Data Structures

G.P. Biswas

Prof/IIT, Dhanbad

Stacks and Queues

Two of the more common data objects found in computer algorithms are stacks and queues. They arise so often that we will discuss them separately before moving on to more complex objects. Both these data objects are special cases of the more general data object, an ordered list which we considered in the previous chapter. Recall that $A = (a_1, a_2, \ldots, a_n)$, is an ordered list of $n \geqq 0$ elements. The a_i are referred to as atoms which are taken from some set. The null or empty list has $n = 0$ elements.

A *stack* is an ordered list in which all insertions and deletions are made at one end, called the *top*. A *queue* is an ordered list in which all insertions take place at one end, the *rear*, while all deletions take place at the other end, the *front*. Given a stack $S = (a_1, \ldots, a_n)$ then we say that a_1 is the *bottommost* element and element a_i is on *top* of element a_{i-1}, $1 < i \leqslant n$. When viewed as a queue with a_n as the rear element one says that a_{i+1} is behind a_i, $1 \leqslant i < n$.

Contd.

- If elements *A, B, C, D* and *E* are added in that order, then, the restrictions on stack and queue are:

added to the stack, in that order, then the first element to be removed / deleted must be E. Equivalently we say that the last element to be inserted into the stack will be the first to be removed. For this reason stacks are sometimes referred to as *Last In First Out* (LIFO) lists. The restrictions on a queue require that the first element which is inserted into the queue will be the first one to be removed. Thus A is the first letter to be removed, and queues are known as *First In First Out* (FIFO) lists. Note that the data object queue as defined here need not necessarily correspond to the mathematical concept of queue in which the insert / delete rules may be different.

An Example for Stack Application

One natural example of stacks which arises in computer programming is the processing of subroutine calls and their returns. Suppose we have a main procedure and three subroutines as below:

proc MAIN	proc A1	proc A2	proc A3
___	___	___	___
___	___	___	___
___	___	___	___
call A1	call A2	call A3	___
r:	s:	t:	
___	___	___	___
___	___	___	___
end	end	end	end

Figure 3.2. Sequence of subroutine calls

The MAIN program calls subroutine $A1$. On completion of $A1$ execution of MAIN will resume at location r. The address r is passed to $A1$ which saves it in some location for later processing. $A1$ then invokes $A2$ which in turn calls $A3$. In each case the calling procedure passes the return address to the called procedure. If we examine the memory while $A3$ is computing there will be an implicit stack which looks like

$$(q,r,s,t).$$

The first entry, q, is the address in the operating system where MAIN returns control. This list operates as a stack since the returns will be made in the reverse order of the calls. Thus t is removed before s,

ADT of Stack Data Structure

Associated with the object stack there are several operations that are necessary:

CREATE(S) which creates S as an empty stack;

ADD(i,S) which inserts the element i onto the stack S and returns the new stack;

DELETE(S) which removes the top element of stack S and returns the new stack;

TOP(S) which returns the top element of stack S;

ISEMTS(S) which returns true if S is empty else false;

These five functions constitute a working definition of a stack.

However we choose to represent a stack, it must be possible to build these operations. But before we do this let us describe formally the structure STACK.

```
structure STACK (item)
   declare CREATE( ) → stack
           ADD(item,stack) → stack
           DELETE(stack) → stack
           TOP(stack) → item
           ISEMTS(stack) → boolean;
   for all S ∈ stack, i ∈ item let
        ISEMTS (CREATE)       :: = true
        ISEMTS (ADD(i,S))     :: = false
        DELETE (CREATE)       :: = error
        DELETE (ADD(i,S))     :: = S
        TOP (CREATE)          :: = error
        TOP (ADD(i,S))        :: = i
   end
end STACK
```

Stack Representation using Array

The simplest way to represent a stack is by using a one-dimensional array, say STACK($1:n$), where n is the maximum number of allowable entries. The first or bottom element in the stack will be stored at STACK(1), the second at STACK(2) and the i-th at STACK(i). Associated with the array will be a variable, top, which points to the top element in the stack. With this decision made the following implementations result:

CREATE ( ) :: = **declare** *STACK*(1:*n*); *top* $\leftarrow$ 0
ISEMTS(*STACK*) :: = **if** *top* = 0 **then true**
else false
TOP(*STACK*) :: = **if** *top* = 0 **then** *error*
else *STACK*(*top*)

