Unit-II

STACKS AND QUEUES

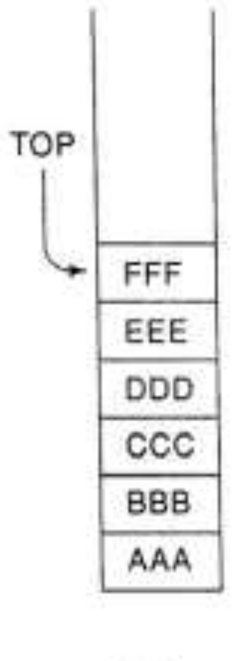
Stack

- A stack is a list of elements in which an element may be inserted or deleted only at one end.
- This end is called the **top** of the stack.
- The elements can be removed in the reverse order of that in which they were inserted into the stack.
- Follows LIFO (**L**ast **I**n **F**irst **O**ut)
- There are two basic operations associated with stacks:
  - PUSH
    - To insert element into a stack.
  - POP
    - To delete element from a stack.

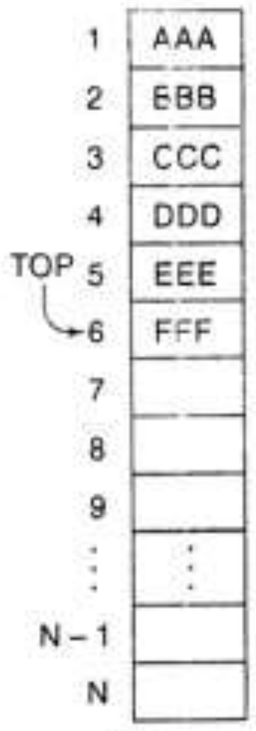
Example

- Suppose following 6 elements are pushed in the given order into the stack:

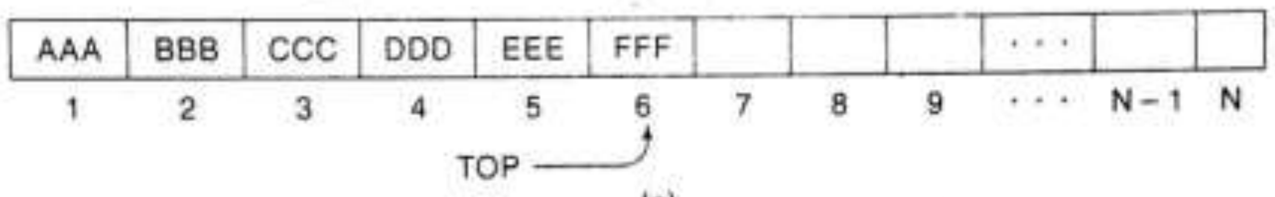
  AAA, BBB, CCC, DDD, EEE, FFF



(a)



(b)

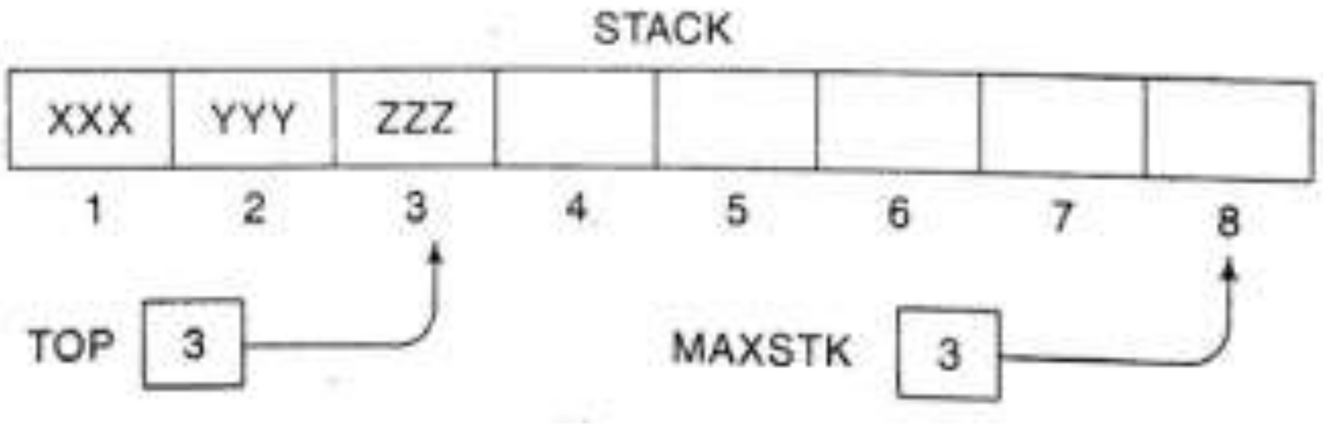


(c)

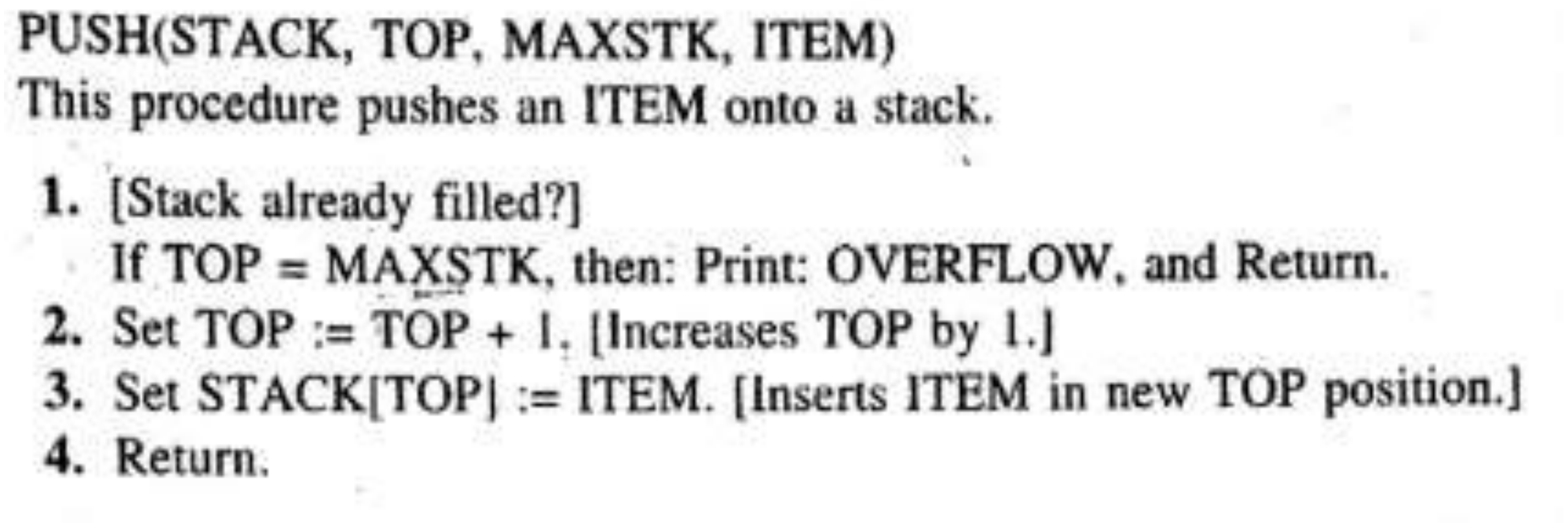
Array Representation of Stacks

- Each of the stack will be maintained by:
  - A linear array **STACK**
  - A pointer variable **TOP**
    - contains the location of top element of the stack.
  - A variable **MAXSTK**
    - gives the maximum number of elements that can be held by a stack.
  - If TOP=0 or TOP=NULL
    - indicates that the stack is empty.

