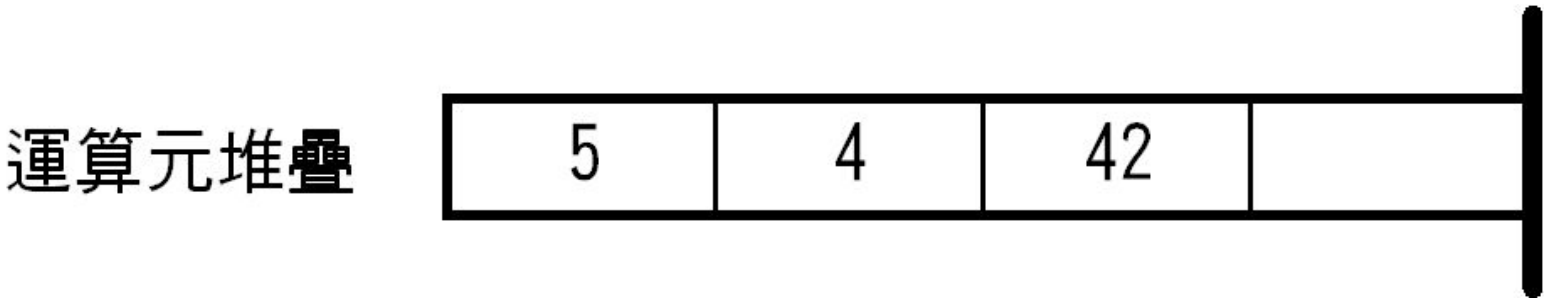
後序運算式的計算-過程3

- 繼續讀入字元'*'是運算子，從運算元堆疊取出2個運算元7和6，將6*7的計算結果42存回堆疊，如下圖所示：



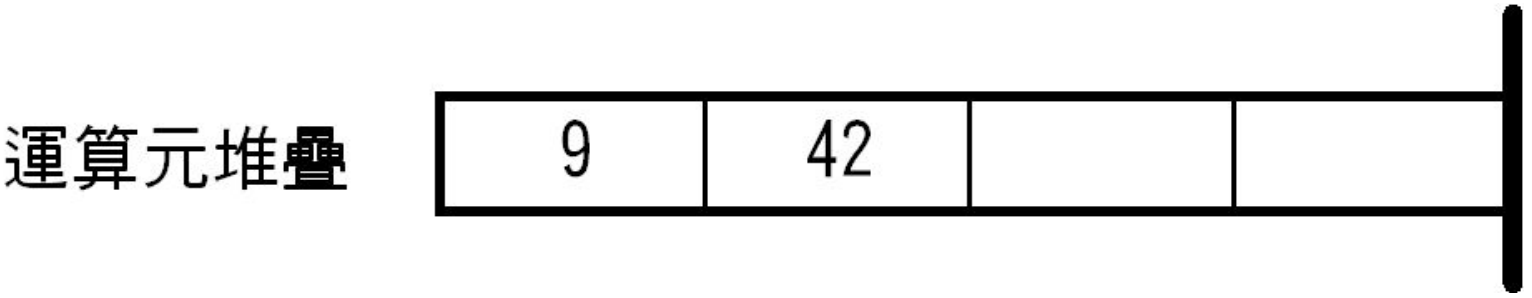
後序運算式的計算-過程4

- 接著讀入的是字元'4'和字元'5'兩個運算元，依序存入運算元堆疊，如下圖所示：



後序運算式的計算-過程5

- 繼續讀入的字元'+'是運算子，所以從運算元堆疊取出2個運算元5和4，然後將4+5計算結果9存回運算元堆疊，如下圖所示：



後序運算式的計算-過程6

- 最後一個讀入的字元是運算子'+'，從運算元堆疊取出2個運算元9和42，就可以得到後序運算式的計算結果是51。

後序運算式的計算-演算法

- 後序運算式計算的演算法可以依照前述計算過程來推導，其完整步驟如下所示：
  - Step 1：使用迴圈從左至右依序讀入後序運算式。
    - (1) 如果讀入的是運算元，直接存入運算元堆疊。
    - (2) 如果讀入的是運算子, 則：
      - 1) 從運算元堆疊取出所需的運算元。
      - 2) 計算此運算元和運算子的值後，存回運算元堆疊。
  - Step 2：最後取出運算元堆疊的內容，就是後序運算式的計算結果。

Evaluating postfix expression:如何求postfix expression之值 -演算法

(1) 自左而右scan expression.

(2) 將operands 放入stack中，直到碰到一個operator.