ADD and DELETE Operations on Stack

```
procedure ADD (item, STACK, n, top)
  //insert item into the STACK of maximum size n; top is the number
    of elements currently in STACK//
    if top ≥ n then call STACK_FULL
    top ← top + 1
    STACK (top) ← item
end ADD

procedure DELETE (item, STACK, top)
  //removes the top element of STACK and stores it in item
    unless STACK is empty//
    if top ≤ 0 then call STACK_EMPTY
    item ← STACK (top)
    top ← top −1
end DELETE
```

Queue Data Structure

As mentioned earlier, when we talk of queues we talk about two distinct ends: the front and the rear. Additions to the queue take place at the rear. Deletions are made from the front. So, if a job is submitted for execution, it joins at the rear of the job queue. The job at the front of the queue is the next one to be executed. A minimal set of useful operations on a queue includes the following:

CREATEQ(Q) which creates Q as an empty queue;
ADDQ(i,Q) which adds the element i to the rear of a queue and returns the new queue;
DELETEQ(Q) which removes the front element from the queue Q and returns the resulting queue;
FRONT(Q) which returns the front element of Q;
ISEMTQ(Q) which returns true if Q is empty else false.

ADT of Queue Data Structure

A complete specification of this data structure is

```
structure QUEUE (item)
   declare CREATEQ( ) → queue
           ADDQ(item, queue) → queue
           DELETEQ(queue) → queue
           FRONT(queue) → item
           ISEMTQ(queue) → boolean;
   for all Q ∈ queue, i ∈ item let
       ISEMTQ(CREATEQ)     :: = true
       ISEMTQ(ADDQ(i,Q))   :: = false
       DELETEQ(CREATEQ)    :: = error
       DELETEQ(ADDQ(i,Q))  :: =
         if ISEMTQ(Q) then CREATEQ
           else ADDQ(i, DELETEQ(Q))
       FRONT(CREATEQ)      :: = error
       FRONT(ADDQ(i,Q))    :: =
         if ISEMTQ(Q) then i else FRONT(Q)
   end
end QUEUE
```

Queue representation using Array

The representation of a finite queue in sequential locations is somewhat more difficult than a stack. In addition to a one dimensional array $Q(1:n)$, we need two variables, *front* and *rear*. The conventions we shall adopt for these two variables are that *front* is always 1 less than the actual front of the queue and *rear* always points to the last element in the queue. Thus, *front* = *rear* if and only if there are no elements in the queue. The initial condition then is *front* = *rear* = 0. With these conventions, let us try an example by inserting and deleting jobs, J_i, from a job queue.

front	rear	Q(1)	(2)	(3)	(4)	(5)	(6)	(7)	. . .	Remarks
0	0		queue		empty					Initial
0	1	J1								Job 1 joins Q
0	2	J1	J2							Job 2 joins Q
0	3	J1	J2	J3						Job 3 joins Q
1	3		J2	J3						Job 1 leaves Q
1	4		J2	J3	J4					Job 4 joins Q
2	4			J3	J4					Job 2 leaves Q

ADD and DELETE Operations

With this scheme, the following implementation of the CREATEQ, ISEMTQ, and FRONT operations results for a queue with capacity n:

```
CREATEQ(Q) ::= declare Q(1:n); front ← rear ← 0
ISEMTQ(Q)  ::= if front = rear then true
                                else false
FRONT(Q)   ::= if ISEMTQ(Q) then error
                                 else Q(front + 1)
```

The following algorithms for ADDQ and DELETEQ result:

```
procedure ADDQ(item, Q, n, rear)
  //insert item into the queue represented in Q(1:n)//
    if rear = n then call QUEUE__FULL
    rear ← rear + 1
    Q(rear) ← item
end ADDQ

procedure DELETEQ(item, Q, front, rear)
  //delete an element from a queue//
    if front = rear then call QUEUE__EMPTY
    front ← front + 1
    item ← Q(front)
end DELETEQ
```

Queue Contd.

The representation of a finite queue in sequential locations is somewhat more difficult than a stack. In addition to a one dimensional array $Q(1:n)$, we need two variables, *front* and *rear*. The conventions we

shall adopt for these two variables are that *front* is always 1 less than the actual front of the queue and *rear* always points to the last element in the queue. Thus, *front* = *rear* if and only if there are no elements in the queue. The initial condition then is *front* = *rear* = 0. With these conventions, let us try an example by inserting and deleting jobs, J_i, from a job queue.

front	rear	Q(1)	(2)	(3)	(4)	(5)	(6)	(7)	. . .	Remarks
0	0		queue		empty					Initial
0	1	J1								Job 1 joins Q
0	2	J1	J2							Job 2 joins Q
0	3	J1	J2	J3						Job 3 joins Q
1	3		J2	J3						Job 1 leaves Q
1	4		J2	J3	J4					Job 4 joins Q
2	4			J3	J4					Job 2 leaves Q

Contd.

