CHAPTER 3

STACKS AND QUEUES

3.1 The stack abstract data type

– A *stack* is an ordered list in which insertions and deletions are made at one end called the *top*.

- Given a stack $S = (a_0, \ldots, a_{n-1})$, we say that a_0 is the bottom element, a_{n-1} is the top element, and a_i is on top of element a_{i-1}, $0 < i < n$.
- The restrictions on the stack imply that 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
  - Figure 3.1 illustrates this sequence of operations.
- Since the last element inserted into a stack is the first element removed, a stack is also known as a *Last-In-First-Out* (*LIFO*) list.

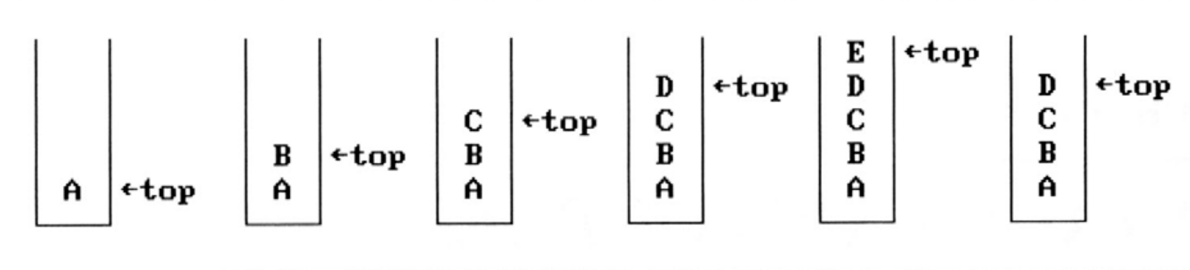


Figure 3.1: Inserting and deleting elements in a stack

– The ADT specification of the stack is shown in Structure 3.1

---

structure *Stack* is
  objects: a finite ordered list with zero or more elements.
  functions:
    for all $stack \in Stack$, $item \in element$, $max\_stack\_size \in$ 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 easiest way to implement this ADT is by using a one dimensional array, say stack[MAX_STACK_SIZE], where MAX_STACK_SIZE is the maximum number of entries.

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;`

```
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

3.2 The queue abstract data type

– A *queue* is an ordered list in which all insertions take place at one end and all deletions take place at the opposite end.
  - Given a queue $Q = (a_0, \ldots, a_{n-1})$, we say that a_0 is the front element, a_{n-1} is the rear element, and a_{i+1} is behind a_i, $0 \leq i < n-1$.
  - The restrictions on the queue imply that if we insert the elements *A*, *B*, *C*, *D*, *E*, in that order, then *A* is the first element we delete from the queue
    – Figure 3.4 illustrates this sequence of operations.
  - Since the first element inserted into a queue is the first element removed, a stack is also known as a *First-In-First-Out* (*FIFO*) list.

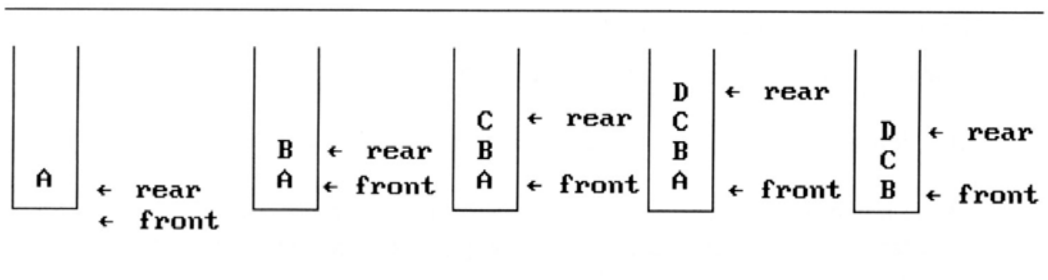


Figure 3.4: Inserting and deleting elements in a queue

– The ADT specification of the queue appears in Structure 3.2.

structure *Queue* is
  objects: a finite ordered list with zero or more elements.
  functions:
    for all $queue \in Queue$, $item \in element$, $max\_queue\_size \in$ 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 simplest way to implement this ADT is by 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`

```
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

– **Example 3.2 [*Job scheduling*]:**

- Figure 3.5 illustrates how an operating system might process jobs if it used a sequential representation for its queue.