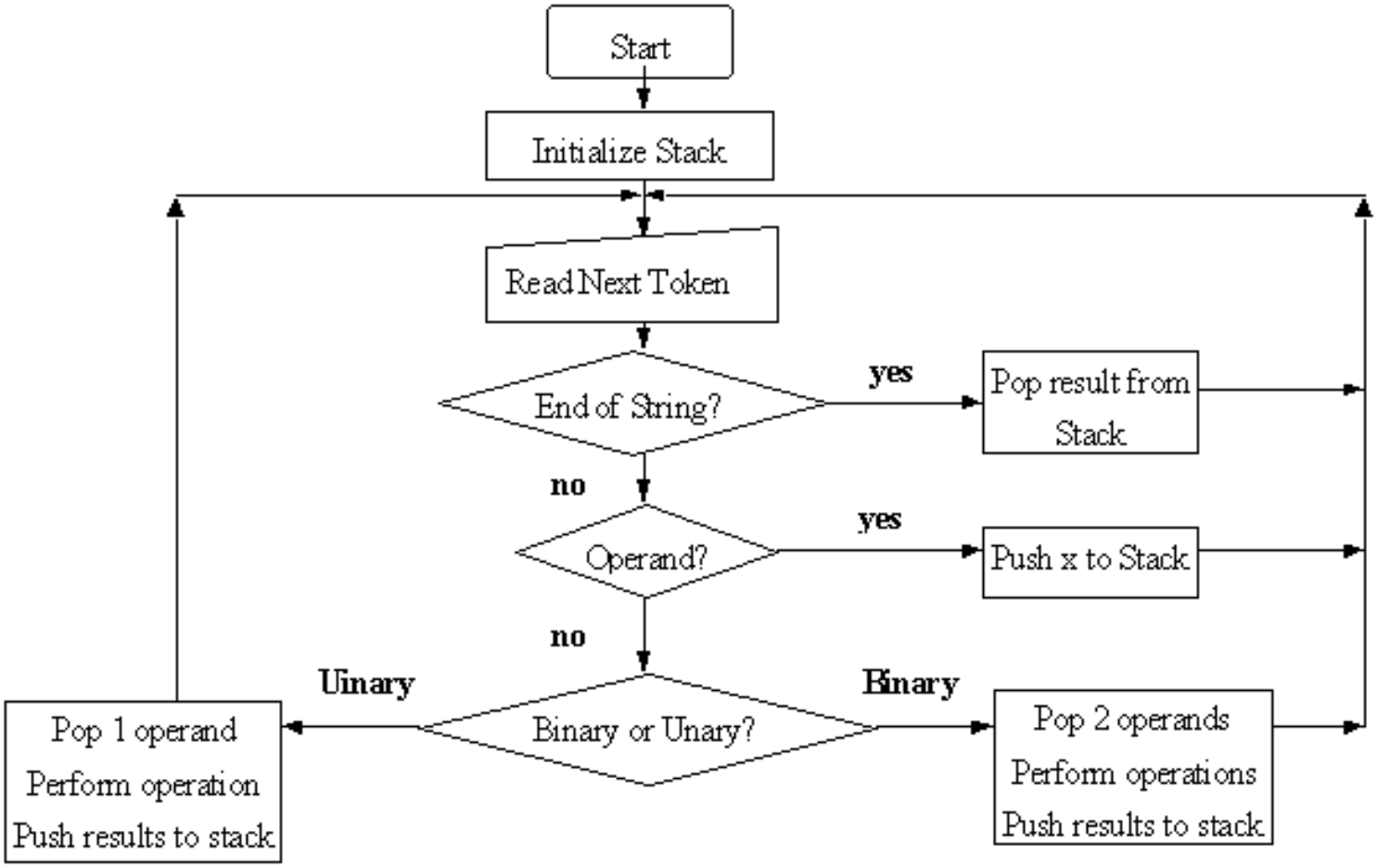
(3) 針對此operator,自stack中拿出正確數目的operands.

(4) 執行此operator

(5) 將結果放回 stack

(6) 重複1~5，直到scan完expression.

後序運算式之流程圖


Start
Initialize Stack
Read Next Token
End of String?
yes
Pop result from Stack
no
Operand?
yes
Push x to Stack
no
Uinary
Binary
Binary or Unary?
Pop 1 operand
Perform operation
Push results to stack
Pop 2 operands
Perform operations
Push results to stack

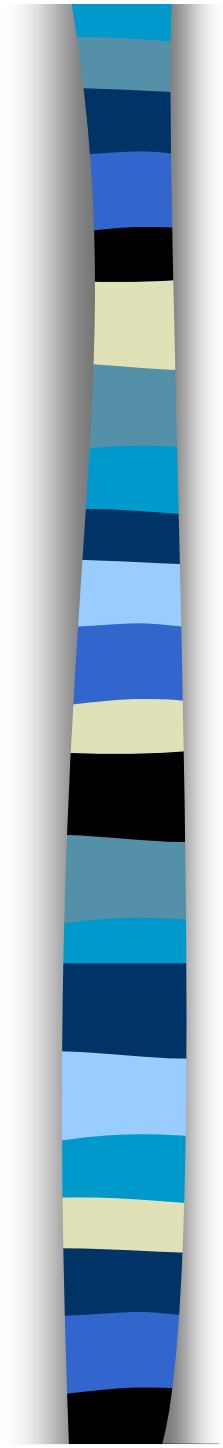
Token	Stack [0]	[1]	[2]	Top
6	6			0
2	6	2		1
/	6/2			0
3	6/2	3		1
-	6/2-3			0
4	6/2-3	4		1
2	6/2-3	4	2	2
*	6/2-3	4*2		1
+	6/2-3+4*2			0

Fig3.14 Postfix evaluation 62 / 3 – 42 * +)p.131)

```
int eval(void)
{
/* evaluate a postfix expression, expr, maintained as a
   global variable, '\0' is the the end of the expression.
   The stack and top of the stack are global variables.
   get_token is used to return the token type and
   the character symbol. Operands are assumed to be single
   character digits */
  precedence token;
  char symbol;
  int op1, op2;
  int n = 0;  /* counter for the expression string */
  int top = -1;
  token = get_token(&symbol, &n);
  while (token != eos)  {
     if (token == operand)          exp: character array
         add(&top, symbol-'0');   /* stack insert */
```

```
    else {
        /* remove two operands, perform operation, and
           return result to the stack */
      op2 = delete(&top);  /* stack delete */
      op1 = delete(&top);
      switch(token) {
          case plus: add(&top, op1+op2); break;
          case minus: add(&top, op1-op2); break;
          case times: add(&top, op1*op2); break;
          case divide: add(&top, op1/op2); break;
          case mod: add(&top, op1%op2);
      }
    }
    token = get_token (&symbol, &n);
  }
  return delete(&top); /* return result */
}
```