The correctness of this implementation may be established in a manner akin to that used for stacks. With this set up, notice that unless the front regularly catches up with the rear and both pointers are reset to zero, then the QUEUE_FULL signal does not necessarily imply that there are n elements in the queue. That is, the queue will gradually move to the right. One obvious thing to do when QUEUE_FULL is signaled is to move the entire queue to the left so that the first element is again at $Q(1)$ and front $= 0$. This is time consuming, especially when there are many elements in the queue at the time of the QUEUE_FULL signal.

Let us look at an example which shows what could happen, in the worst case, if each time the queue becomes full we choose to move the entire queue left so that it starts at $Q(1)$. To begin, assume there are n elements $J_1, \ldots, J_n$ in the queue and we next receive alternate requests to delete and add elements. Each time a new element is added, the entire queue of $n - 1$ elements is moved left.

front	rear	Q(1)	(2)	(3)	......	(n)	next operation
0	n	J_1	J_2	J_3	......	J_n	initial state
1	n		J_2	J_3	......	J_n	delete J_1
0	n	J_2	J_3	J_4	......	J_{n+1}	add J_{n+1} (jobs J_2 through J_n are moved)
1	n		J_3	J_4	......	J_{n+1}	delete J_2
0	n	J_3	J_4	J_5	......	J_{n+2}	add J_{n+2}

Circular Queue

A more efficient queue representation is obtained by regarding the array $Q(1{:}n)$ as circular. It now becomes more convenient to declare the array as $Q(0{:}n-1)$. When rear $= n - 1$, the next element is entered at $Q(0)$ in case that spot is free. Using the same conventions as before, *front* will always point one position counterclockwise from the first element in the queue. Again, *front* = *rear* if and only if the queue is empty. Initially we have *front* = *rear* = 1. Figure 3.4 illustrates some of the possible configurations for a circular queue containing the four elements $J1$–$J4$ with $n > 4$. The assumption of circularity changes the ADD and DELETE algorithms slightly. In order to add an element, it will be necessary to move *rear* one position clockwise, i.e.,

if $rear = n - 1$ **then** $rear \leftarrow 0$
else $rear \leftarrow rear + 1$.

Contd.