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

– It should be obvious that as jobs enter and leave the system, the queue gradually shift to right. This means eventually the rear index equals *MAX_QUEUE_SIZE-1*, suggest the queue is full.

– In this case, *queue_full* should move the entire queue to the left so that the first element is again at *queue*[0] and *front* is at -1. It should also recalculate *rear* so that it is correctly positioned.

» Shifting an array is very time-consuming, particularly when there are many elements in it. In fact, *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 <u>circular</u>.

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

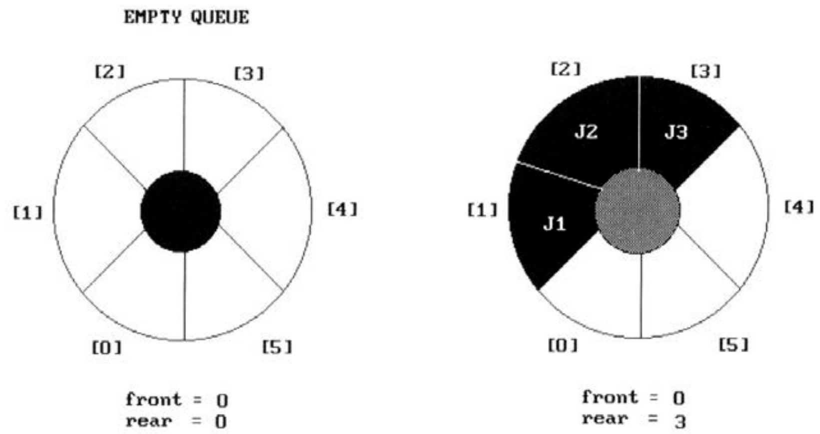


Figure 3.6: Empty and nonempty circular queues

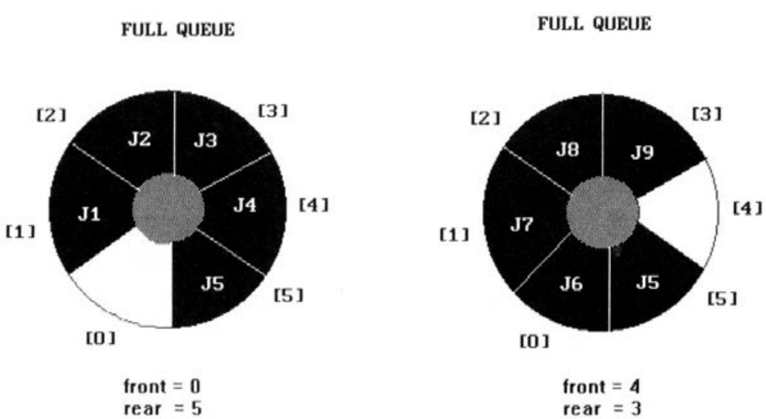


Figure 3.7: Full circular queues

Implementation

In order to implement such a circular queue, we need to perform two pointer operations:

*rear = (*rear+1)%Max_Q_size

*front = (*front+1)%Max_Q_size

Where % is the modulo operation and Max_Q_size is the queue size (6 in this example).