Array Representation of Stacks



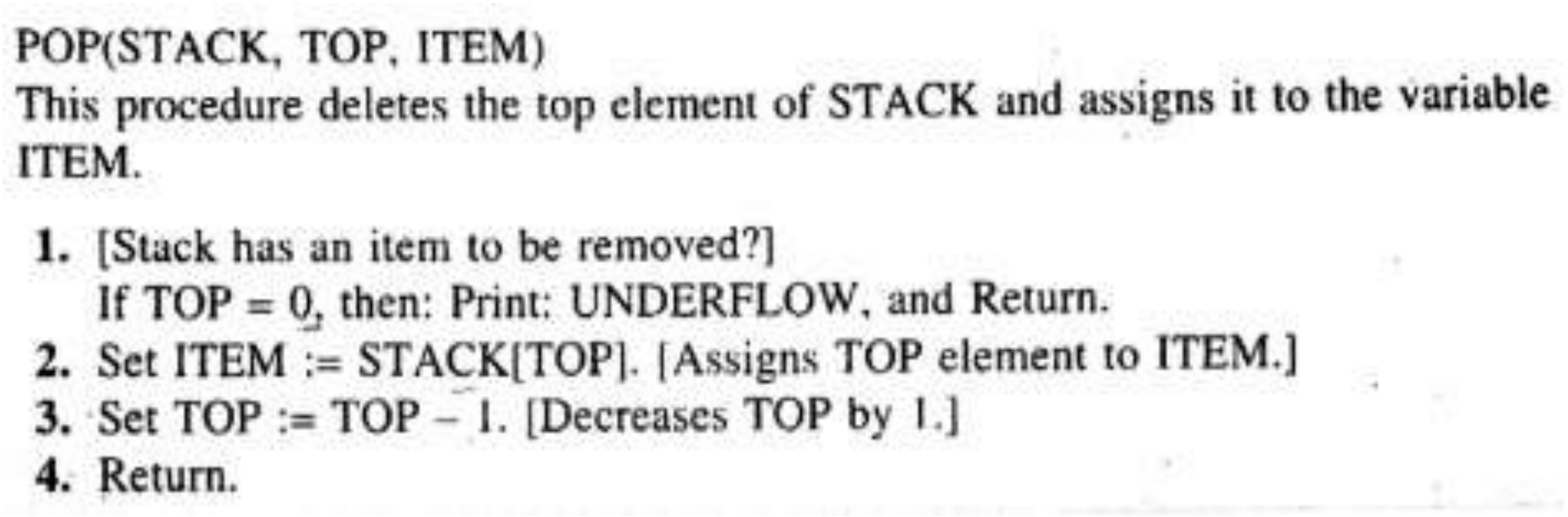
Algorithm to insert an element into Stack



PUSH(STACK, TOP, MAXSTK, ITEM)
This procedure pushes an ITEM onto a stack.

```
1. [Stack already filled?]
   If TOP = MAXSTK, then: Print: OVERFLOW, and Return.
2. Set TOP := TOP + 1. [Increases TOP by 1.]
3. Set STACK[TOP] := ITEM. [Inserts ITEM in new TOP position.]
4. Return.
```

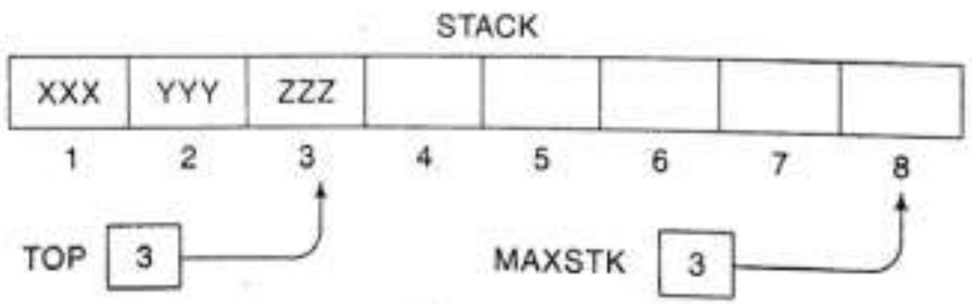
Algorithm to delete an element from Stack



POP(STACK, TOP, ITEM)
This procedure deletes the top element of STACK and assigns it to the variable ITEM.

```
1. [Stack has an item to be removed?]
   If TOP = 0, then: Print: UNDERFLOW, and Return.
2. Set ITEM := STACK[TOP]. [Assigns TOP element to ITEM.]
3. Set TOP := TOP – 1. [Decreases TOP by 1.]
4. Return.
```

Example



(a) Consider the stack in Fig. 6.5. We simulate the operation PUSH(STACK, WWW):

1. Since TOP = 3, control is transferred to Step 2.
2. TOP = 3 + 1 = 4.
3. STACK[TOP] = STACK[4] = WWW.
4. Return.

Note that WWW is now the top element in the stack.

(b) Consider again the stack in Fig. 6.5. This time we simulate the operation POP(STACK, ITEM):

1. Since TOP = 3, control is transferred to Step 2.
2. ITEM = ZZZ.
3. TOP = 3 − 1 = 2.
4. Return.

Observe that STACK[TOP] = STACK[2] = YYY is now the top element in the stack.

Arithmetic Expressions

- Let Q be an arithmetic expression.
- Binary operations in Q may have different levels of precedence as follows:
  - Highest: Exponentiation(↑)
  - Next highest: Multiplication(*) and Division(/)
  - Lowest: Addition(+) and Subtraction(-)

- Operations in the same level are performed from left to right.

- Example:

  2 ↑ 3 + 5 * 2 ↑ 2 – 12 / 6

Arithmetic Expressions: Notations

- Types of notations depends upon the position of operators and operands as mentioned below:
  - Infix Notation
    - Operator symbol is placed between two operands

      (A+B)*C

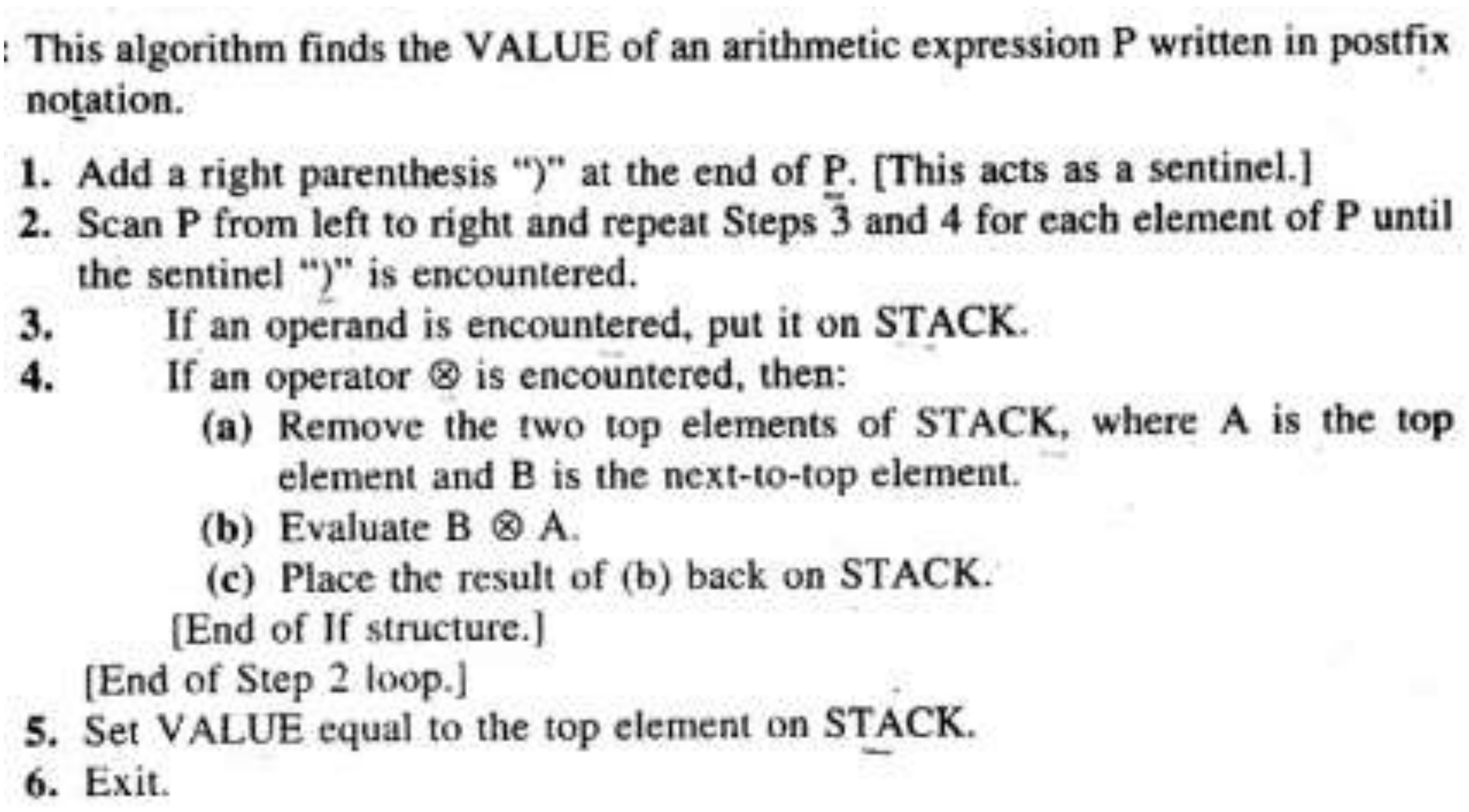
  - Polish Notation (Prefix Notation)
    - Operator symbol is placed before two operands