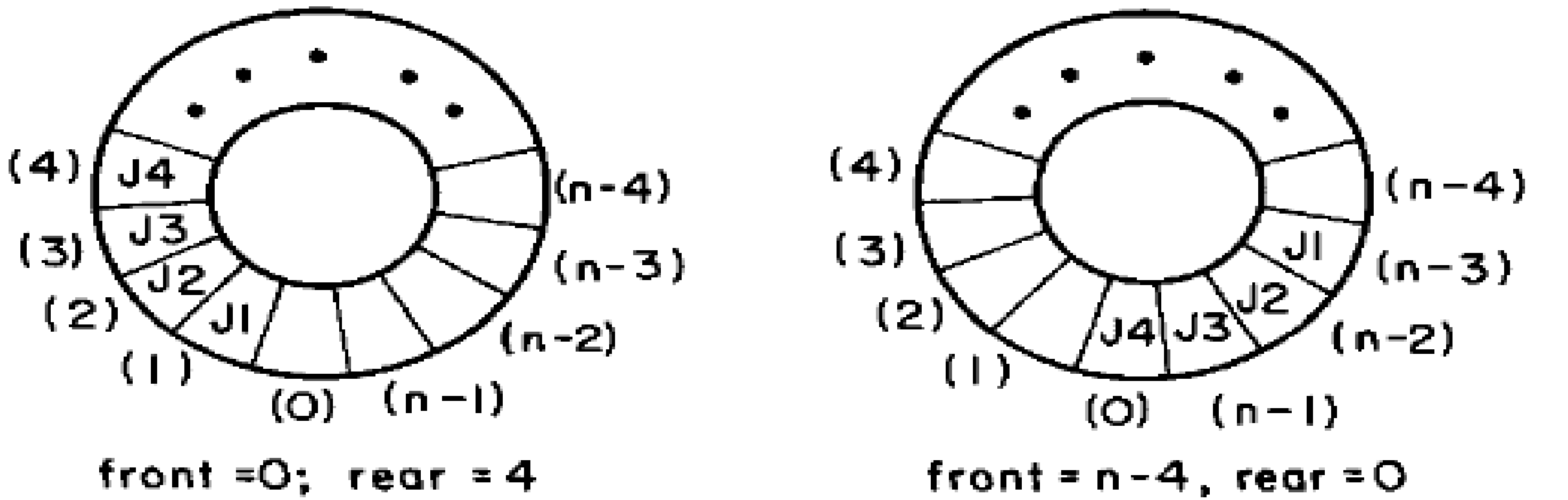

Figure 3.4: Circular queue of n elements and four jobs $J1$, $J2$, $J3$, $J4$

Using the modulo operator which computes remainders, this is just $rear \leftarrow (rear + 1) \textbf{ mod } n$. Similarly, it will be necessary to move *front* one position clockwise each time a deletion is made. Again, using the modulo operation, this can be accomplished by $front \leftarrow (front + 1) \textbf{ mod } n$. An examination of the algorithms indicates that addition and deletion can now be carried out in a fixed amount of time or $O(1)$.

ADD and DELETE Operations on Circular Queue

Using the modulo operator which computes remainders, this is just $rear \leftarrow (rear + 1) \textbf{mod}\ n$. Similarly, it will be necessary to move *front* one position clockwise each time a deletion is made. Again, using the modulo operation, this can be accomplished by $front \leftarrow (front + 1) \textbf{mod}\ n$. An examination of the algorithms indicates that addition and deletion can now be carried out in a fixed amount of time or $O(1)$.

```
procedure ADDQ(item, Q, n, front, rear)
   //insert item in the circular queue stored in Q(0:n − 1);
     rear points to the last item and front is one position
     counterclockwise from the first item in Q//
   rear ← (rear + 1)mod n          //advance rear clockwise//
   if front = rear then call QUEUE-FULL
   Q(rear) ← item          //insert new item//
end ADDQ

procedure DELETEQ(item, Q, n, front, rear)
   //removes the front element of the queue Q(0:n − 1)//
   if front = rear then call QUEUE-EMPTY
   front ← (front + 1)mod n          //advance front clockwise//
   item ← Q(front)          //set item to front of queue//
end DELETEQ
```

Circular Queue Contd.

One surprising point in the two algorithms is that the test for queue full in ADDQ and the test for queue empty in DELETEQ are the same. In the case of ADDQ, however, when *front* = *rear* there is actually one space free, i.e. Q(*rear*), since the first element in the queue is not at Q(*front*) but is one position clockwise from this point. However,

if we insert an item here, then we will not be able to distinguish between the cases full and empty, since this insertion would leave *front* = *rear*. To avoid this, we signal queue-full, thus permitting a maximum of $n - 1$ rather than n elements to be in the queue at any time. One way to use all n positions would be to use another variable, *tag*, to distinguish between the two situations, i.e. *tag* = 0 if and only if the queue is empty. This would however slow down the two algorithms. Since the ADDQ and DELETEQ algorithms will be used many times in any problem involving queues, the loss of one queue position will be more than made up for by the reduction in computing time.

Evaluation of Expression

- Consider the following expression:

$$X \leftarrow A / B ** C + D * E - A * C \qquad (3.1)$$

might have several meanings; and even if it were uniquely defined, say by a full use of parentheses, it still seemed a formidable task to generate a correct and reasonable instruction sequence. Fortunately the

An expression is made up of operands, operators and delimiters. The expression above has five operands: A, B, C, D, and E. Though these are all one letter variables, operands can be any legal variable name or constant in our programming language. In any expression the values that variables take must be consistent with the operations performed on them. These operations are described by the operators. In most programming languages there are several kinds of operators which correspond to the different kinds of data a variable can hold. First, there are the basic arithmetic operators: plus, minus, times, divide, and exponentiation ($+, -, *, /, **$). Other arithmetic operators include unary plus, unary minus and **mod, ceil,** and **floor**. The latter three may sometimes be library subroutines rather than predefined operators. A second class are the relational operators: $<, \not<, \leq, =, \neq, \geq, \not>, >$. These are usually

Contd

The first problem with understanding the meaning of an expression is to decide in what order the operations are carried out. This means that every language must uniquely define such an order. For instance, if $A = 4$, $B = C = 2$, $D = E = 3$, then in eq. 3.1 we might want X to be assigned the value

$$\begin{aligned} & 4/(2 ** 2) + (3 * 3) - (4 * 2) \\ = & (4/4) + 9 - 8 \\ = & 2. \end{aligned}$$