– Implementing *addq* and *deleteq* for a circular queue is slightly more difficult since we must assure that a circular rotation occurs.

```
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

3.3 A mazing problem

- In creating this program the first issue that confronts us is the representation of the maze.
  - The most obvious choice is a two dimensional array in which zeros represent the open paths and ones the barriers.
  - Figure 3.8 shows a simple maze.
  - Notice that not every position has eight neighbors.
    - To avoid checking for these border conditions we can surround the maze by a border of ones. Thus an $m \times p$ maze will require an $(m+2) \times (p+2)$ array.
    - The entrance is at position [1][1] and the exit at $[m][p]$.

entrance →	0	1	0	0	0	1	1	0	0	0	1	1	1	1	1	
	1	0	0	0	1	1	0	1	1	1	0	0	1	1	1	
	0	1	1	0	0	0	0	1	1	1	1	0	0	1	1	
	1	1	0	1	1	1	1	0	1	1	0	1	1	0	0	
	1	1	0	1	0	0	1	0	1	1	1	1	1	1	1	
	0	0	1	1	0	1	1	1	0	1	0	0	1	0	1	
	0	1	1	1	1	0	0	1	1	1	1	1	1	1	1	
	0	0	1	1	0	1	1	0	1	1	1	1	1	0	1	
	1	1	0	0	0	1	1	0	1	1	0	0	0	0	0	
	0	0	1	1	1	1	1	0	0	0	1	1	1	1	0	
	0	1	0	0	1	1	1	1	1	0	1	1	1	1	0	→ exit

Figure 3.8: An example maze

– If X marks the spot of our current location, maze[row][col], then Figure 3.9 shows the possible moves from this position.

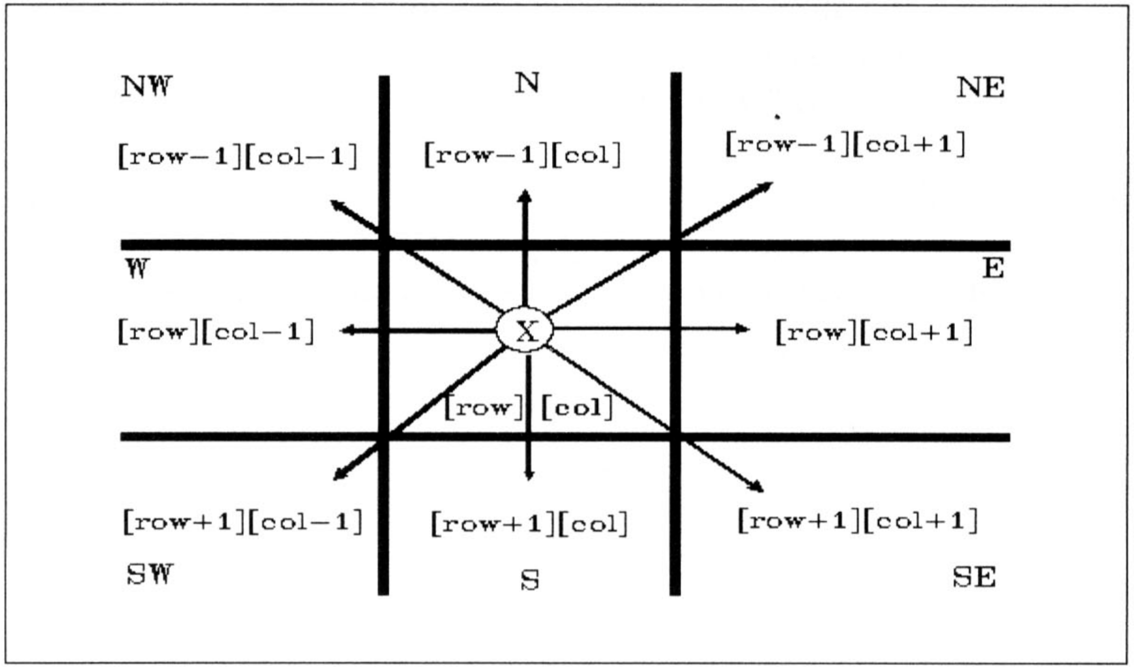


Figure 3.9: Allowable moves

– Another device that will simplify the problem is to predefine the possible directions to move in an array, *move*, as in Figure 3.10.

  - This is obtained from Figure 3.9.

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

Figure 3.10: Table of moves

– Program 3.7 is our initial attempt at a maze traversal algorithm.

- It use a stack to save our current position.
- It maintain a second two-dimensional array, *mark*, to record the maze positions already checked.