***Program 3.9:** Function to evaluate a postfix expression (p.122)



```
precedence get_token(char *symbol, int *n)
{
/* get the next token, symbol is the character
   representation, which is returned, the token is
   represented by its enumerated value, which
   is returned in the function name */

  *symbol =expr[(*n)++];
  switch (*symbol)  {
     case '(' : return lparen;
     case ')' : return rparen;
     case '+': return plus;
     case '-' : return minus;
```

```
    case '/' :  return divide;
    case '*' : return times;
    case '%' : return mod;
    case '\0' : return eos;
    default  : return operand;
               /* no error checking, default is operand */
    }
}
```

***Program 3.14:** Function to get a token from the input string (p.134)

3.6.3 中序運算式轉換成後序運算式-說明

- 中序運算式轉換成後序運算式可以使用堆疊配合運算子優先順序來進行轉換。例如：相同運算結果的中序和後序運算式，如下所示：

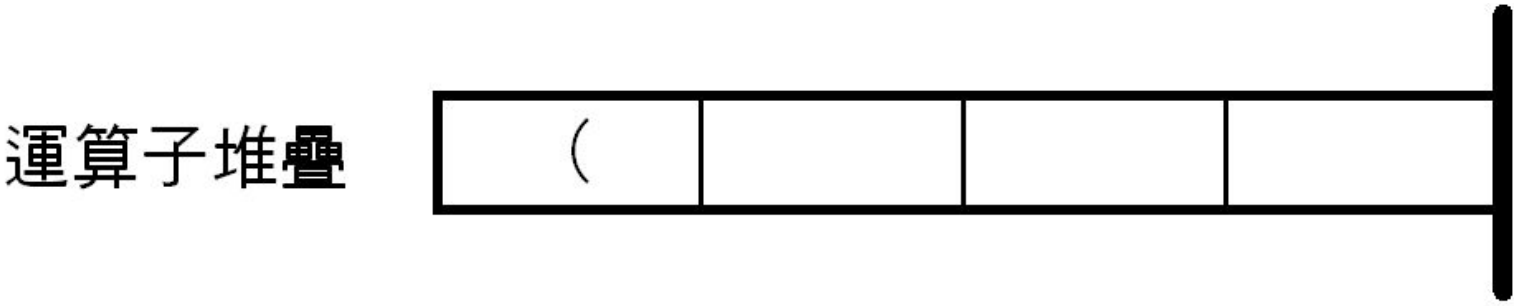
  中序運算式： (9+6)*4

  後序運算式： 96+4*

- 上述後序運算式是從中序運算式轉換而成，可以**看出兩個運算式中的運算元排列順序是相同的**，**只有運算子執行順序有優先順序的差異**，所以中序運算式的轉換**只需一個運算子堆疊**，如果讀入運算元馬上輸出即可，運算子需要進行優先順序的比較，以決定輸出或存入堆疊。

中序運算式轉換成後序運算式-過程1

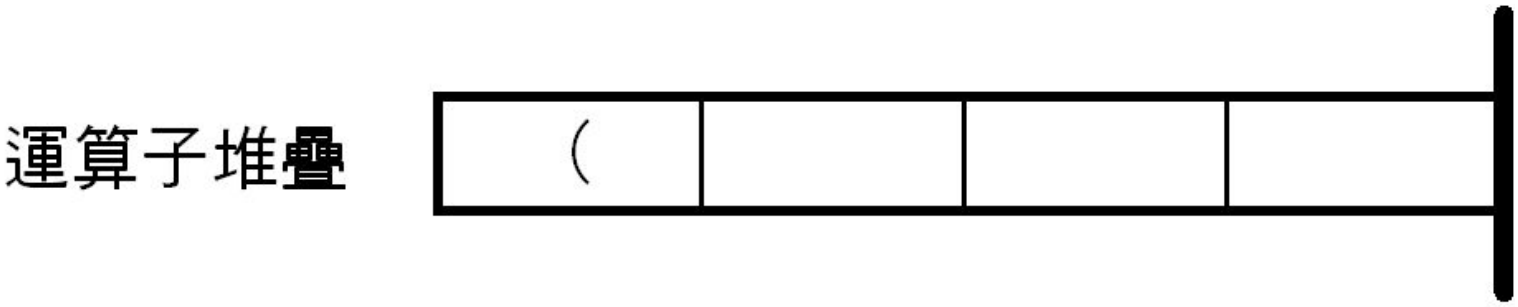
- 中序運算式：(9+6)*4的計算過程是在主迴圈從左至右依序讀取運算式的運算元或運算子，第1個讀入的字元是'('左括號，存入運算子堆疊，如下圖所示：