However, the true intention of the programmer might have been to assign X the value

$$\begin{aligned} & (4/2) ** (2 + 3) * (3 - 4) * 2 \\ = & (4/2) ** 5 * -1 * 2 \\ = & (2 ** 5) * -2 \\ = & 32 * -2 \\ = & -64. \end{aligned}$$

Of course, he could specify the latter order of evaluation by using parentheses:

$$X \leftarrow ((((A / B) ** (C + D)) * (E - A)) * C).$$

Priority of Operators

To fix the order of evaluation, we assign to each operator a priority. Then within any pair of parentheses we understand that operators with the highest priority will be evaluated first. A set of sample priorities from PL/*I* is given in Figure 3.7.

Operator	Priority
$**$, unary $-$, unary $+$, $\neg$	6
$*$, $/$	5
$+$, $-$	4
$<$, $\nless$, $\leqslant$, $=$, $\neq$, $\geqslant$, $>$, $\ngtr$	3
and	2
or	1

exponentiation, unary minus, unary plus and Boolean negation all have top priority. When we have an expression where two adjacent operators have the same priority, we need a rule to tell us which one to perform first. For example, do we want the value of $-A ** B$ to be understood as $(-A) ** B$ or $-(A ** B)$? Convince yourself that there will be a difference by trying $A = -1$ and $B = 2$. From algebra we normally consider $A ** B ** C$ as $A ** (B ** C)$ and so we rule that operators in priority 6 are evaluated right-to-left. However, for expressions such as $A * B / C$ we generally execute left-to-right or $(A * B) / C$. So we rule that for all other priorities, evaluation of operators of the same priority will proceed left to right. Remember that by using parentheses we can override these rules, and such expressions are always evaluated with the innermost parenthesized expression first.

Infix to Postfix Translation

Now that we have specified priorities and rules for breaking ties we know how $X \leftarrow A / B ** C + D * E - A * C$ will be evaluated, namely as

$$X \leftarrow ((A / (B ** C)) + (D * E)) - (A * C).$$

How can a compiler accept such an expression and produce correct code? The answer is given by reworking the expression into a form we call postfix notation. If e is an expression with operators and operands, the conventional way of writing e is called *infix*, because the operators come *in*-between the operands. (Unary operators precede their operand.) The *postfix* form of an expression calls for each operator to appear *after* its operands. For example,

$$\text{infix: } A * B / C \text{ has postfix: } AB * C / .$$

If we study the postfix form of $A * B / C$ we see that the multiplication comes immediately after its two operands A and B. Now imagine that $A * B$ is computed and stored in T. Then we have the division operator, $/$, coming immediately after its two arguments T and C.

Let us look at our previous example

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

$$\text{postfix: } ABC ** / DE * + AC * -$$

and trace out the meaning of the postfix.

Let us look at our previous example

infix: $A/B ** C + D * E - A * C$

postfix: $ABC ** / DE * + AC * -$

and trace out the meaning of the postfix.

Every time we compute a value let us store it in the temporary location T_i, $i \geqq 1$. Reading left to right, the first operation is exponentiation:

Operation	Postfix
$T_1 \leftarrow B ** C$	$AT_1 / DE * + AC * -$
$T_2 \leftarrow A / T_1$	$T_2 DE * + AC * -$
$T_3 \leftarrow D * E$	$T_2 T_3 + AC * -$
$T_4 \leftarrow T_2 + T_3$	$T_4 AC * -$
$T_5 \leftarrow A * C$	$T_4 T_5 -$
$T_6 \leftarrow T_4 - T_5$	T_6

So T_6 will contain the result. Notice that if we had parenthesized the expression, this would change the postfix only if the order of normal evaluation were altered. Thus, $A/(B ** C) + (D * E) - A * C$ will have the same postfix form as the previous expression without parentheses. But $(A/B) ** (C + D) * (E - A) * C$ will have the postfix form $AB/CD + ** EA - * C *$.

