資料結構
DATA STRUCTURE

Chap.02 Stacks and Queues

Ming-Han Tsai

2020 Spring

陣列(Array)

- 陣列(array)由一連串的索引(index)和值(value)構成
- 對每個索引index，必有一個相關聯的值value

- 如果可能的話，盡量使用連續空間的記憶體
  - 減少存取時間

The stack(堆疊) ADT

- A stack is an ordered list in which insertions and deletions are made at one end called the top.
  - *插入和刪除只作用於最頂層的元素：top*
- If we add the elements *A*, *B*, *C*, *D*, *E* to the stack, in that order, then *E* is the first element we delete from the stack
- A stack is also known as a *Last-In-First-Out* (*LIFO*) list. => 後進先出

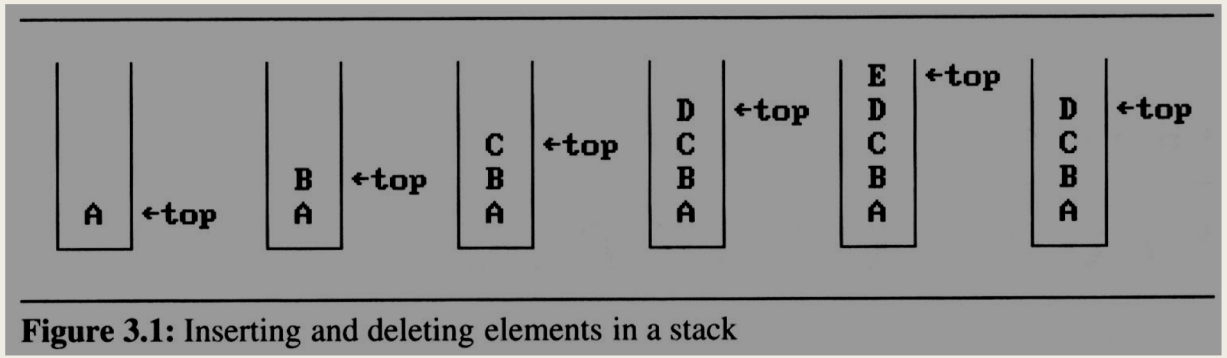
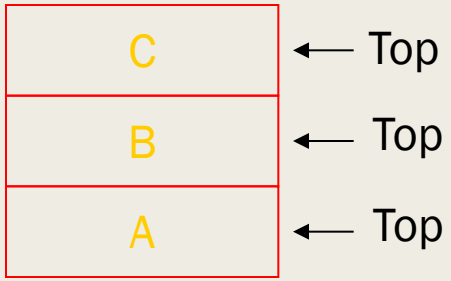


Figure 3.1: Inserting and deleting elements in a stack



The stack ADT

記憶體中的函數呼叫方式

(activation record)

fp: a pointer to current stack frame

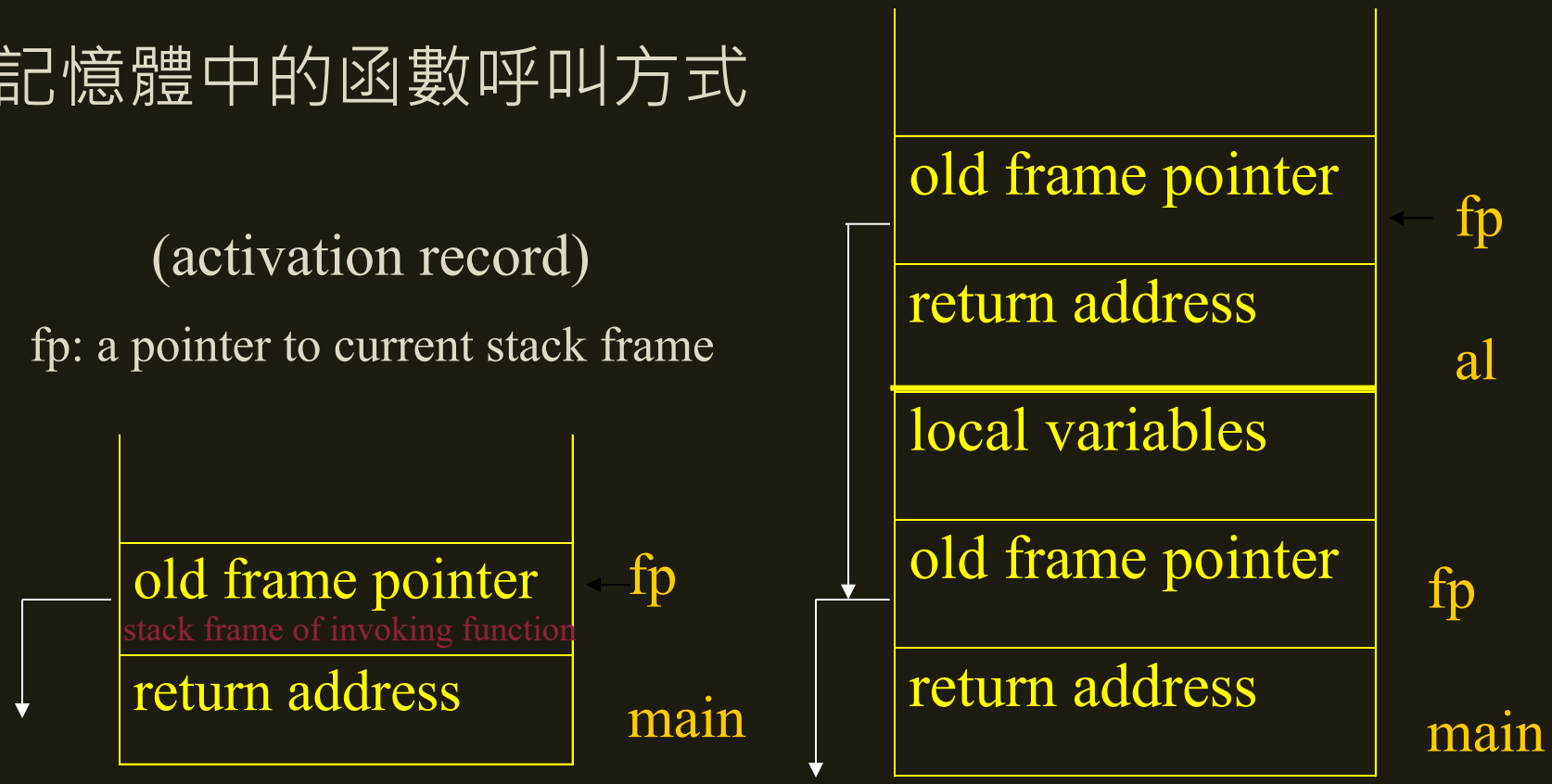


***Figure 3.2:** System stack after function call a1 (p.103)

Recall：遞迴(recursive)