輸出：

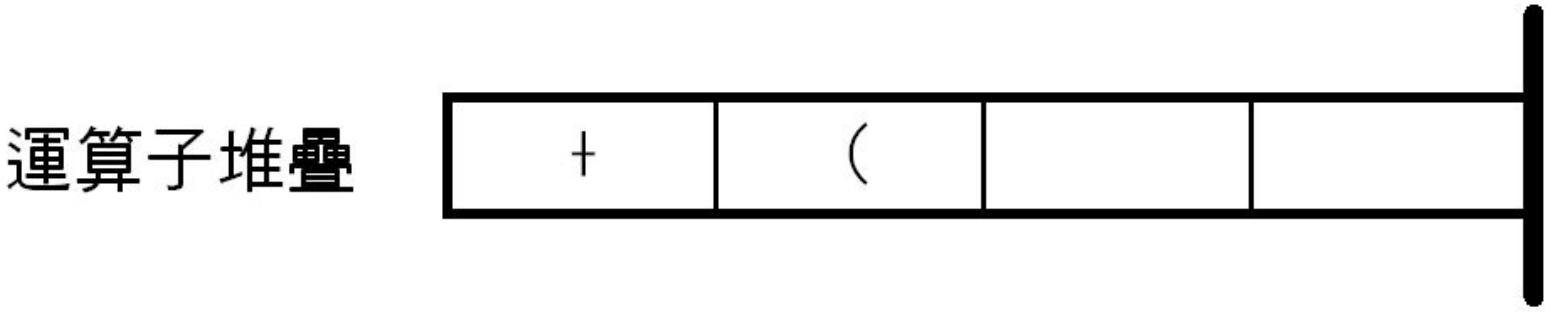
中序運算式轉換成後序運算式-過程2

- 接著讀入的字元是運算元'9'，直接輸出運算元，如下圖所示：



中序運算式轉換成後序運算式-過程3

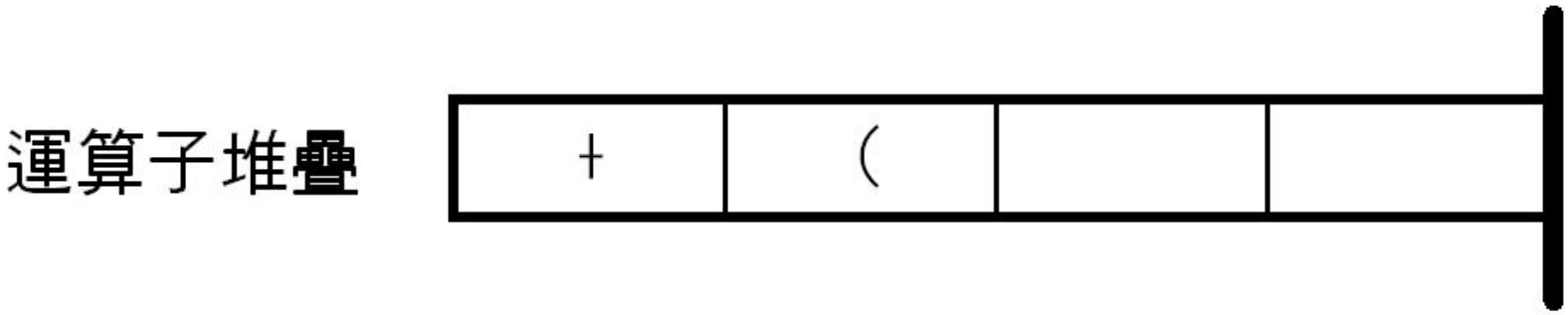
- 繼續讀入字元'+'是運算子，因為不是右括號且'+'號的優先順序大於堆疊中的'('號（注意！括號的優先順序和第5-3-1節不同，左括號的優先順序最小），所以將運算子存入運算子堆疊，如下圖所示：



輸出：　9

中序運算式轉換成後序運算式-過程4

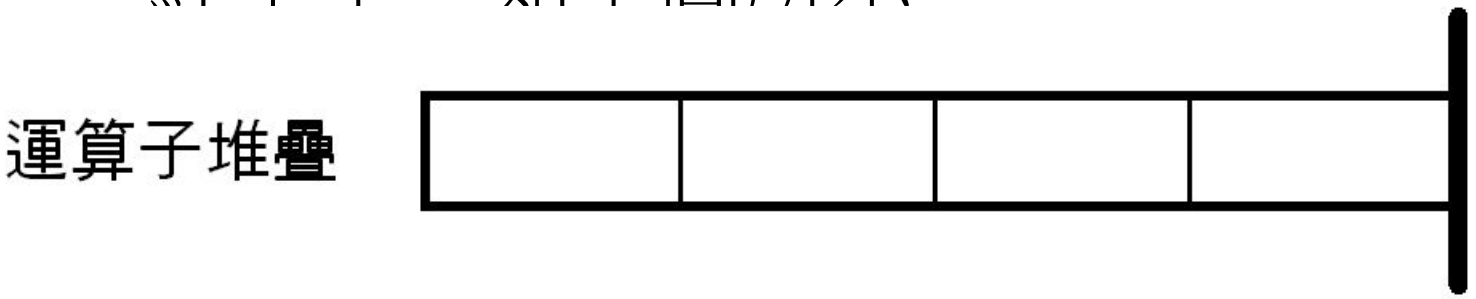
- 然後讀入的字元是運算元'6'，直接輸出運算元，如下圖所示：



輸出：　　96

中序運算式轉換成後序運算式-過程5

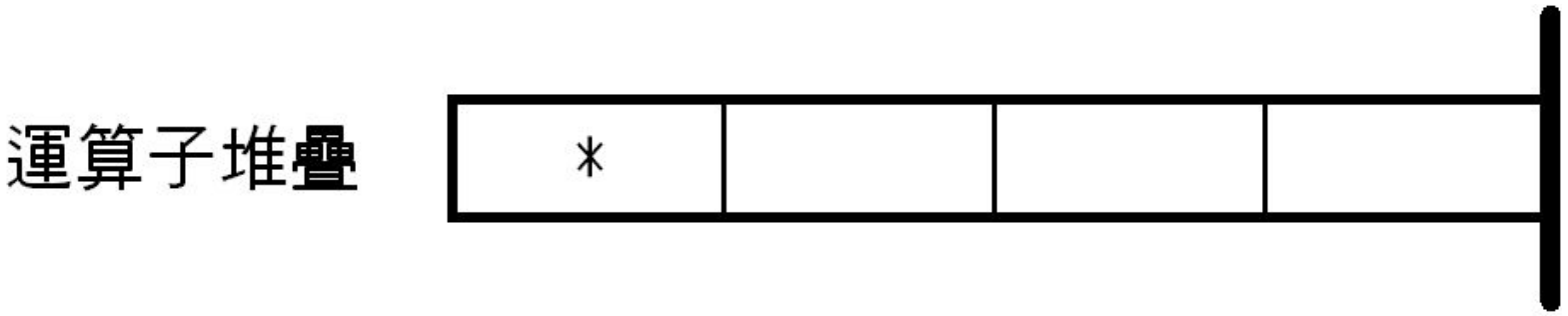
- 接著讀入的是右括號')'，我們需要從運算子堆疊取出運算子'+'輸出，直到左括號為止，左括號在此的功能只是作為一個標籤，所以並不用輸出，此時的堆疊已經全空，如下圖所示：