      (A+B)*C = [+AB]*C = *+ABC

  - Reverse Polish Notation (Postfix Notation)
    - Operator symbol is placed after two operands

      (A+B)*C = [AB+]*C = AB+C*

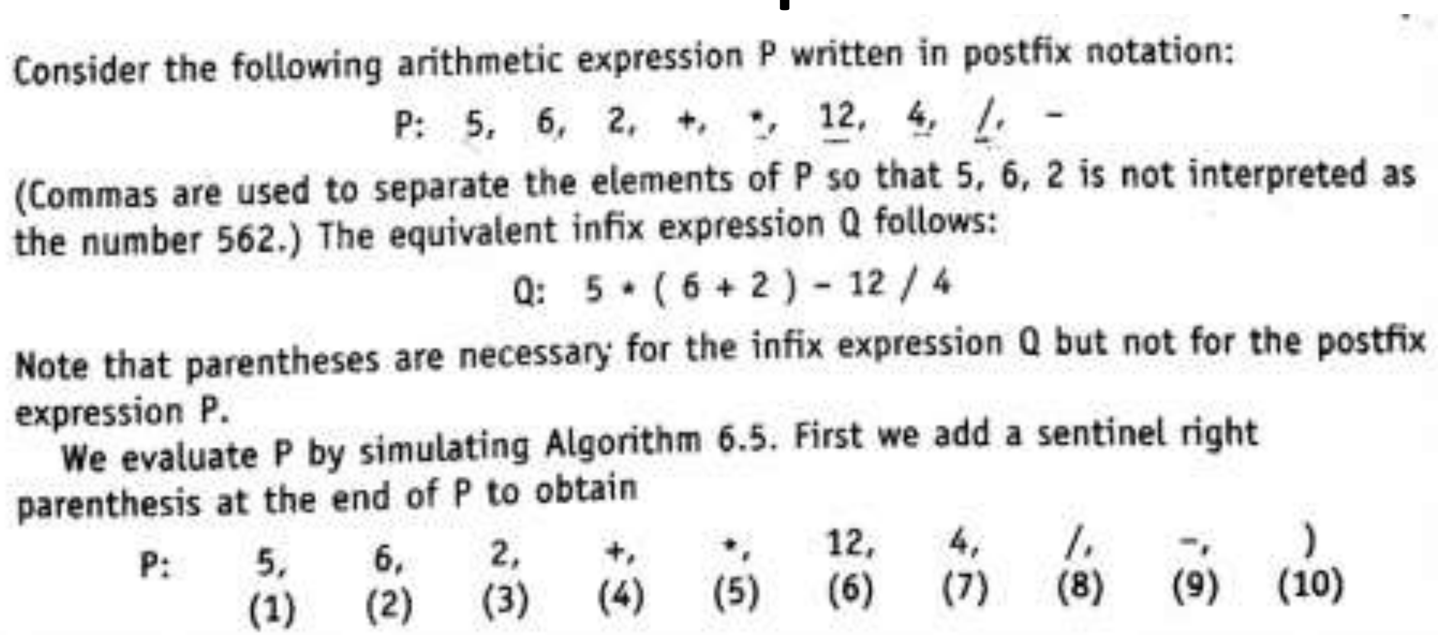
Evaluation of Postfix Notation



```
: This algorithm finds the VALUE of an arithmetic expression P written in postfix
  notation.

1. Add a right parenthesis ")" at the end of P. [This acts as a sentinel.]
2. Scan P from left to right and repeat Steps 3 and 4 for each element of P until
   the sentinel ")" is encountered.
3.        If an operand is encountered, put it on STACK.
4.        If an operator ⊗ is encountered, then:
            (a) Remove the two top elements of STACK, where A is the top
                element and B is the next-to-top element.
            (b) Evaluate B ⊗ A.
            (c) Place the result of (b) back on STACK.
          [End of If structure.]
   [End of Step 2 loop.]
5. Set VALUE equal to the top element on STACK.
6. Exit.
```

Example



Consider the following arithmetic expression P written in postfix notation:

P: 5, 6, 2, +, *, 12, 4, /, −

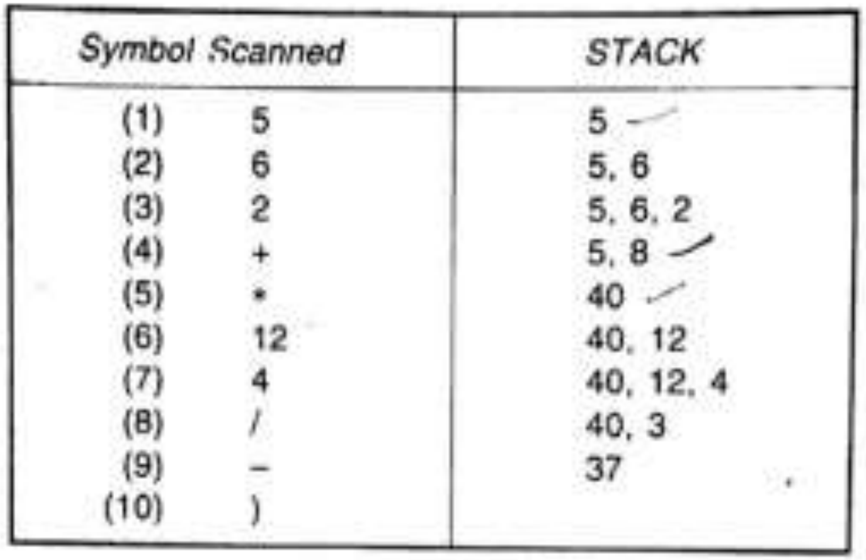
(Commas are used to separate the elements of P so that 5, 6, 2 is not interpreted as the number 562.) The equivalent infix expression Q follows:

Q: 5 * (6 + 2) − 12 / 4

Note that parentheses are necessary for the infix expression Q but not for the postfix expression P.