The stack ADT

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*

The stack ADT

- **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;
```

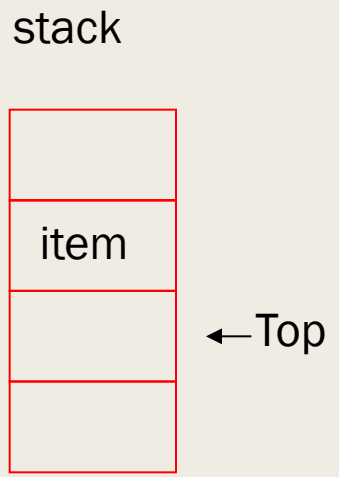
Boolean IsEmpty(Stack) ::= `top < 0;`

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

The stack ADT

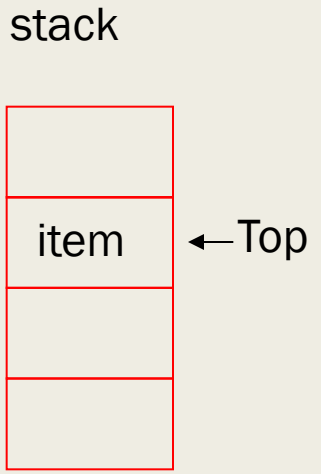
```
void add(int *top, element item)
{
/* add an item to the global stack */
   if (*top >= MAX_STACK_SIZE-1) {
      stack_full();
      return;
   }
   stack[++*top] = item;
}
```

Program 3.1: Add to a stack



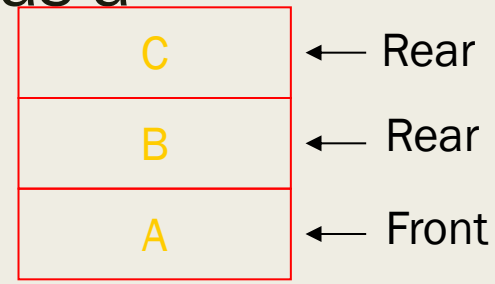

```
element delete(int *top)
{
/* return the top element from the stack */
   if (*top == -1)
      return stack_empty(); /* returns an error key */
   return stack[(*top)--];
}
```

Program 3.2: Delete from a stack



The queue(佇列) ADT

- A queue is an ordered list in which all insertion take place one end, called the rear and all deletions take place at the opposite end called the front
- *First-In-First-Out* (*FIFO*) list
  - *先進先出；最後放進queue的元素要等前面都取光之後才能取出來*
- If we insert the elements *A*, *B*, *C*, *D*, *E*, in that order, then *A* is the first element we delete from the queue as a



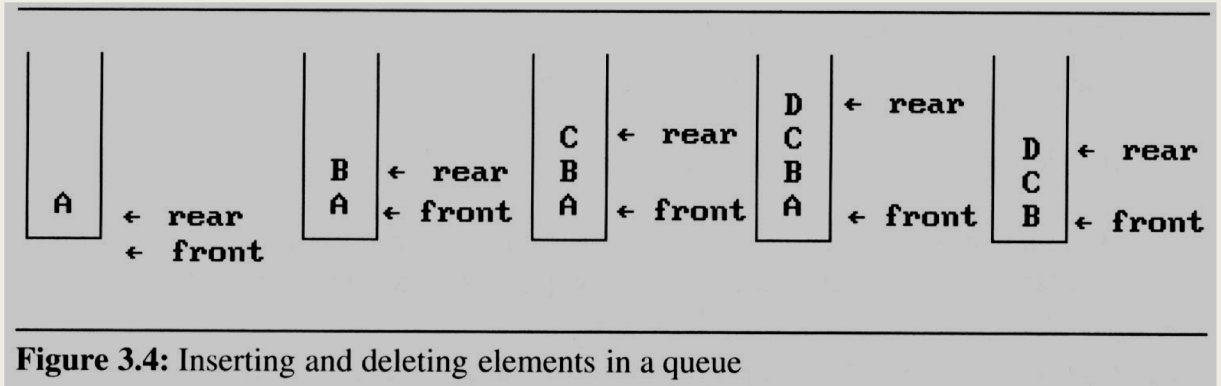


Figure 3.4: Inserting and deleting elements in a queue

The queue ADT

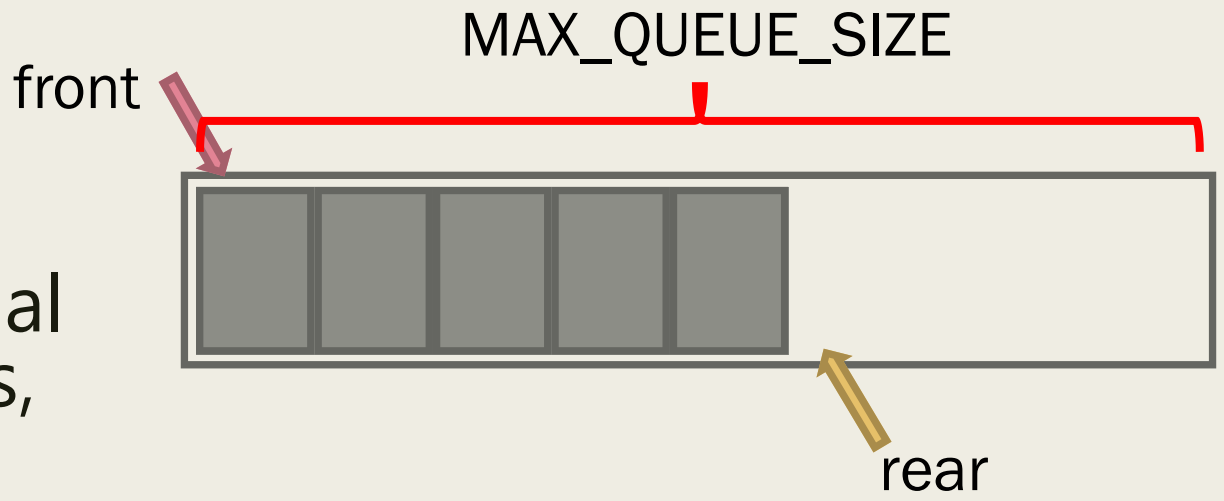
---

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*

The queue ADT

- **Implementation 1:** 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 IsEmptyQ(queue) ::= front == rear
Boolean IsFullQ(queue) ::= rear == MAX_QUEUE_SIZE-1
```