輸出：　　96+

中序運算式轉換成後序運算式-過程6

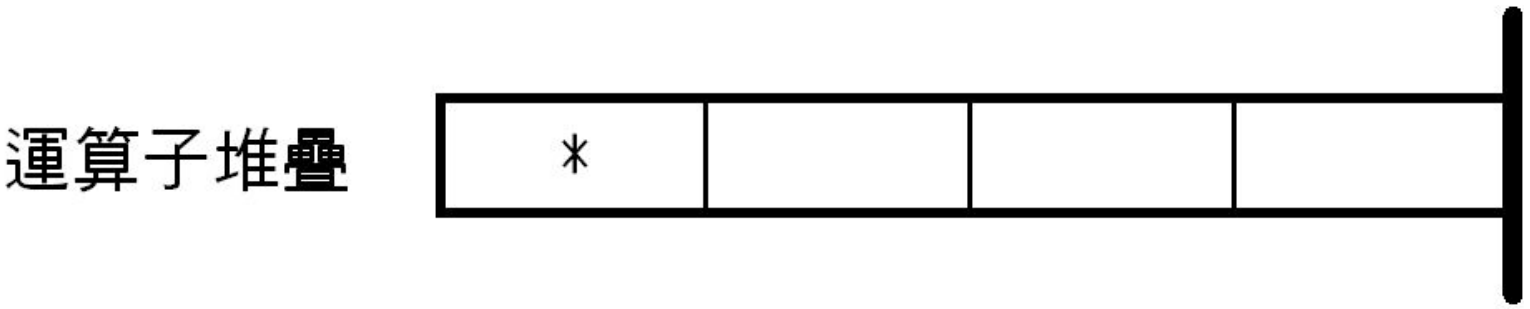
- 繼續讀入字元'*'是運算子，存入運算子堆疊，如下圖所示：



輸出：　　96+

中序運算式轉換成後序運算式-過程7

- 接著讀入運算元'4'，直接輸出運算元，如下圖所示：



輸出：　　96+4

中序運算式轉換成後序運算式-過程8

- 現在已經讀完整個中序運算式，接著從運算子堆疊取出剩下的運算子'*'且輸出，可以得到轉換的後序運算式，如下所示：

  96+4*

中序運算式轉換成後序運算式-演算法

- 中序轉成後序運算式的演算法可以依照前述計算過程來推導，其完整步驟如下所示：
  - Step 1：使用迴圈讀取中序運算式的運算元和運算子，若：
    - (1) 讀取的是運算元，直接輸出運算元。
    - (2) 讀取的是運算子：
      - 1) 如果運算子堆疊是空的或是左括號，存入運算子堆疊。
      - 2) 如果是右括號，從堆疊取出運算子輸出，直到左括號為止。
      - 3) 如果堆疊不是空的，持續和堆疊的運算子比較優先順序，若欲加入之運算子其優先順序比：
        - » a.堆疊中的運算子低，則將運算子存入運算子堆疊。
        - » b.堆疊中的運算子高或相同，則輸出堆疊中的運算子。
      - 4)如果堆疊空了，則將運算子存入運算子堆疊。
  - Step 2：如果運算子堆疊不是空的，依序取出運算子堆疊的運算子。

中序運算式轉換成後序運算式-演算法

(1) The order of operands is the same in infix and postfix.

(2) During scanning, operands are outputted

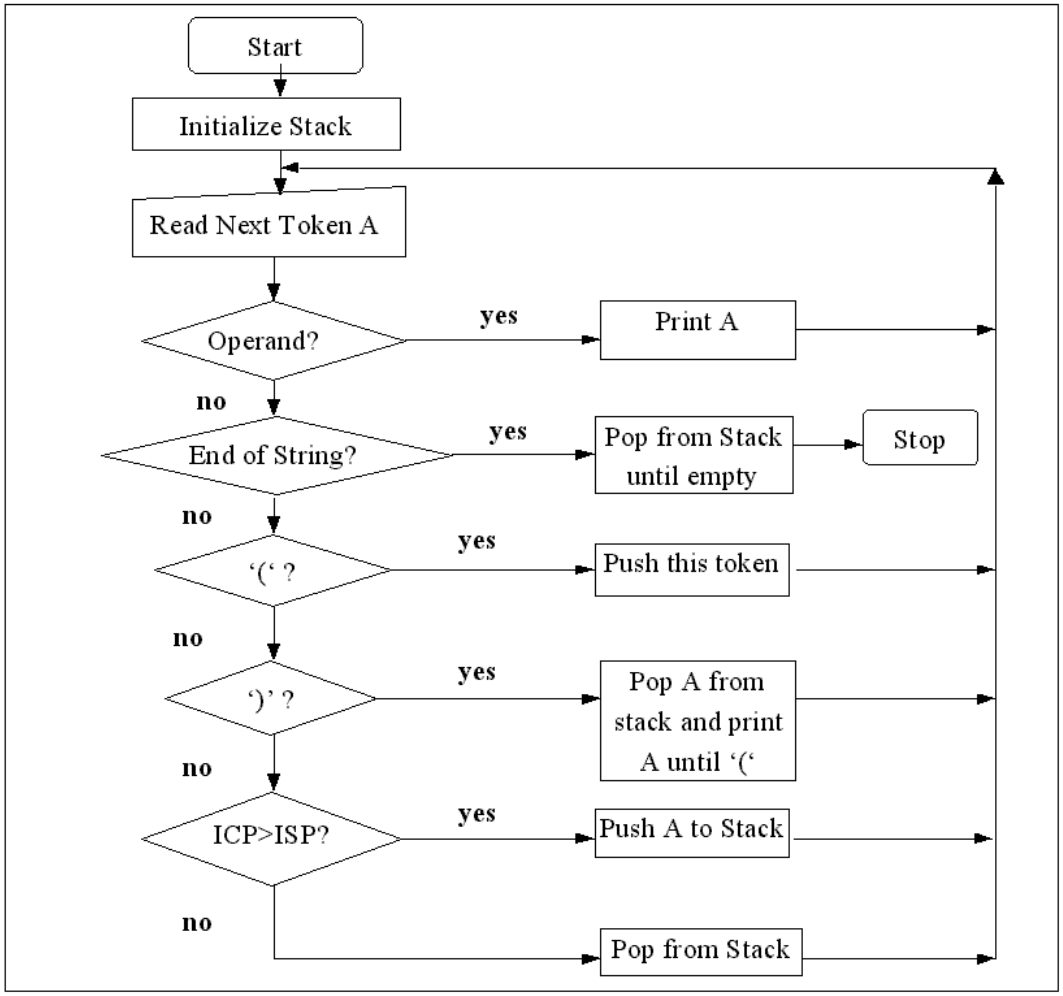
(3) The order in which the operators are output depends on their precedence.