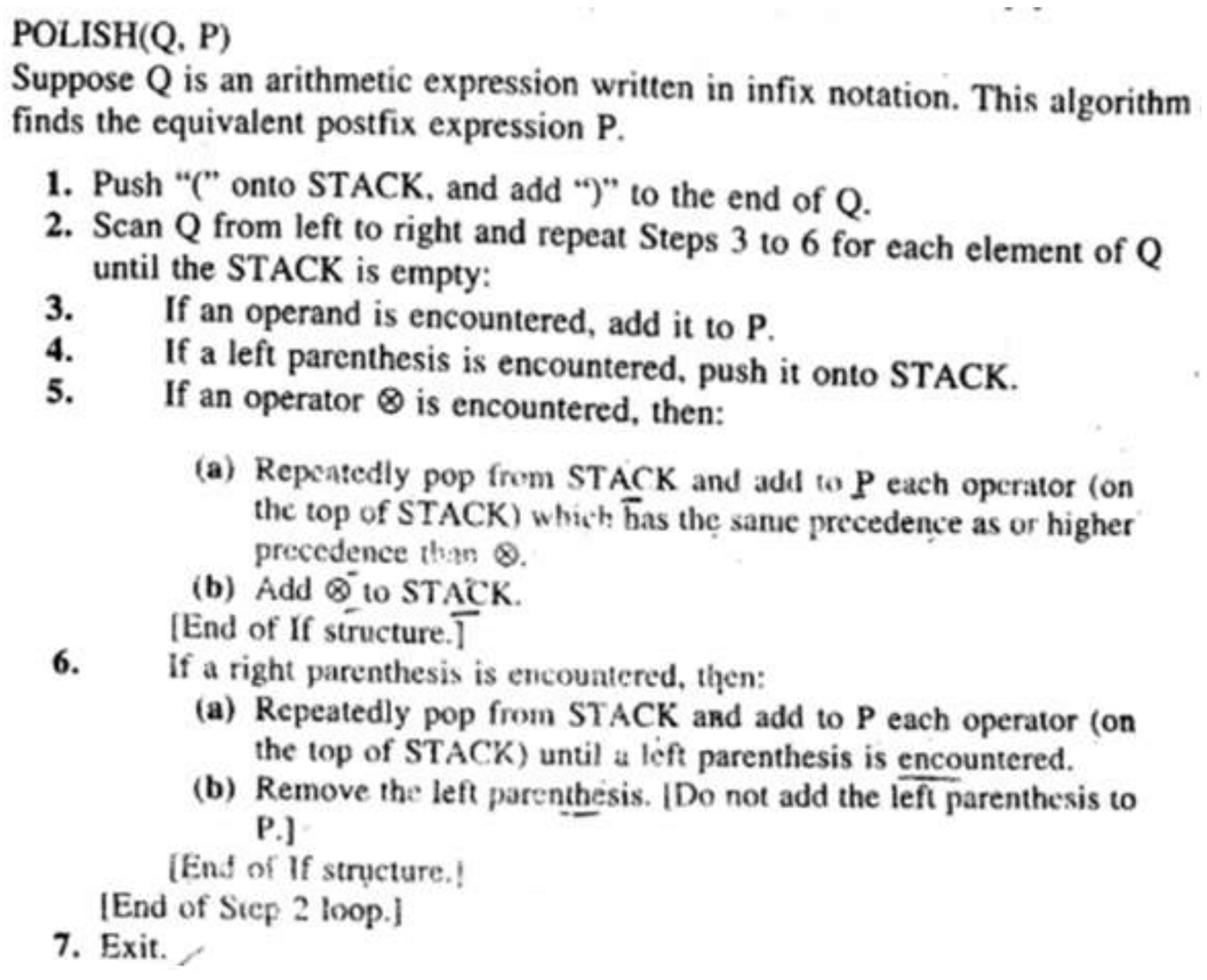
We evaluate P by simulating Algorithm 6.5. First we add a sentinel right parenthesis at the end of P to obtain

P:	5,	6,	2,	+,	*,	12,	4,	/,	−,	)
	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)



Symbol Scanned		*STACK*
(1)	5	5
(2)	6	5, 6
(3)	2	5, 6, 2
(4)	+	5, 8
(5)	*	40
(6)	12	40, 12
(7)	4	40, 12, 4
(8)	/	40, 3
(9)	−	37
(10)	)	

Infix to Postfix conversion

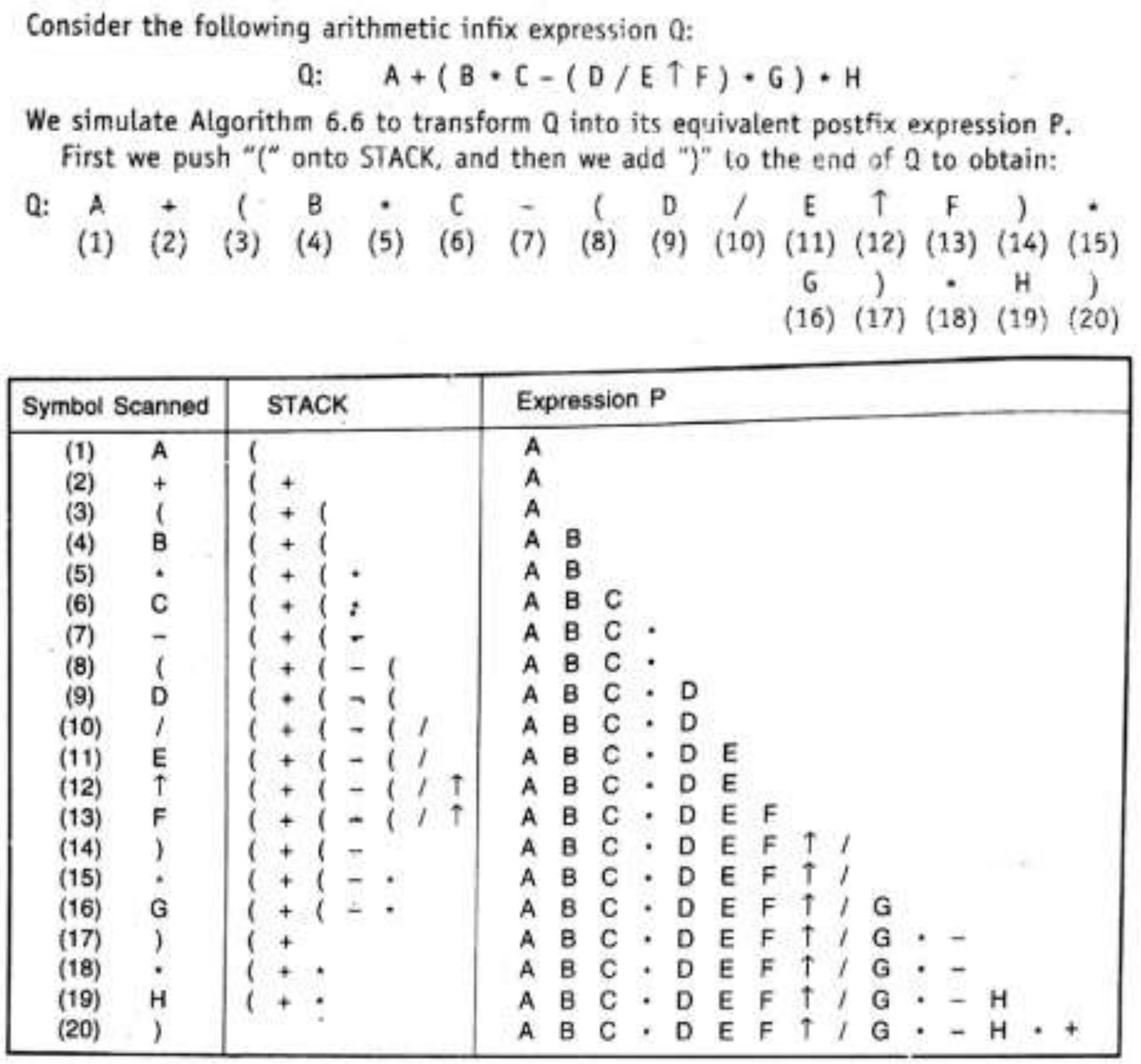


POLISH(Q, P)

Suppose Q is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression P.

```
1. Push "(" onto STACK, and add ")" to the end of Q.
2. Scan Q from left to right and repeat Steps 3 to 6 for each element of Q
   until the STACK is empty:
3.       If an operand is encountered, add it to P.
4.       If a left parenthesis is encountered, push it onto STACK.
5.       If an operator ⊗ is encountered, then:
           (a) Repeatedly pop from STACK and add to P each operator (on
               the top of STACK) which has the same precedence as or higher
               precedence than ⊗.
           (b) Add ⊗ to STACK.
         [End of If structure.]
6.       If a right parenthesis is encountered, then:
           (a) Repeatedly pop from STACK and add to P each operator (on
               the top of STACK) until a left parenthesis is encountered.
           (b) Remove the left parenthesis. [Do not add the left parenthesis to
               P.]
         [End of If structure.]
   [End of Step 2 loop.]
7. Exit.
```

Example



Consider the following arithmetic infix expression Q:

Q: A + (B * C - (D / E ↑ F) * G) * H

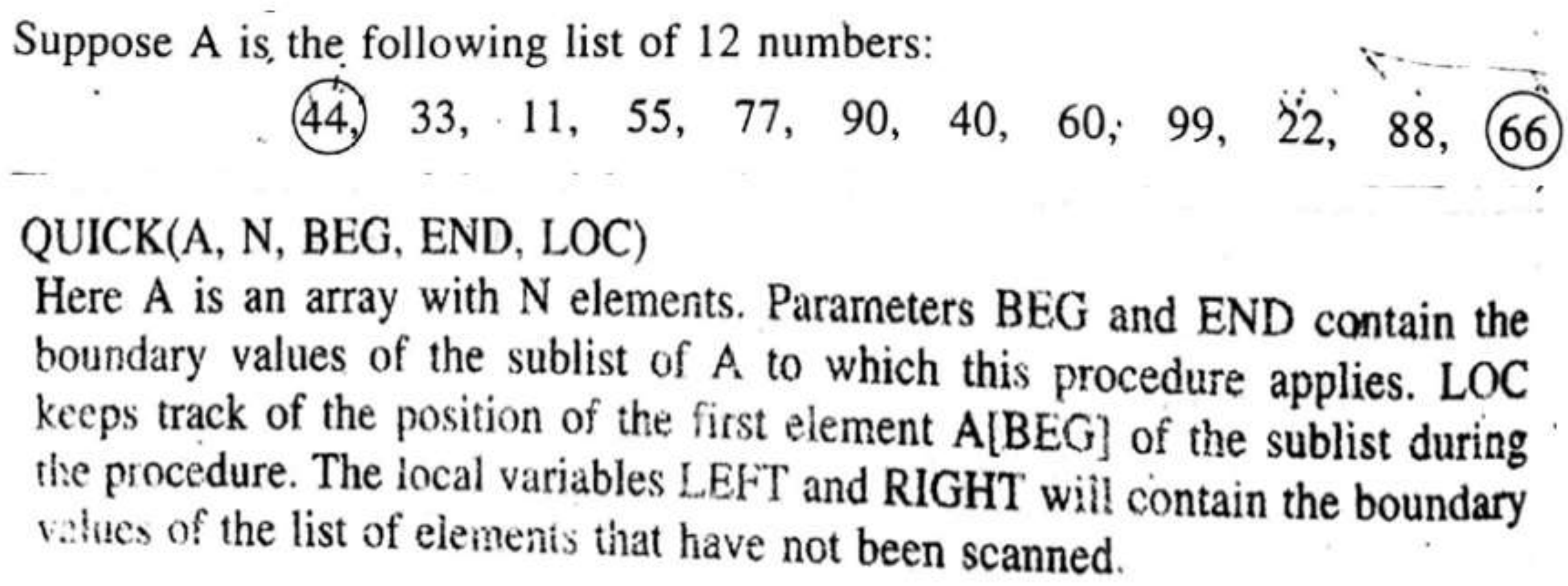
We simulate Algorithm 6.6 to transform Q into its equivalent postfix expression P.

First we push "(" onto STACK, and then we add ")" to the end of Q to obtain:

Q:	A	+	(	B	*	C	-	(	D	/	E	↑	F	)	*
	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)
											G	)	*	H	)
											(16)	(17)	(18)	(19)	(20)

Symbol Scanned		STACK	Expression P
(1)	A	(	A
(2)	+	(+	A
(3)	(	(+ (	A
(4)	B	(+ (	A B
(5)	*	(+ (*	A B
(6)	C	(+ (*	A B C
(7)	-	(+ (-	A B C *
(8)	(	(+ (- (	A B C *
(9)	D	(+ (- (	A B C * D
(10)	/	(+ (- (/	A B C * D
(11)	E	(+ (- (/	A B C * D E
(12)	↑	(+ (- (/ ↑	A B C * D E
(13)	F	(+ (- (/ ↑	A B C * D E F
(14)	)	(+ (-	A B C * D E F ↑ /
(15)	*	(+ (- *	A B C * D E F ↑ /
(16)	G	(+ (- *	A B C * D E F ↑ / G
(17)	)	(+	A B C * D E F ↑ / G * -
(18)	*	(+ *	A B C * D E F ↑ / G * -
(19)	H	(+ *	A B C * D E F ↑ / G * - H
(20)	)		A B C * D E F ↑ / G * - H * +

Quick Sort: Application of Stack



Suppose A is the following list of 12 numbers:

44, 33, 11, 55, 77, 90, 40, 60, 99, 22, 88, 66

QUICK(A, N, BEG, END, LOC)

Here A is an array with N elements. Parameters BEG and END contain the boundary values of the sublist of A to which this procedure applies. LOC keeps track of the position of the first element A[BEG] of the sublist during the procedure. The local variables LEFT and RIGHT will contain the boundary values of the list of elements that have not been scanned.

: (Quicksort) This algorithm sorts an array A with N elements.

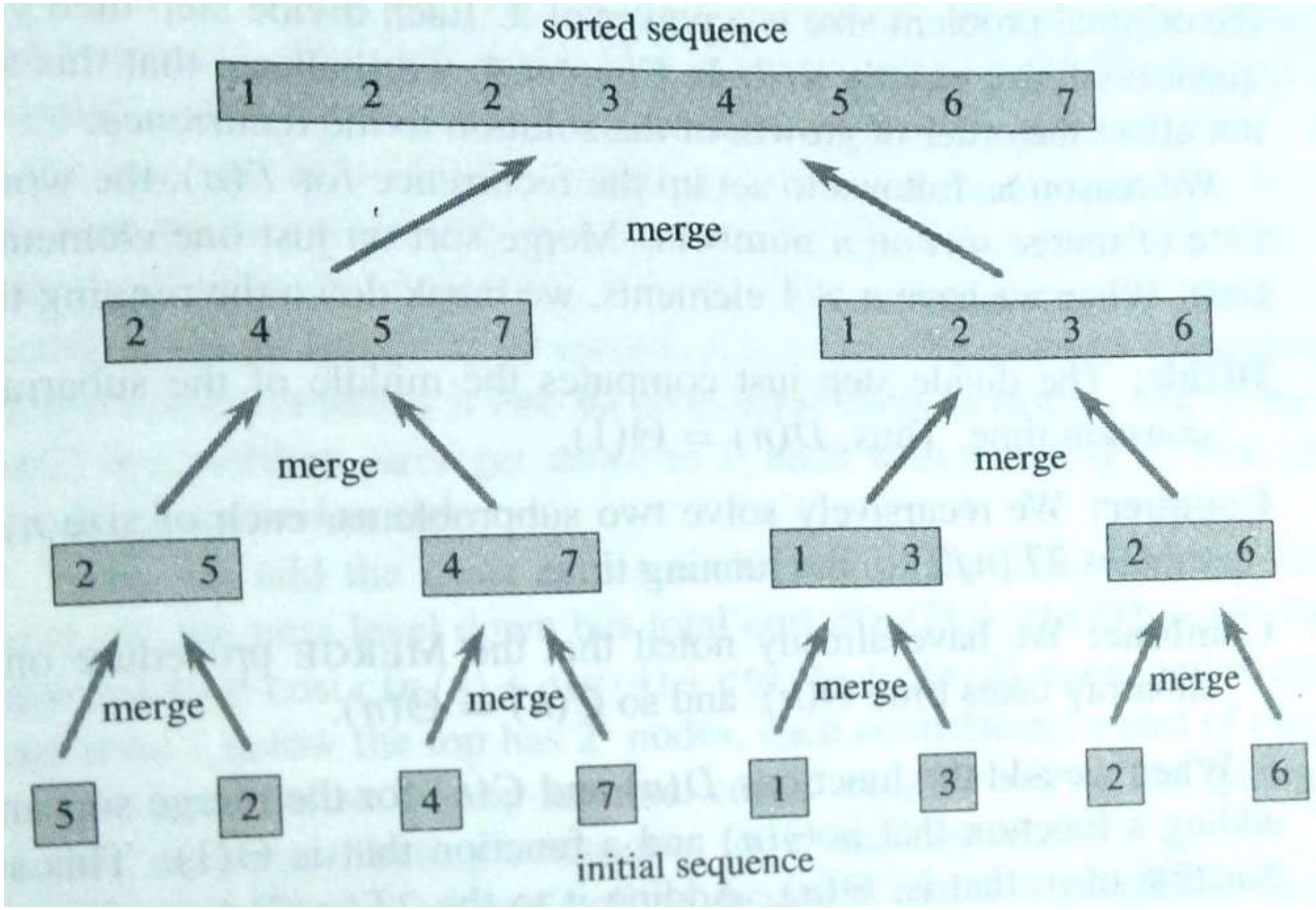
```
1. [Initialize.] TOP := NULL.
2. [Push boundary values of A onto stacks when A has 2 or more elements.]
   If N > 1, then: TOP := TOP + 1, LOWER[1] := 1, UPPER[1] := N.
3. Repeat Steps 4 to 7 while TOP ≠ NULL.
4.         [Pop sublist from stacks.]
           Set BEG := LOWER[TOP], END := UPPER[TOP].
           TOP := TOP - 1.
5.         Call QUICK(A, N, BEG, END, LOC). [Procedure 6.5.]
6.         [Push left sublist onto stacks when it has 2 or more elements.]
           If BEG < LOC - 1, then:
                TOP := TOP + 1, LOWER[TOP] := BEG,
                UPPER[TOP] = LOC - 1.
           [End of If structure.]
7.         [Push right sublist onto stacks when it has 2 or more elements.]
           If LOC + 1 < END, then:
                TOP := TOP + 1, LOWER[TOP] := LOC + 1.
                UPPER[TOP] := END.
           [End of If structure.]
   [End of Step 3 loop.]
8. Exit.
```