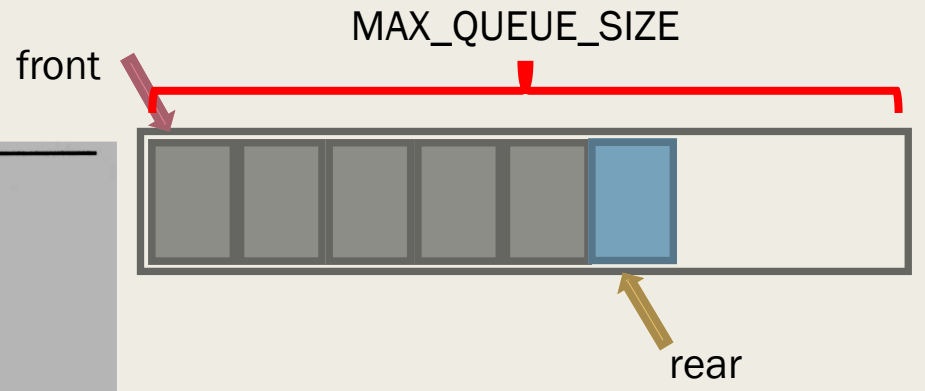
The queue ADT

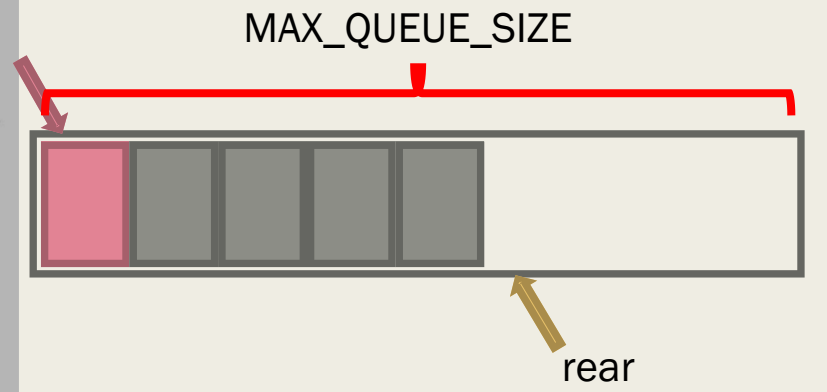
```
void addq(int *rear, element item)
{
/* add an item to the queue */
   if (*rear == MAX_QUEUE_SIZE-1) {
      queue_full();
      return;
   }
   queue[++*rear] = item;
}
```