```
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 3.7: Initial maze algorithm

– Program 3.8 contains the maze search algorithm.

```
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;
      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 3.8: Maze search function

- **Analysis of *path*:**
  - The worst case of computing time of *path* is O(*mp*), where *m* and *p* are, respectively, the number of rows and columns of the maze.

(My recommendation to process a Maze problem)

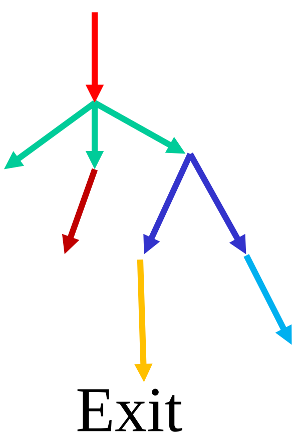
entrance →

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

→ exit

(設若是個小迷宮)

Entrance



可用 Stack 來輔助搜尋 tree

3.4 Evaluation of expressions

- Introduction
  - The representation and evaluation of expressions is of great interest to computer scientists.
    - `((rear+1==front)||((rear==MAX_QUEUE_SIZE-1)&&!front))` (3.1)
    - `x=a/b-c+d*e-a*c` (3.2)
  - If we examine expressions (3.1) and (3.2), we notice that they contains operators, operands, and parentheses.

- The first problem with understanding the meaning of these or any other expressions and statements is figuring out the order in which the operations are performed.
  - We would have written (3.2) differently by using parentheses to change the order of evaluation:
    `x=((a/(b-c+d))*(e-a)*c   (3.3)`
- Within any programming language, there is a precedence hierarchy that determines the order in which we evaluate operators.
  - Figure 3.12 contains the precedence hierarchy for C.
    (next page)

<table>
<tr><th>Token</th><th>Operator</th><th>Precedence[1]</th><th>Associativity</th></tr>
<tr><td>()
[]
-> .</td><td>function call
array element
struct or union member</td><td>17</td><td>left-to-right</td></tr>
<tr><td>-- ++</td><td>increment, decrement[2]</td><td>16</td><td>left-to-right</td></tr>
<tr><td>-- ++
!
~
- +
& *
sizeof</td><td>decrement, increment[3]
logical not
one's complement
unary minus or plus
address or indirection
size (in bytes)</td><td>15</td><td>right-to-left</td></tr>
<tr><td>(type)</td><td>type cast</td><td>14</td><td>right-to-left</td></tr>
<tr><td>* / %</td><td>multiplicative</td><td>13</td><td>left-to-right</td></tr>
<tr><td>+ -</td><td>binary add or subtract</td><td>12</td><td>left-to-right</td></tr>
<tr><td><< >></td><td>shift</td><td>11</td><td>left-to-right</td></tr>
<tr><td>> >=
< <=</td><td>relational</td><td>10</td><td>left-to-right</td></tr>
<tr><td>== !=</td><td>equality</td><td>9</td><td>left-to-right</td></tr>
<tr><td>&</td><td>bitwise and</td><td>8</td><td>left-to-right</td></tr>
<tr><td>^</td><td>bitwise exclusive or</td><td>7</td><td>left-to-right</td></tr>
<tr><td>|</td><td>bitwise or</td><td>6</td><td>left-to-right</td></tr>
<tr><td>&&</td><td>logical and</td><td>5</td><td>left-to-right</td></tr>
<tr><td>||</td><td>logical or</td><td>4</td><td>left-to-right</td></tr>
<tr><td>?:</td><td>conditional</td><td>3</td><td>right-to-left</td></tr>
<tr><td>= += -= /= *= %=
<<= >>= &= ^= |=</td><td>assignment</td><td>2</td><td>right-to-left</td></tr>
<tr><td>,</td><td>comma</td><td>1</td><td>left-to-right</td></tr>
</table>

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