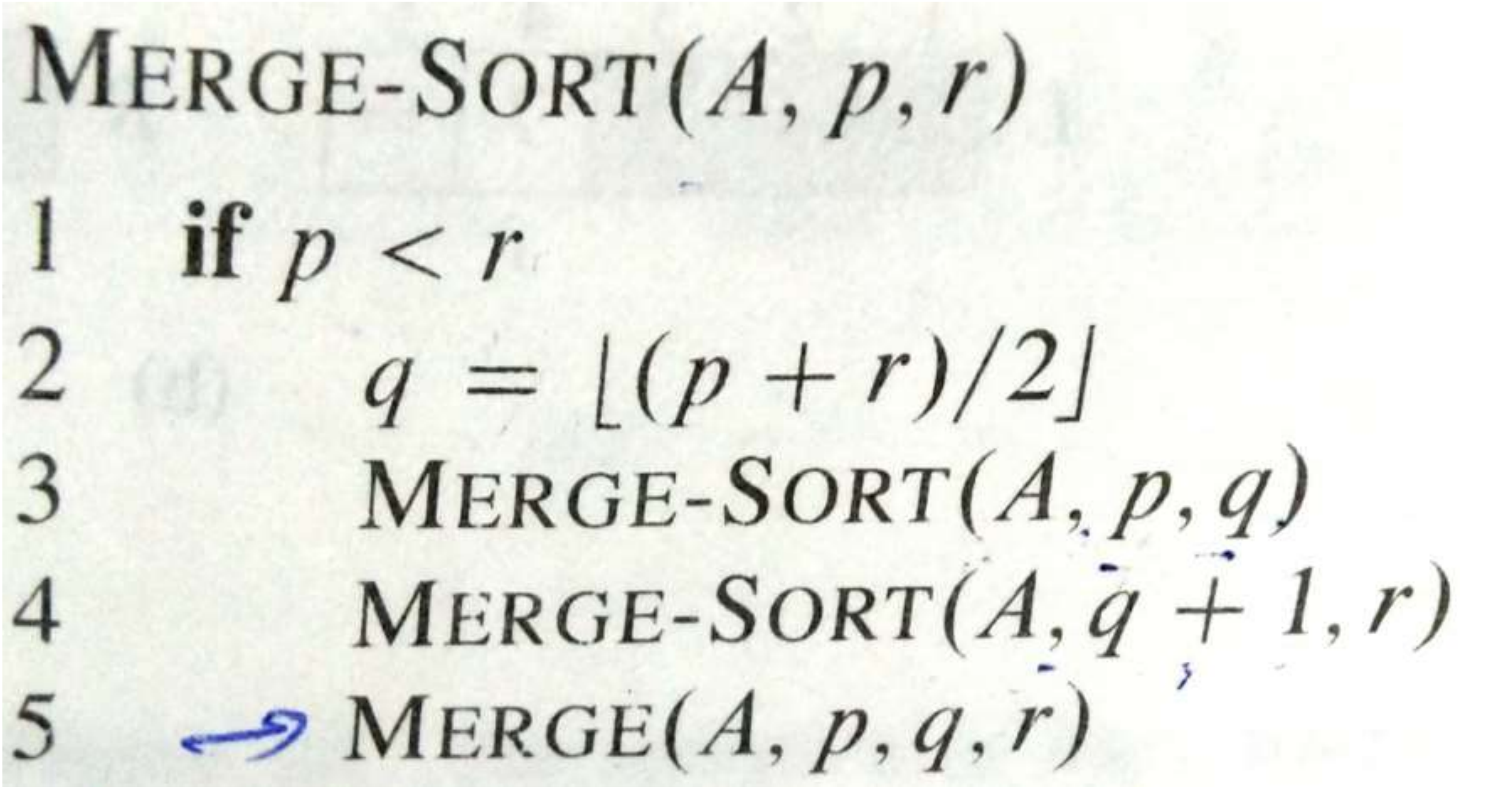
```
1. [Initialize.] Set LEFT := BEG, RIGHT := END and LOC := BEG.
2. [Scan from right to left.]
   (a) Repeat while A[LOC] ≤ A[RIGHT] and LOC ≠ RIGHT:
           RIGHT := RIGHT - 1.
       [End of loop.]
   (b) If LOC = RIGHT, then: Return.
   (c) If A[LOC] > A[RIGHT], then:
           (i) [Interchange A[LOC] and A[RIGHT].]
                   TEMP := A[LOC], A[LOC] := A[RIGHT],
                   A[RIGHT] := TEMP.
           (ii) Set LOC := RIGHT.
           (iii) Go to Step 3.
       [End of If structure.]
3. [Scan from left to right.]
   (a) Repeat while A[LEFT] ≤ A[LOC] and LEFT ≠ LOC:
           LEFT := LEFT + 1.
       [End of loop.]
   (b) If LOC = LEFT, then: Return.
   (c) If A[LEFT] > A[LOC], then
           (i) [Interchange A[LEFT] and A[LOC].]
                   TEMP := A[LOC], A[LOC] := A[LEFT],
                   A[LEFT] := TEMP.
           (ii) Set LOC := LEFT.
           (iii) Go to Step 2.
       [End of If structure.]
```

Merge Sort: The divide-and-conquer approach

- Merge sort is a technique which closely follows the divide and conquer paradigm as follows:
  - Divide: divide the n-element sequence to be sorted into two subsequences of n/2 elements each.
  - Conquer: sort the two subsequences recursively using merge-sort.
  - Combine: merge the two sorted subsequences to produce the sorted answer.

Example


sorted sequence
1 2 2 3 4 5 6 7
merge
2 4 5 7
1 2 3 6
merge
merge
2 5
4 7
1 3
2 6
merge
merge
merge
merge
5
2
4
7
1
3
2
6
initial sequence



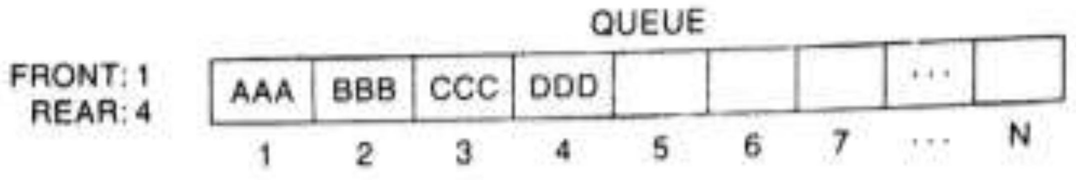
```
MERGE-SORT($A, p, r$)
1  if $p < r$
2      $q = \lfloor (p + r)/2 \rfloor$
3      MERGE-SORT($A, p, q$)
4      MERGE-SORT($A, q + 1, r$)
5      MERGE($A, p, q, r$)
```

```
MERGE(A, p, q, r)
$n_1 = q - p + 1$
$n_2 = r - q$
let $L[1 \,..\, n_1 + 1]$ and $R[1 \,..\, n_2 + 1]$ be new arrays
for $i = 1$ to $n_1$
    $L[i] = A[p + i - 1]$
for $j = 1$ to $n_2$
    $R[j] = A[q + j]$
$L[n_1 + 1] = \infty$
$R[n_2 + 1] = \infty$
$i = 1$
$j = 1$
for $k = p$ to $r$
    if $L[i] \leq R[j]$
        $A[k] = L[i]$
        $i = i + 1$
    else $A[k] = R[j]$
        $j = j + 1$
```