Program 3.3: Add to a queue

```
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





problem: there may be available space when IsFullQ is true i.e. movement is required.

範例：工作排程Job scheduling

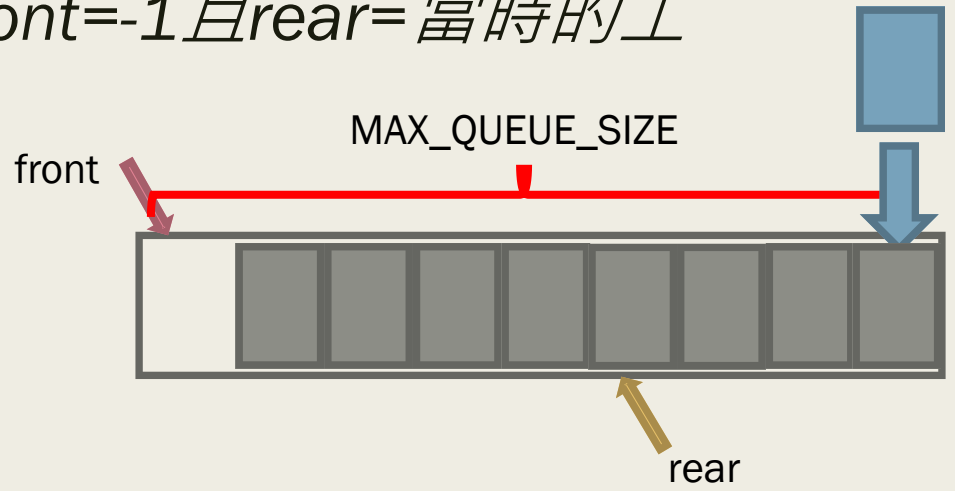
- 利用Queue來實作作業系統中的工作排程(循序)

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

  - *當工作(job)進入/離開系統時，整個queue會往右移動*
  - *當queue_full時，整個queue必須由右移動(搬運)至最左邊，使第一個元素對齊到queue[0]，front=-1且rear=當時的工作數*

    - Shifting an array is very time-consuming, queue_full has a worst case complexity of O(MAX_QUEUE_SIZE).



- We can obtain a more efficient representation if we regard the array *queue*[*MAX_QUEUE_SIZE*] as circular.

Implementation 2:

circular queue

front: one position counterclockwise from the first element

rear: current end

Problem: one space is left when queue is full.

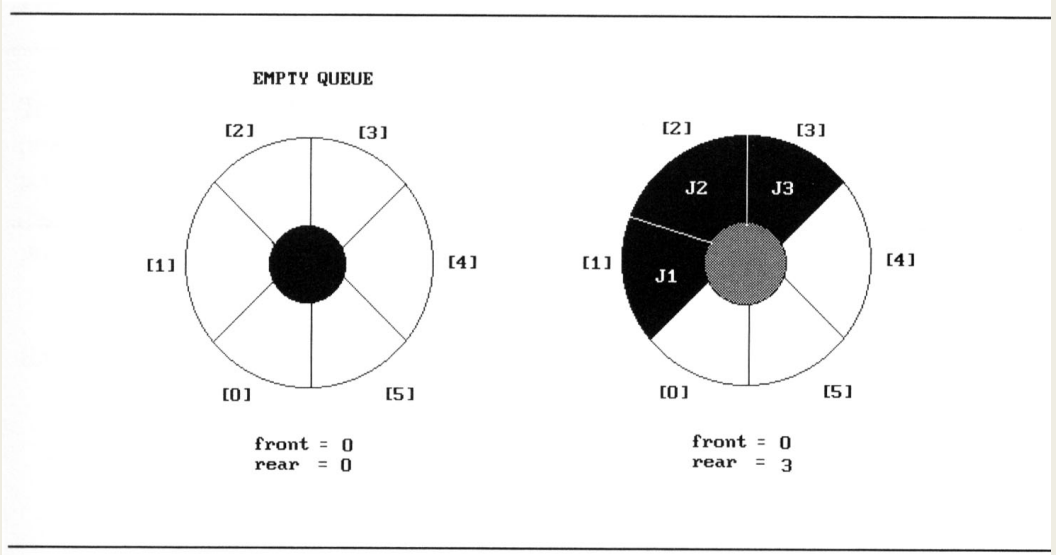


Figure 3.6: Empty and nonempty circular queues

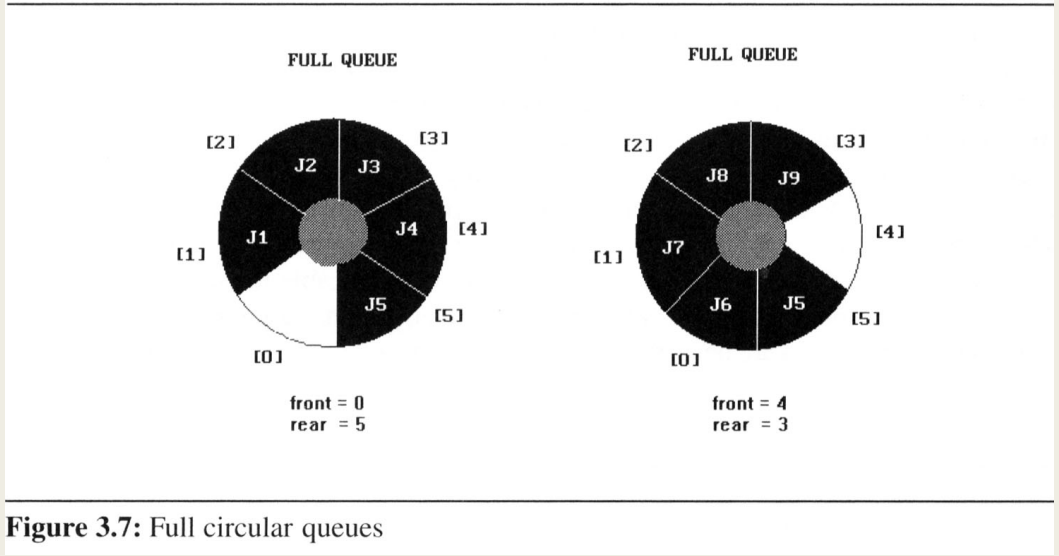


Figure 3.7: Full circular queues

- Implementing *addq* and *deleteq* for a circular queue is slightly more difficult since 需要考慮繞過一圈的情形

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

Program 3.5: Add to 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

走迷宮問題

1: 不能通過　0: 可以通過

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

方向的表示方式

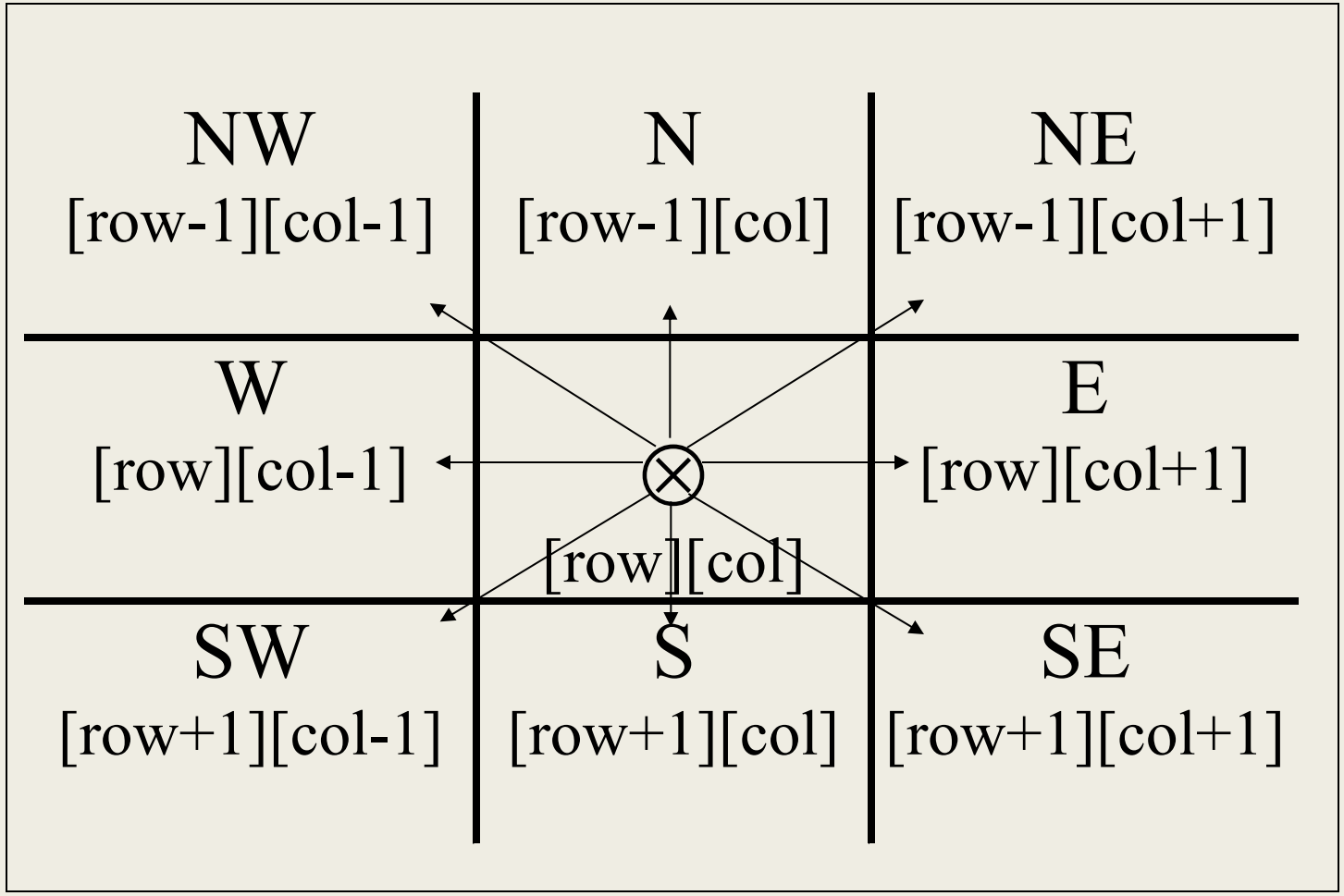


*Figure 3.9: Allowable moves (p.113)

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

利用Stack來記錄走過的路徑

```
#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) {
```

取出步伐

可以走&沒有走過

```
    /* 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");
```

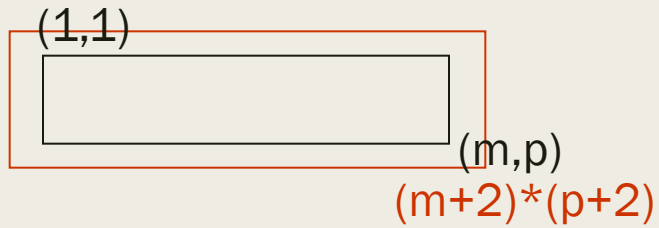
***Program 3.7:** Initial maze algorithm (p.115)

The size of a stack?

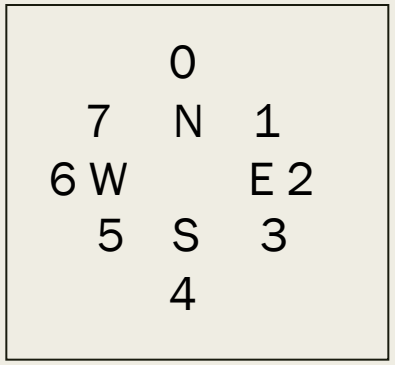
$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}_{m*p}$$

mp --> $\lceil m/2 \rceil$ p, mp --> $\lceil p/2 \rceil$ m

*Figure 3.11: Simple maze with a long path (p.116)