Figure 3.12: Precedence hierarchy for C

- Evaluating postfix expressions
  - The standard way of writing expressions is known as infix notation because in it we place a binary operator in-between its two operands.
  - Although infix notation is the most common way of writing expressions, it is not the one used by compilers to evaluate expressions.
  - Instead compilers typically use a parenthesis-free notation referred to as postfix.
    - In this notation, each operator appears after its operands.
    - Figure 3.13 contains several infix expressions and their postfix equivalents.

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

- Evaluating postfix expressions is much simpler than the evaluation of infix expressions because:
  - There are no parentheses to consider.
  - To evaluate an expression we make **a single left-to-right scan** of it.
    - » We can evaluate an expression easily by using a stack.
    - » Figure 3.14 shows this processing when the input is nine character string 6 2/3-4 2*+.
    - »

Token	Stack [0]	Stack [1]	Stack [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

Infix

Postfix

X = A/B – C + D * E – A * C

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

$X = A/B - C + D * E - A * C$

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

$X = A/B - C + D * E - A * C$

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

X = A/B – C + D * E – A * C

$\boxed{\text{A B /}}$ C - D E * + A C * -

$X = A/B – C + D * E – A * C$

C - D E * + A C * -

$A/B = X_1$

$X = \boxed{A/B} - C + D * E - A * C$

X_1

C - D E * + A C * -

$X = \boxed{A/B} – C + D * E – A * C$

X_1 C

$- \quad D \quad E \quad * \quad + \quad A \quad C \quad * \quad -$

$X = \boxed{A/B} - C + D * E - A * C$

X_1 C -   D   E   *   +   A   C   *   -

$X = \boxed{A/B} - C + D * E - A * C$

D E * + A C * -

$X_1 - C = X_2$

$X = \boxed{A/B - C} + D * E - A * C$

X_2

D E * + A C * -

$X = \boxed{A/B - C} + D * E - A * C$

X_2 D E * + A C * -

$X = \boxed{A/B - C} + D * E - A * C$

X_2 D E * + A C * -

$X = \boxed{A/B - C} + D * E - A * C$

$X_2\ D\ E\ *\qquad +\ \ A\ \ C\ \ *\ \ -$

$X = \boxed{A/B - C} + D * E - A * C$

X_2

+ A C * -

$D*E = X_3$

$X = \boxed{A/B - C} + \boxed{D * E} - A * C$

$X_2\ X_3$

+ A C * -

$X = \boxed{A/B - C} + \boxed{D * E} - A * C$

$X_2 \ X_3 \ +$ A C * -

$X = \boxed{A/B - C} + \boxed{D * E} - A * C$

A C * -

$X_2 + X_3 = X_4$

$X = \boxed{A/B - C + D * E} - A * C$

X_4

A C * -

$X = \boxed{A/B - C + D * E} - A * C$

X_4 A	C	*	-

$X = \boxed{A/B - C + D * E} - A * C$

X_4 A C * -

$X = \boxed{A/B - C + D * E} - A * C$

X_4 A C *   -

$X = \boxed{A/B - C + D * E} - A * C$

X_4 -

$A * C = X_5$

$X = \boxed{A/B - C + D * E} - \boxed{A * C}$

$X_4 \ X_5$ -

$X = \boxed{A/B - C + D * E} - \boxed{A * C}$

$X_4\ X_5\ -$

$X = \boxed{A/B - C + D * E - A * C}$

$x_4 - x_5 = X$

Back

- 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 */
```

(Next page: program 3.9, 3.10)

```
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

```
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

- Infix to postfix

  (第一法: 加括號的作法)