(4) 使用stack 存放operator ：

若欲push 之operator 其precedence

大於operators of top of stack 則push此operator: 否則一直pop operators 直到stack 中之operator 之precedence小於，欲push之operator’s;再push此operator;

中序運算式轉換為後序運算式之流程圖



Infix to Postfix Conversion
(Intuitive Algorithm)

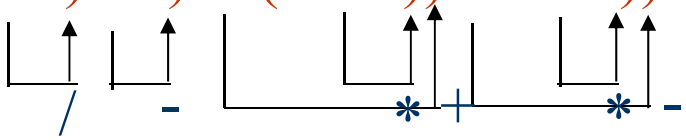
(1) Fully parenthesize expression

a / b - c + d * e - a * c -->

((((a / b) - c) + (d * e)) - a * c))

(2) All operators replace their corresponding right parentheses.

((((a / b) - c) + (d * e)) - a * c))



(3) Delete all parentheses.

ab/c-de*+ac*-

two passes

The orders of operands in infix and postfix are the same.

a + b * c, * > +

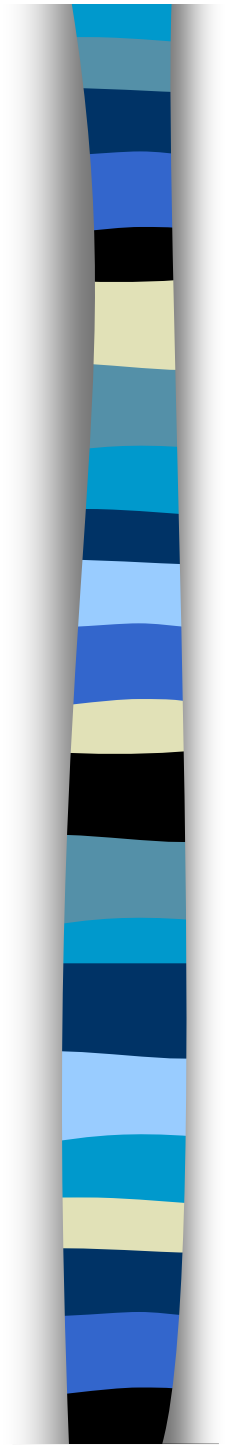
Token	Stack [0]	[1]	[2]	Top	Output
a				-1	a
+	+			0	a
b	+			0	ab
*	+	*		1	ab
c	+	*		1	abc
eos				-1	abc*+

***Figure 3.15:** Translation of a+b*c to postfix (p.135)

$a *_1 (b + c) *_2 d$

Token	Stack [0]	[1]	[2]	Top	Output
a				-1	a
$*_1$	$*_1$			0	a
(	$*_1$	(		1	a
b	$*_1$	(		1	ab
+	$*_1$	(	+	2	ab
c	$*_1$	(	+	2	abc
)	$*_1$	match)		0	abc+
$*_2$	$*_2$	$*_1 = *_2$		0	abc+$*_1$
d	$*_2$			0	abc+$*_1$d
eos	$*_2$			0	abc+$*_1$d$*_2$

* **Figure 3.16:** Translation of a*(b+c)*d to postfix (p.135)

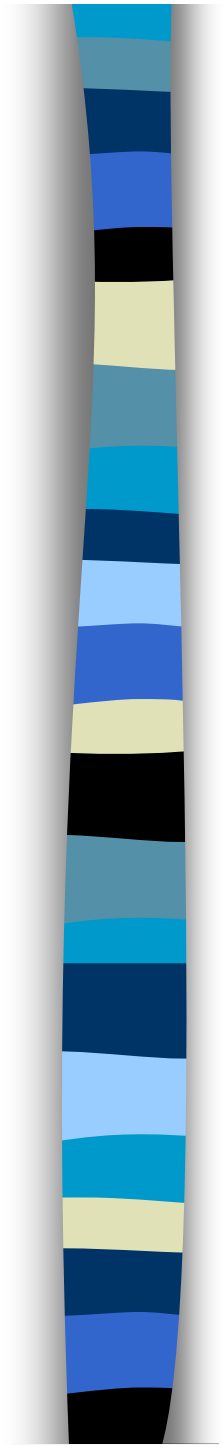


Rules

(1) Operators are taken out of the stack as long as their in-stack precedence is higher than or equal to the incoming precedence of the new operator.