```
void path (void)
{
/* 輸出能走出迷宮的(其中一條)正確路徑(如果存在) */
   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) {
            /*往dir的方向移動*/
            next_row = row + move[dir].vert;
            next_col  = col + move[dir].horiz;
```



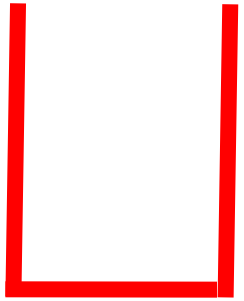
```
        if (next_row==EXIT_ROW && next_col==EXIT_COL)
            found = TRUE;
        else if ( !maze[next_row][next_col] && !mark[next_row][next_col] {
            /*          不是牆壁          而且          沒有走過*/
            mark[next_row][next_col] = 1;
            position.row = row;
            position.col = col;
            position.dir = ++dir;
            add(&top, position);
            row = next_row; col = next_col; dir = 0;
         }
         else ++dir;
    }
}
```

放入目前位置 + 下一步要走的方向

```
    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 3.8:** Maze search function (p.117)

```
while (top > -1 && !found) {
    position = pop();
    row = position.row;
    col = position.col;
    dir = position.dir;
    while (dir < 8 && !found) {
        next_row = row + moves[dir].vert;
        next_col = col + moves[dir].horiz;
        if (next_row == EXIT_ROW && next_col == EXIT_COL)
            found = 1;
        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;
            push(position);
            row = next_row;
            col = next_col;
            dir = 0;
        }
        else ++dir;
    }
}
```

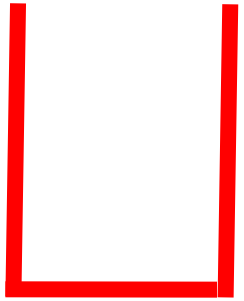


Maze

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

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

```
while (top > -1 && !found) {
    position = pop();
    row = position.row;
    col = position.col;
    dir = position.dir;
    while (dir < 8 && !found) {
        next_row = row + moves[dir].vert;
        next_col = col + moves[dir].horiz;
        if (next_row == EXIT_ROW && next_col == EXIT_COL)
            found = 1;
        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;
            push(position);
            row = next_row;
            col = next_col;
            dir = 0;
        }
        else ++dir;
    }
}
```

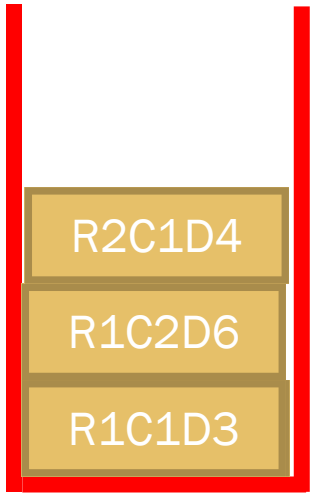


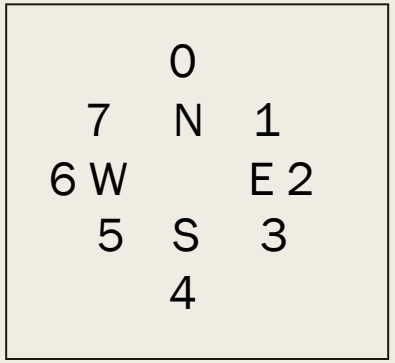
Maze

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

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

```
while (top > -1 && !found) {
    position = pop();
    row = position.row;
    col = position.col;
    dir = position.dir;
    while (dir < 8 && !found) {
        next_row = row + moves[dir].vert;
        next_col = col + moves[dir].horiz;
        if (next_row == EXIT_ROW && next_col == EXIT_COL)
            found = 1;
        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;
            push(position);
            row = next_row;
            col = next_col;
            dir = 0;
        }
        else ++dir;
    }
}
```





Mark

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

Maze

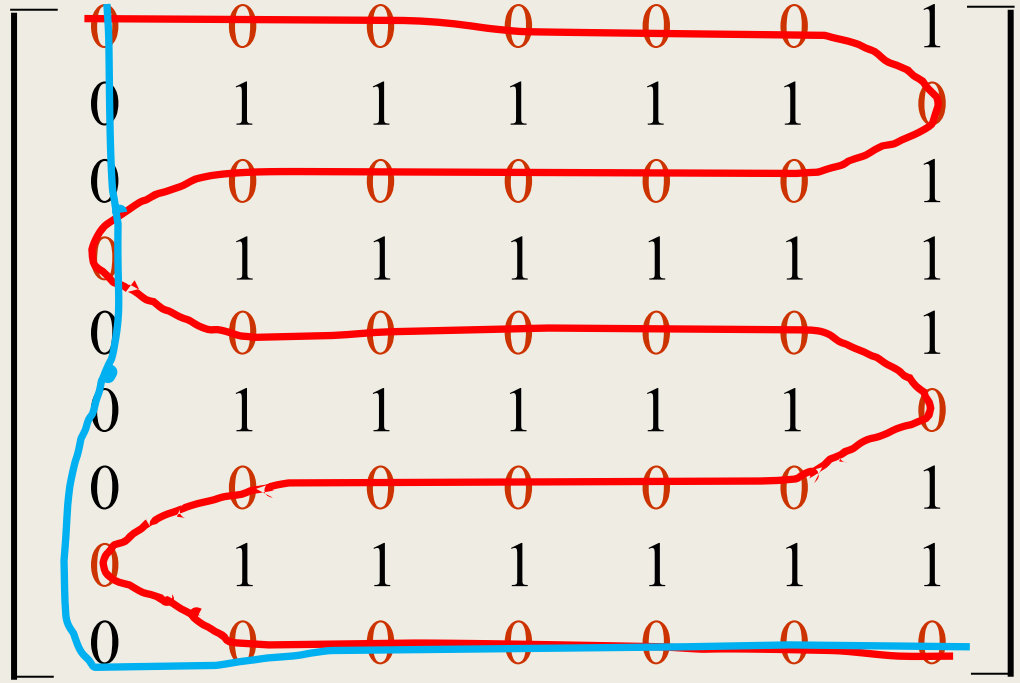
1	1	1	1	1
1	0	0	0	1
1	0	1	1	1
1	0	0	0	1
1	1	1	1	1