  - We can describe an algorithm for producing a postfix expression from an infix one as follows:
    1) Fully parenthesize the expression.
    2) Move all binary operators so that they replace their corresponding right parentheses.
    3) Delete all parentheses.
  - Example:

    a/b-c+d*e-a*c when fully parenthesize becomes:

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

    Performing steps 2 and 3 gives

    ab/c-de*+ac*-

(第二法: 不必加括號的作法)

Case 1. 沒有（）

- 變數，放入 output
- Incoming operator, 看 stack 裡最靠外的 op 的 precedence 之高低。**若** stack 內的 op 的 prec. **較低**，才將 incoming op 壓入 stack. **若較高**，則將 stack 的 op 先彈出，然後才將 incoming op 壓入 stack. **若相等**，則將 stack 裡 op 彈出，才將 incoming op 壓入

– **Example 3.3 [*Simple expression*]:** Suppose we have the simple expression a+b*c, which yields abc*+ in postfix.

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

– **Example 3.3 [*Simple expression*]:** We use as our example the expression a*(b+c)*d, which yields abc+*d* in postfix.

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

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

Case 2. 有（）

基本上與上面 case 一樣，只是：

- 在 “(” 壓入 stack 之後，作法就和上面一樣。
- 直到碰到 incoming 的是“)”，則將 stack 內 op 全彈出，直到碰到第一個 “(” 為止，並將“(” delete 掉。

- The function *postfix* (Program 3.11) convert an infix expression into a postfix one.

```
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

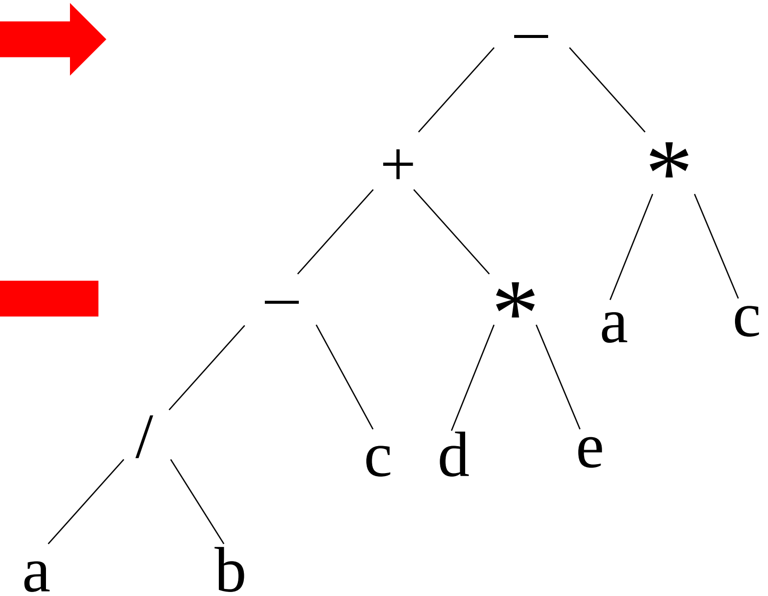
- **Analysis of *postfix*:**
  - Complexity of function postfix is $\Theta(n)$, where n is the number of tokens in the expression.

<u>Another Method</u>

製作成 binary tree

Infix a/b – c + d*e – a*c

Postfix ab/c – de* + ac* –



Visiting order (都是 depth first search)

Infix: LVR Postfix: LRV Prefix: VLR

Postfix Notation

Expressions are converted into Postfix notation before compiler can accept and process them.

$X = A/B – C + D * E – A * C$

Infix A/B–C+D*E–A*C

Postfix => AB/C-DE*+AC*-

Operation	Postfix
$T_1 = A / B$	T_1C-DE*+AC*-
$T_2 = T_1 - C$	T_2 DE*+AC*-
$T_3 = D * E$	T_2T_3+AC*-
$T_4 = T_2 + T_3$	T_4AC*-
$T_5 = A * C$	T_4 T_5 -
$T_6 = T_4 - T_5$	T_6