(2) (has low in-stack precedence, and high incoming precedence.

	(	)	+	-	*	/	%	eos
isp	0	19	12	12	13	13	13	0
icp	20	19	12	12	13	13	13	0



```
precedence stack[MAX_STACK_SIZE];
/* isp and icp arrays -- index is value of precedence
lparen, rparen, plus, minus, times, divide, mod, eos */
static int isp [ ] = {0, 19, 12, 12, 13, 13, 13, 0};
static int icp [ ] = {20, 19, 12, 12, 13, 13, 13, 0};
```

isp: in-stack precedence

icp: incoming precedence

```
void postfix(void)
{
/* output the postfix of the expression. The expression
    string, the stack, and top are global */
   char symbol;
   precedence token;
   int n = 0;
   int top = 0; /* place eos on stack */
   stack[0] = eos;
   for (token = get _token(&symbol, &n); token != eos;
                    token = get_token(&symbol, &n)) {
     if (token == operand)
        printf (“%c”, symbol);
     else if (token == rparen ){
```

```
    /*unstack tokens until left parenthesis */
    while (stack[top] != lparen)
        print_token(delete(&top));
    delete(&top); /*discard the left parenthesis */
   }
   else{
    /* remove and print symbols whose isp is greater
        than or equal to the current token's icp */
    while(isp[stack[top]] >= icp[token] )
        print_token(delete(&top));
    add(&top, token);
   }
 }
 while ((token = delete(&top)) != eos)
     print_token(token);
 print("\n");
}
```

$\theta(n)$

f(n)=θ(g(n)) iff there exist positive constants c_1, c_2, and n_0 such that $c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n, n \ge n_0$.

f(n)=θ(g(n)) iff g(n) is both an upper and lower bound on f(n).

***Program 3.15**: Function to convert from infix to postfix (p.137)

Infix	Prefix
a*b/c a/b-c+d*e-a*c a*(b+c)/d-g	/*abc -+-/abc*de*ac -/*a+bcdg

(1) evaluation
(2) transformation

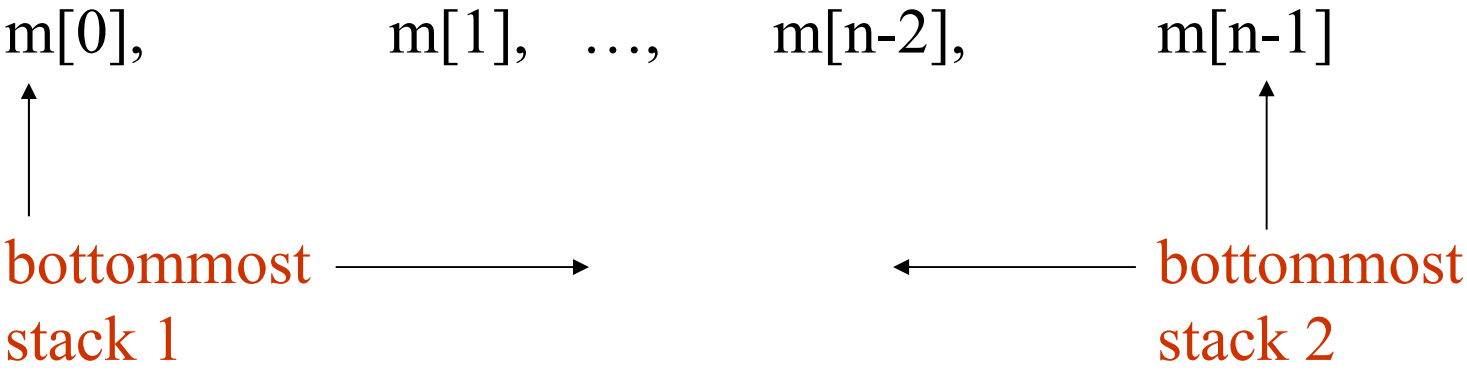
***Figure 3.17:** Infix and postfix expressions (p.138)

作業

- P136: ex1, ex6(a)

Multiple stacks and queues

Two stacks:由陣列memory[MEMORY_SIZE]來表示2個Stack



More than two stacks (n)

memory is divided into n equal segments

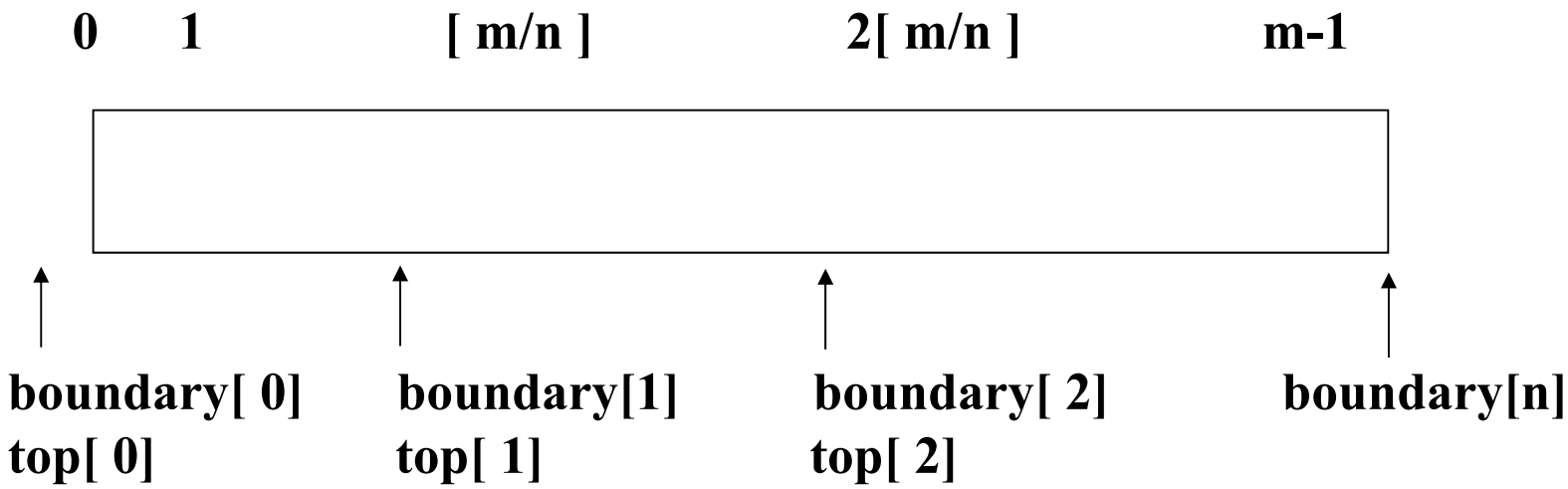
boundary[stack_no]