最多需要走幾步？

O(m/2*p) 或者 O(m*p/2)

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

m*p



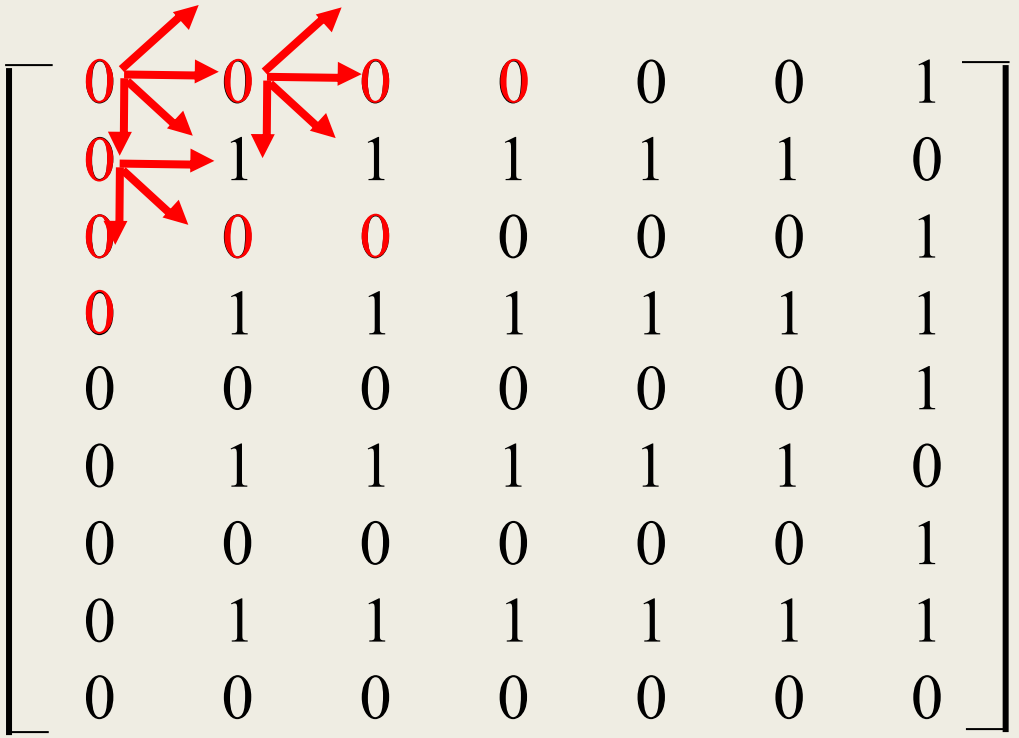
Length = 31

Length = 15

Stack應用在走迷宮

- 總之先往前走，往前走到沒路走再回頭
  - 深度優先(Depth-First)
  - 走錯路的時候也會走到底才回頭

- 換個思考方式：
  - 往每個方向都先走一步
  - 看走到終點了沒？
  - 沒有再往每個方向都走一步
  - 再看走到終點了沒？
  - …



用Queue來達到上面的效果

- 從Queue取出第一個位子，將從該位子可以一步之內走到的位子都丟進Queue
  - 深度優先(Depth-First)
  - 距離原點K步的點都走完之後，才會踏入距離K+1步的點

- 想法很美好，但實際上
  - 遍歷所有距離一步的位子、兩步的位子、...有點不切實際
  - 有沒有折衷的辦法？

- A* Algorithm 採用Priority Queue

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

數學表示式Expression

- ((*rear*+1==*front*) || ((*rear*==*MAX_QUEUE_SIZE-1*) && !*front*))
- *x* = *a*/*b* - *c*+*d***e* - *a***c*
- 包含：
  - *運算子* operators: ==, +, -, ||, &&, !
  - *運算元* operands: *rear, front, MAX_QUEUE_SIZE*
  - *括號* parentheses: ( )

Evaluation of Expressions

- x = a/b - c+d*e - a*c
- assume **a = 4, b = c = 2, d = e = 3,** finding out the value of x
  - Interpretation 1: ((4/2)-2)+(3*3)-(4*2) = 0+8+9 = 1
  - Interpretation 2: (4/(2-2+3))*(3-4)*2 = (4/3)*(-1)*2 = -2.66666…
  - *實際上我們利用括號(parentheses)改變計算的先後順序:*
    - $x = ((a/(b - c+d))*(e - a)*c$
- 如何將表示式轉換成一連串的機器運算？
  - *運算子的先後順序 precedence rule*
  - *結合律 associative rule*

- Precedence hierarchy and associative for C

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
* / %	multiplicative	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

Evaluation of Expressions

- Infix(中置式) and postfix(後置式)
  - *最標準的表示式寫法多為中置式 infix notation*
    - 二元運算子會放在兩個運算元之中，如：2+3
  - *由於運算先後和括號的問題，中置式並不是編譯器真正用來計算表示式的方法*
  - *實際上編譯器常用的是後置式(postfix)*

Postfix:
沒有括號
不須考慮優先序

Infix	Postfix
2+3*4	2 3 4*+
*a*b +5*	*ab*5+*
(1+2)*7	1 2+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

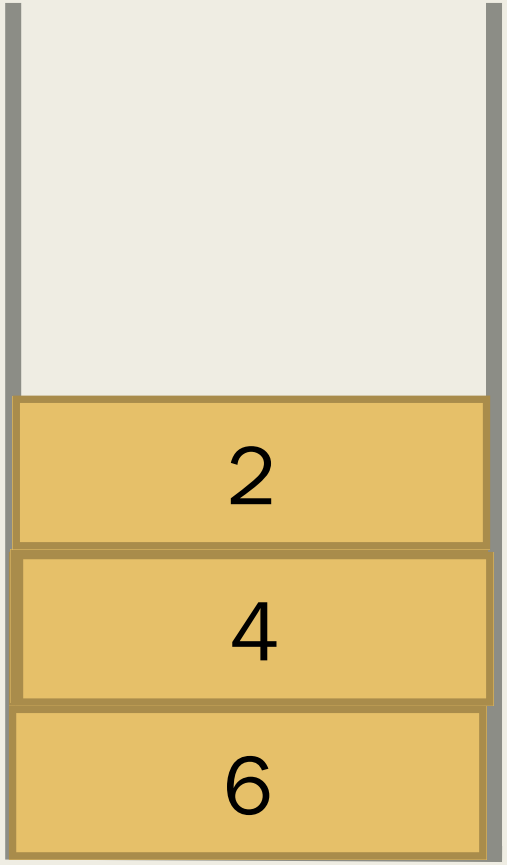
Evaluation of Expressions

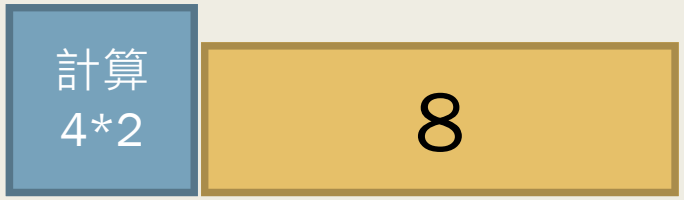
- 計算後置式的表示式比中置式簡單:
  - *不須考慮括號*
  - *只需要由左至右掃描過一遍*
  - *利用Stack可以簡單計算*