Before attempting an algorithm to translate expressions from infix to postfix notation, let us make some observations regarding the virtues of postfix notation that enable easy evaluation of expressions. To begin with, the need for parentheses is eliminated. Secondly, the priority of the operators is no longer relevant. The expression may be evaluated by making a left to right scan, stacking operands, and evaluating operators using as operands the correct number from the stack and finally placing the result onto the stack. This evaluation process is much simpler than attempting a direct evaluation from infix notation.

```
procedure EVAL (E)
  // evaluate the postfix expression E. It is assumed that the
     last character in E is an '∞'. A procedure NEXT-TOKEN is
     used to extract from E the next token. A token is either an
     operand, operator, or '∞'. A one dimensional array STACK(1:n) is
     used as a stack //
     top ← 0 // initialize STACK //
     loop
        x ← NEXT-TOKEN(E)
        case
           : x = '∞' : return // answer is at top of stack //
           : x is an operand: call ADD(x,STACK,n,top)
           :else: remove the correct number of operands
                  for operator x from STACK, perform
                  the operation and store the result, if
                  any, onto the stack
        end
     forever
end EVAL
```

To see how to devise an algorithm for translating from infix to postfix, note that the order of the operands in both forms is the same. In fact, it is simple to describe an algorithm for producing postfix from infix:

1) fully parenthesize the expression;
2) move all operators so that they replace their corresponding right parentheses;
3) delete all parentheses.

For example, $A / B ** C + D * E - A * C$ when fully parenthesized yields

$$(((A / (B ** C)) + (D * E)) - (A * C)).$$

The arrows point from an operator to its corresponding right parenthesis. Performing steps 2 and 3 gives

$$ABC ** / DE * + AC * -.$$

The problem with this as an algorithm is that it requires two passes: the first one reads the expression and parenthesizes it while the second actually moves the operators. As we have already observed, the order of the operands is the same in infix and postfix. So as we scan an expression for the first time, we can form the postfix by immediately passing any operands to the output. Then it is just a matter of handling the operators. The solution is to store them in a stack until just the right moment and then to unstack and pass them to the output.

For example, since we want $A + B * C$ to yield $ABC * +$ our algorithm should perform the following sequence of stacking (these stacks will grow to the right):

Next Token	Stack	Output
none	empty	none
A	empty	A
+	+	A
B	+	AB

At this point the algorithm must determine if * gets placed on top of the stack or if the + gets taken off. Since * has greater priority we should stack * producing

*	+ *	AB
C	+ *	ABC

Now the input expression is exhausted, so we output all remaining operators in the stack to get

ABC * +

For another example, $A * (B + C) * D$ has the postfix form $ABC + * D *$, and so the algorithm should behave as

Next Token	Stack	Output
none	empty	none
A	empty	A
*	*	A
(	* (	A
B	* (	AB
+	* (+	AB
C	* (+	ABC

At this point we want to unstack down to the corresponding left parenthesis, and then delete the left and right parentheses; this gives us:

)	*	ABC +
*	*	ABC + *
D	*	ABC + * D
done	empty	ABC + * D *

These examples should motivate the following hierarchy scheme for binary arithmetic operators and delimiters. The general case involving all the operators of figure 3.7 is left as an exercise.

Symbol	In-Stack Priority	In-Coming Priority
)	—	—
**	3	4
*, /	2	2
binary +, −	1	1
(	0	4

Figure 3.8 Priorities of Operators for Producing Postfix

The rule will be that *operators are taken out of the stack as long as their in-stack priority, isp, is greater than or equal to the in-coming priority, icp* of the new operator. ISP(X) and ICP(X) are functions which reflect the table of figure 3.8.

- x

```
procedure POSTFIX (E)
  //convert the infix expression E to postfix.  Assume the last
    character of E is a '∞', which will also be the last character of
    the postfix.  Procedure NEXT-TOKEN returns either the next
    operator, operand or delimiter—whichever comes next.
    STACK(1:n) is used as a stack and the character '-∞' with
    ISP('-∞') = -1 is used at the bottom of the stack. ISP and ICP
    are functions.//
  STACK(1) ← '-∞'; top ← 1          //initialize stack//
```

```
loop
  x ← NEXT-TOKEN(E)
  case
    :x = '∞': while top > 1 do // empty the stack //
                  print(STACK(top)); top ← top − 1
              end
              print ('∞')
              return
    :x is an operand: print (x)
    :x = ')': while STACK(top) ≠ '(' do // unstack until ')' //
                print (STACK(top)); top ← top − 1
              end
              top ← top − 1          //delete '(' //
      :else: while ISP(STACK(top)) ≥ ICP(x) do
               print (STACK(top)); top ← top − 1
             end
             call ADD(x,STACK,n,top)        //insert x in STACK//
  end
 forever
end POSTFIX
```

As for the computing time, the algorithm makes only one pass across the input. If the expression has n symbols, then the number of operations is proportional to some constant times n. The stack cannot get any deeper than the number of operators plus 1, but it may achieve that bound as it does for $A + B * C ** D$.