$0 \le stack\_no < MAX\_STACKS$

top[stack_no]

$0 \le stack\_no < MAX\_STACKS$

Initially, boundary[i]=top[i].



All stacks are empty and divided into roughly equal segments.

***Figure 3.18:** Initial configuration for *n* stacks in memory [m]. (p.140)

```
#define MEMORY_SIZE 100    /* size of memory */
#define MAX_STACK_SIZE 100
            /* max number of stacks plus 1 */
/* global memory declaration */
element memory[MEMORY_SIZE];
int top[MAX_STACKS];
int boundary[MAX_STACKS];
int n; /* number of stacks entered by the user */
```
*(p.139)

```
top[0] = boundary[0] = -1;
for (i = 1; i < n; i++)
  top[i] =boundary[i] =(MEMORY_SIZE/n)*i;
boundary[n] = MEMORY_SIZE-1;
```
*(p.139)

```
void push(int i, element item)
{
   /* add an item to the ith stack */
   if (top[i] == boundary [i+1])
      stack_full(i);     may have unused storage
      memory[++top[i]] = item;
}
```

***Program 3.16:** Add an item to the stack *stack-no* (p.140)

```
element pop(int i)
{
   /* remove top element from the ith stack */
   if (top[i] == boundary[i])
     return stack_empty(i);
   return memory[top[i]--];
}
```

***Program 3.17:** Delete an *item* from the stack *stack-no* (p.140)

Find j, stack_no < j < n (往右)

such that top[j] < boundary[j+1]

or, 0 ≤ j < stack_no (往左)

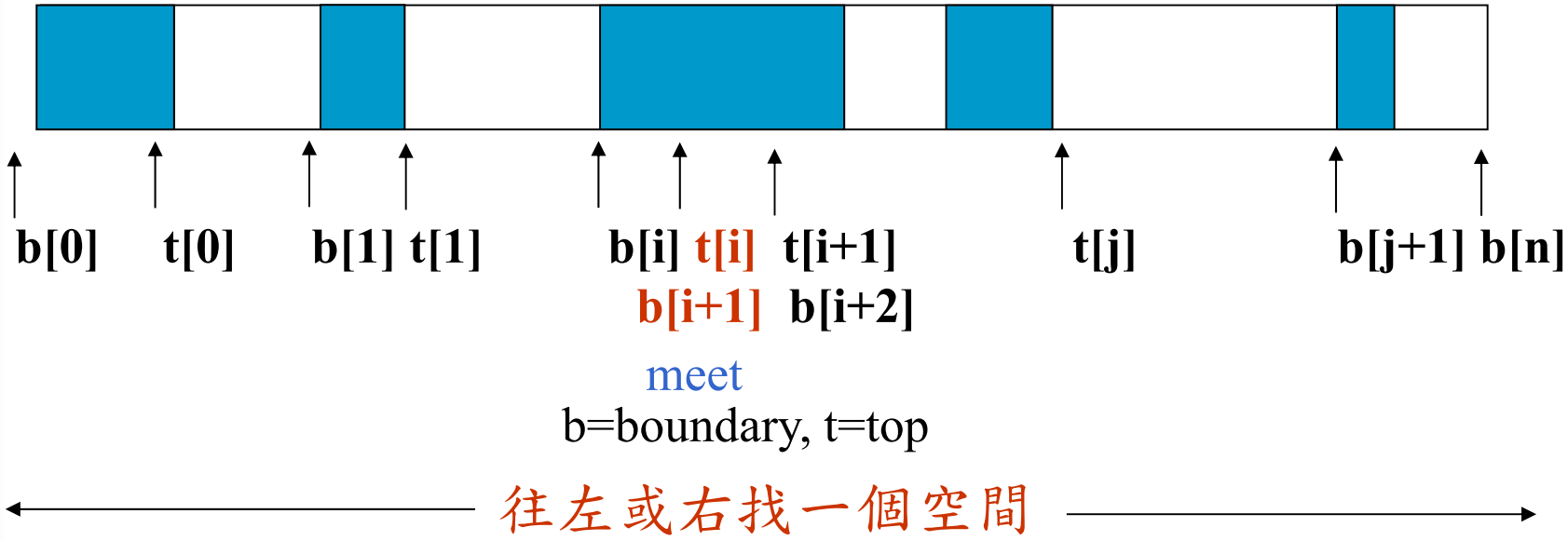


***Figure 3.19:** Configuration when stack i meets stack i+1, but the memory is not full (p.141)

作業

- P136: ex1, ex6(a)