Figure 3.14 shows this processing when the input is nine character string 6 2/3-4 2*+

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

Figure 3.14: Postfix evaluation





6 2 / 3 - 4 2 * +

讀到運算子(符號)：從Stack取出兩個數做運算

3.4 Evaluation of Expressions (6/14)

- Representation
  - *We now consider the representation of both the stack and the expression*

```
#define MAX_STACK_SIZE 100 /*maximum stack size*/
#define MAX_EXPR_SIZE 100 /*max size of expression*/
typedef enum {lparen ,rparen, plus, minus, times, divide,
                        mod, eos, operand} precedence;
int stack[MAX_STACK_SIZE]; /* global stack */
char expr[MAX_EXPR_SIZE]; /* input string */
```

3.4 Evaluation of Expressions (7/14)

- Get Token

```
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 ' ' : return eos;
      default  : return operand; /* no error checking,
                                    default is operand */
   }
}
```

Program 3.10: Function to get a token from the input string

3.4 (8/14)

- Evaluation of Postfix Expression

```
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 tokentype 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)
         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 */
}
```

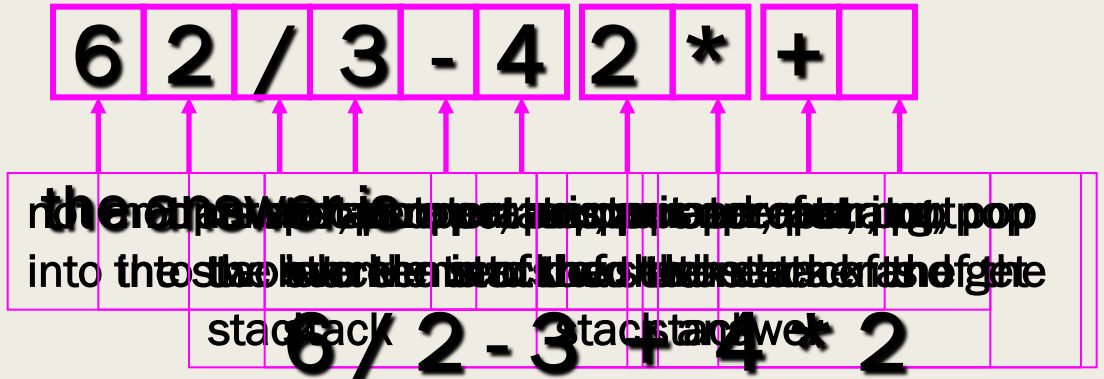
Program 3.9: Function to evaluate a postfix expression

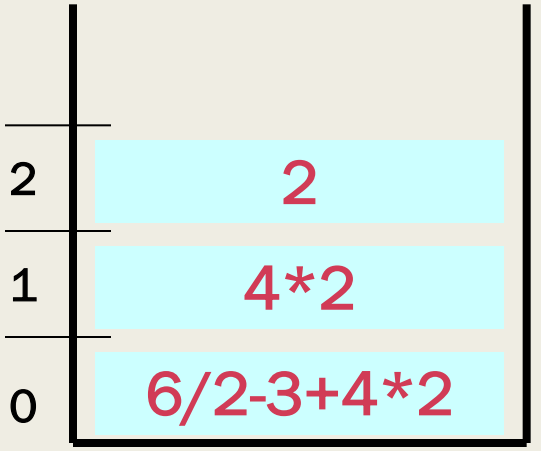
3.4 Evaluation of Expressions (9/14)

string: 6 2/3-4 2*+

we make a single left-to-right scan of it

add the string with the operator





now, top must -1

3.4 Evaluation of Expressions (10/14)

We can describe an algorithm for producing a postfix expression from an infix one as follows

EX: Trans a / b - c + d * e - a * c To postfix

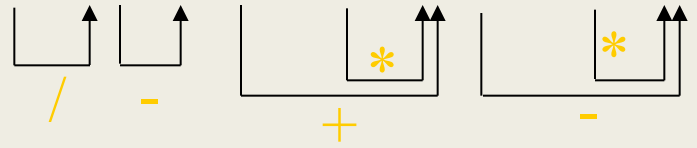
(1) Fully parenthesize expression

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))



two passes

(3)Delete all parentheses

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

➔ The order of operands is the same in infix and postfix

3.4 Evaluation of Expressions (11/14)

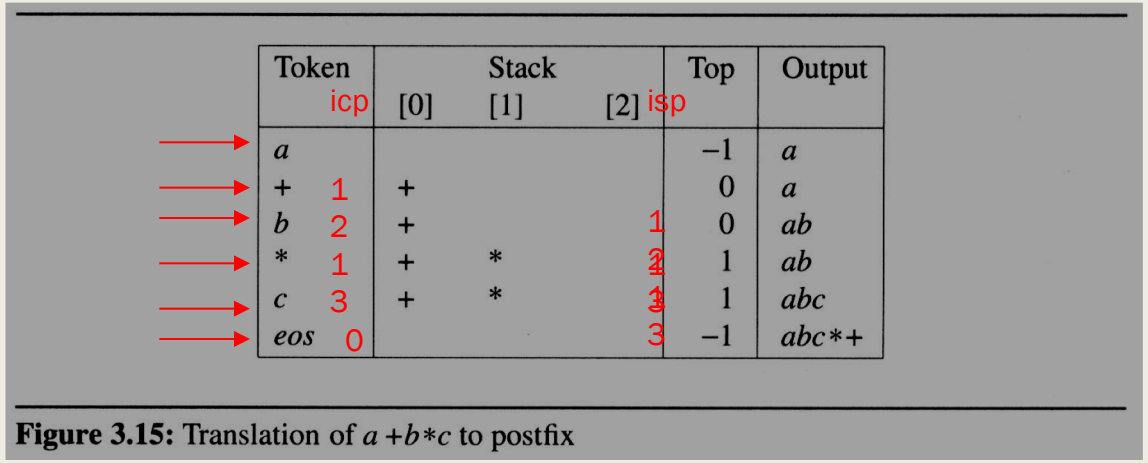
- Algorithm to convert from infix to postfix
  - *Assumptions:*
    - operators: (,), +, -, *, /, %
    - operands: single digit integer or variable of one character
  1. *Scan string from left to right*
  2. *Operands are taken out immediately*
  3. *Operators are taken out of the stack as long as their in-stack precedence (isp) is higher than or equal to the incoming precedence (icp) of the new operator*
  4. *'(' has low isp, and high icp*

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

3.4 Evaluation of Expressions (12/14)

- **Example 3.3 [*Simple expression*]:** Simple expression a+b*c, which yields abc*+ in postfix.

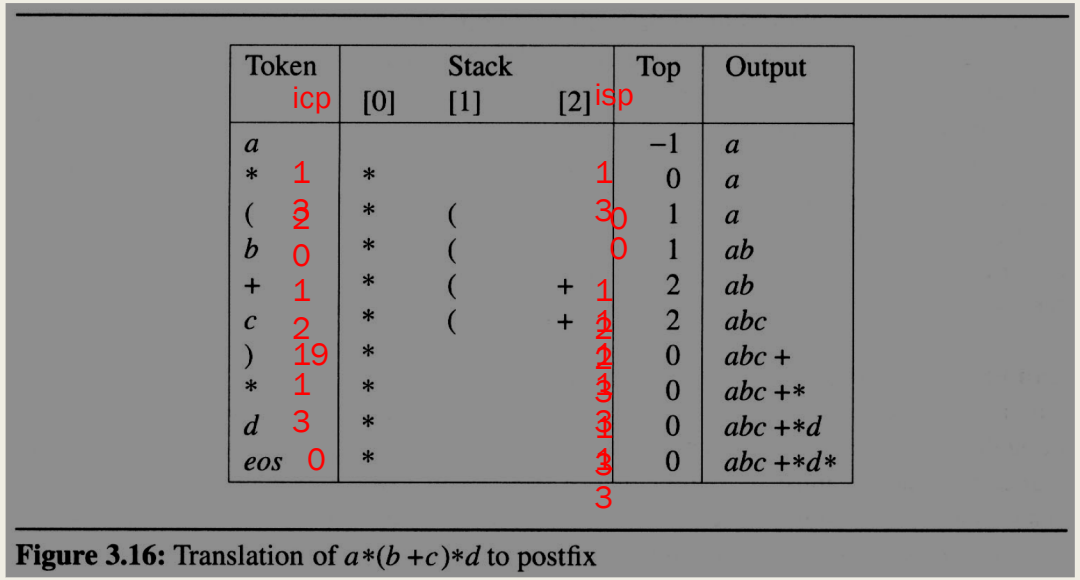
EX: a+b*c



Token	icp	Stack [0]	[1]	[2]	isp	Top	Output
a						−1	*a*
+	1	+				0	*a*
b	2	+			1	0	*ab*
*	1	+	*		2	1	*ab*
c	3	+	*		1	1	*abc*
eos	0				3	−1	*abc**+

Figure 3.15: Translation of *a* +*b***c* to postfix

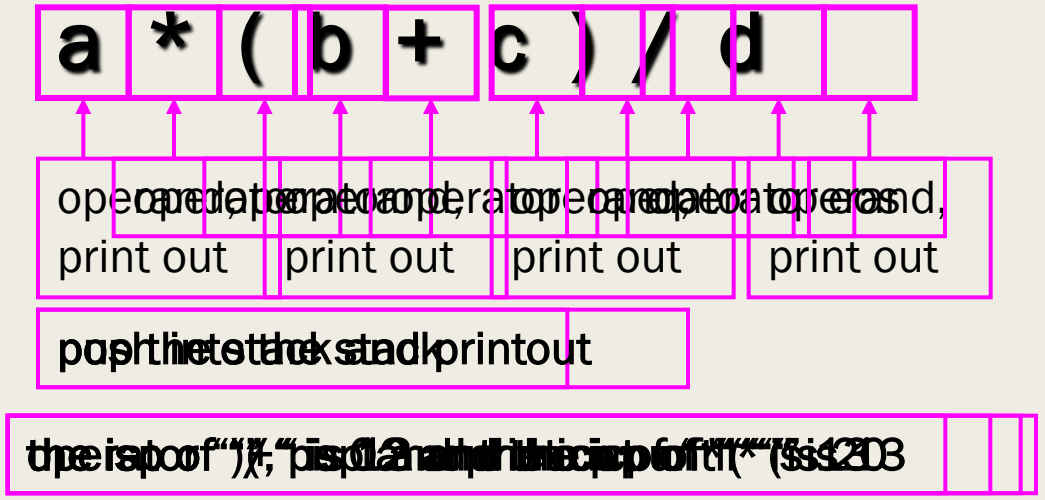
- **Example 3.5 [Parenthesized *expression*]:** The expression a*(b+c)*d, which yields abc+*d* in postfix



Token	icp	Stack [0]	[1]	[2]	isp	Top	Output
a						−1	*a*
*	1	*			1	0	*a*
(	3	*	(		3	1	*a*
b	0	*	(		0	1	*ab*
+	1	*	(	+	1	2	*ab*
c	2	*	(	+	1	2	*abc*
)	19	*			2	0	*abc* +
*	1	*			1	0	*abc* +*
d	3	*			1	0	*abc* +**d*
eos	0	*			1	0	*abc* +**d**

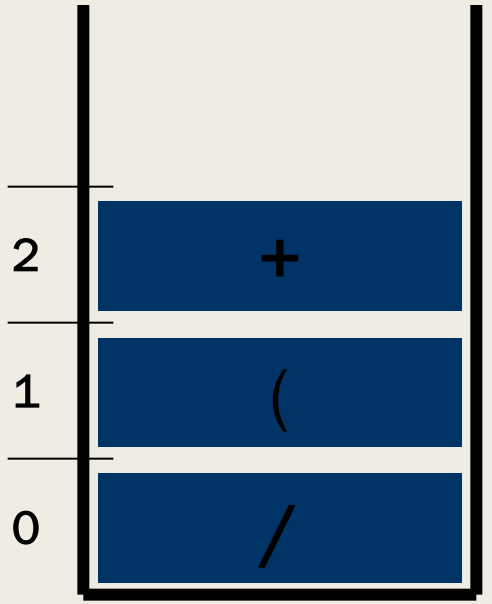
Figure 3.16: Translation of *a**(*b* +*c*)**d* to postfix

3.4 Evaluation of Expressions (13/14)



output

a b c + * d /



top → stack

now, top must +1

3.4 Evaluation of Expressions (14/14)

- Complexity: $\Theta(n)$
  - *The total time spent here is $\Theta(n)$ as the number of tokens that get stacked and unstacked is linear in n*
  - *where n is the number of tokens in the expression*

```
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);
   printf("\n");
}
```

Program 3.11: Function to convert from infix to postfix