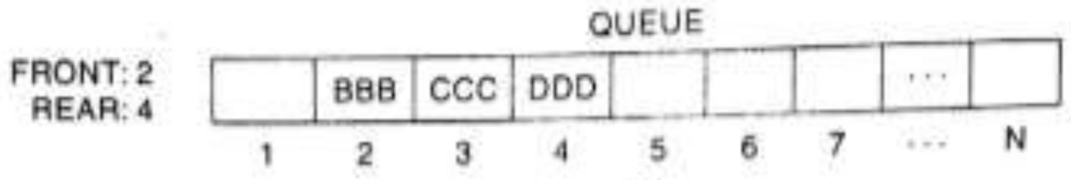
Queues

- A queue is a linear list of elements in which:
  - deletions can take place only at one end called the **Front.**
  - insertions can take place only at the other end called **Rear**.
- Queues are called **F**irst-**I**n-**F**irst-**O**ut lists.
  - The order in which elements enter a queue is the order in which they leave

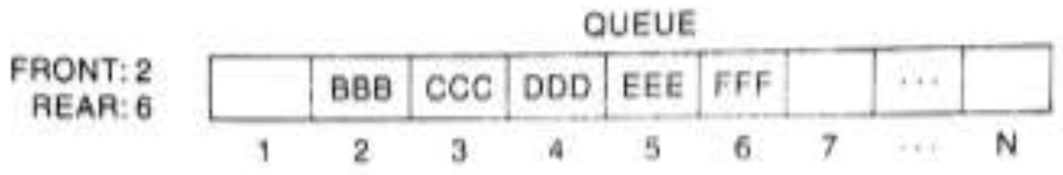
Array representation of queue



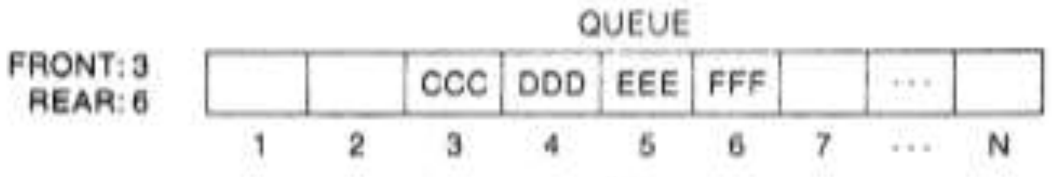
(a)



(b)



(c)



(d)

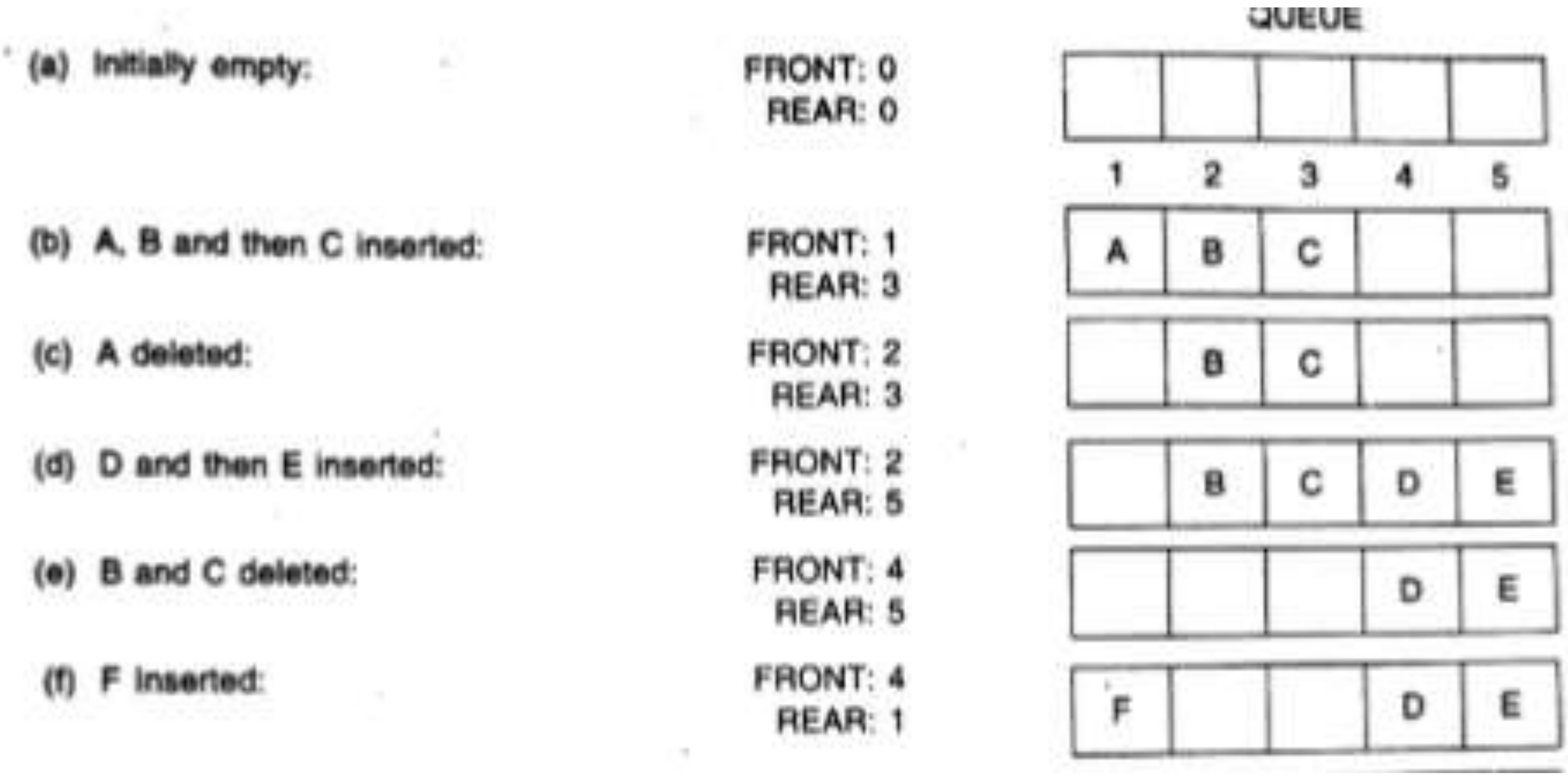
- If a queue contains only one element:

  FRONT = REAR ≠ NULL

- If a queue is empty then:

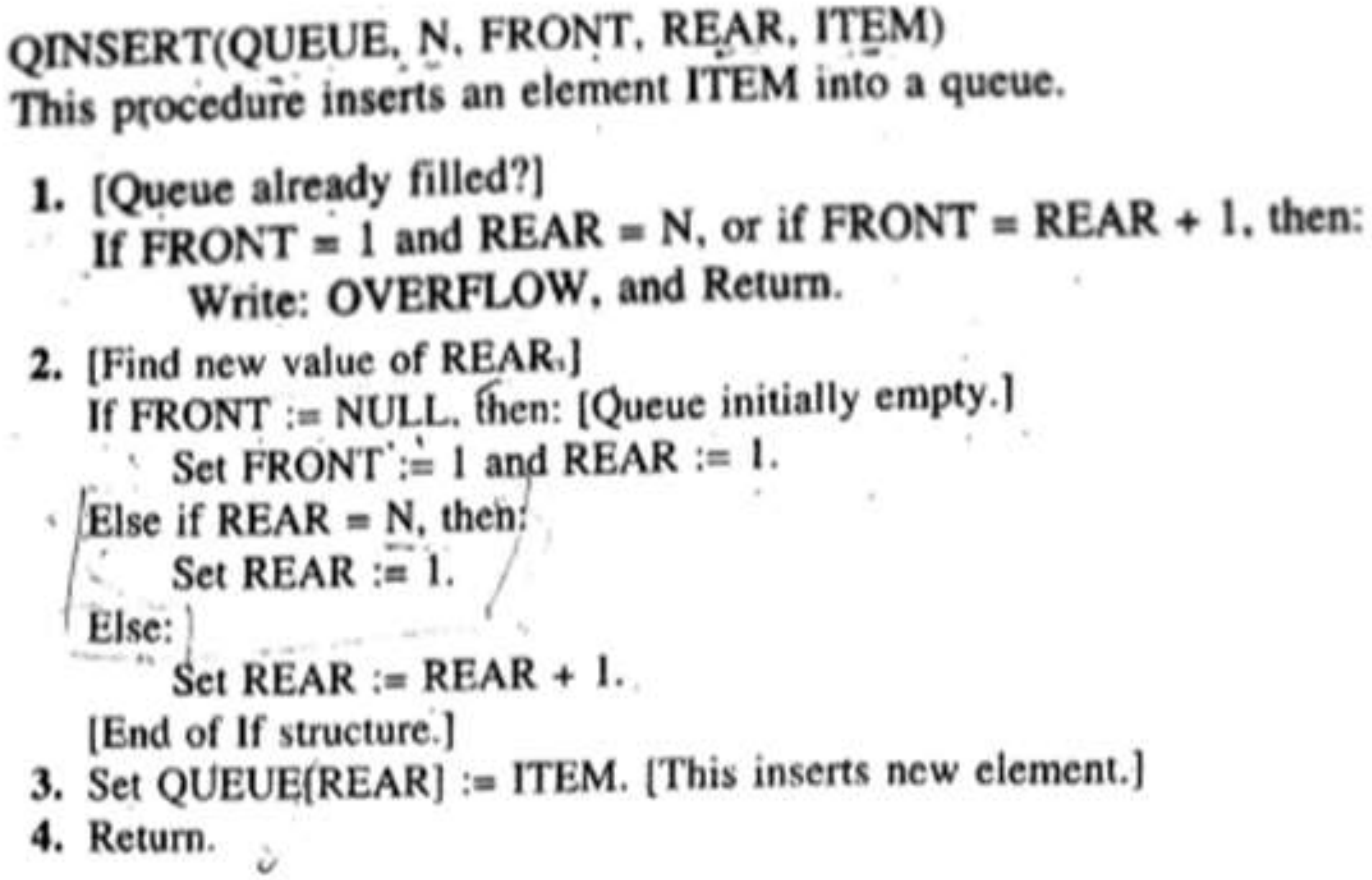
  FRONT = REAR = NULL

Example


QUEUE
(a) Initially empty:
FRONT: 0
REAR: 0
1
2
3
4
5
(b) A, B and then C inserted:
FRONT: 1
REAR: 3
A
B
C
(c) A deleted:
FRONT: 2
REAR: 3
B
C
(d) D and then E inserted:
FRONT: 2
REAR: 5
B
C
D
E
(e) B and C deleted:
FRONT: 4
REAR: 5
D
E
(f) F inserted:
FRONT: 4
REAR: 1
F
D
E

(f) F inserted:	FRONT: 4 REAR: 1	F			D	E
(g) D deleted:	FRONT: 5 REAR: 1	F				E
(h) G and then H inserted:	FRONT: 5 REAR: 3	F	G	H		E
(i) E deleted:	FRONT: 1 REAR: 3	F	G	H		
(j) F deleted:	FRONT: 2 REAR: 3		G	H		
(k) K inserted:	FRONT: 2 REAR: 4		G	H	K	
(l) G and H deleted:	FRONT: 4 REAR: 4				K	
(m) K deleted, QUEUE empty:	FRONT: 0 REAR: 0					

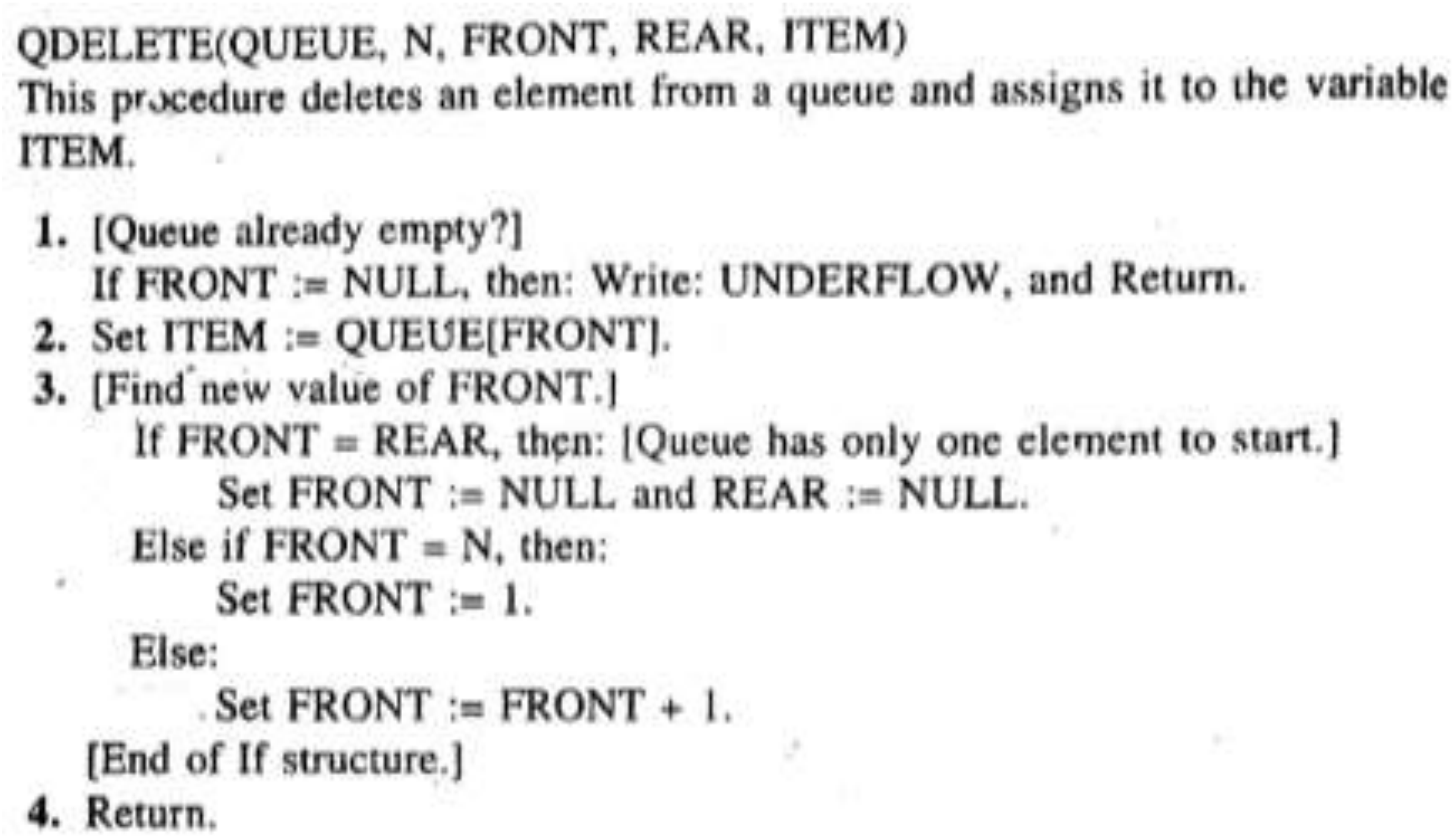
Algorithm: Inserting an Element



QINSERT(QUEUE, N, FRONT, REAR, ITEM)
This procedure inserts an element ITEM into a queue.

```
1. [Queue already filled?]
   If FRONT = 1 and REAR = N, or if FRONT = REAR + 1, then:
         Write: OVERFLOW, and Return.
2. [Find new value of REAR.]
   If FRONT := NULL, then: [Queue initially empty.]
         Set FRONT := 1 and REAR := 1.
   Else if REAR = N, then:
         Set REAR := 1.
   Else:
         Set REAR := REAR + 1.
   [End of If structure.]
3. Set QUEUE[REAR] := ITEM. [This inserts new element.]
4. Return.
```

Algorithm: Deleting an Element

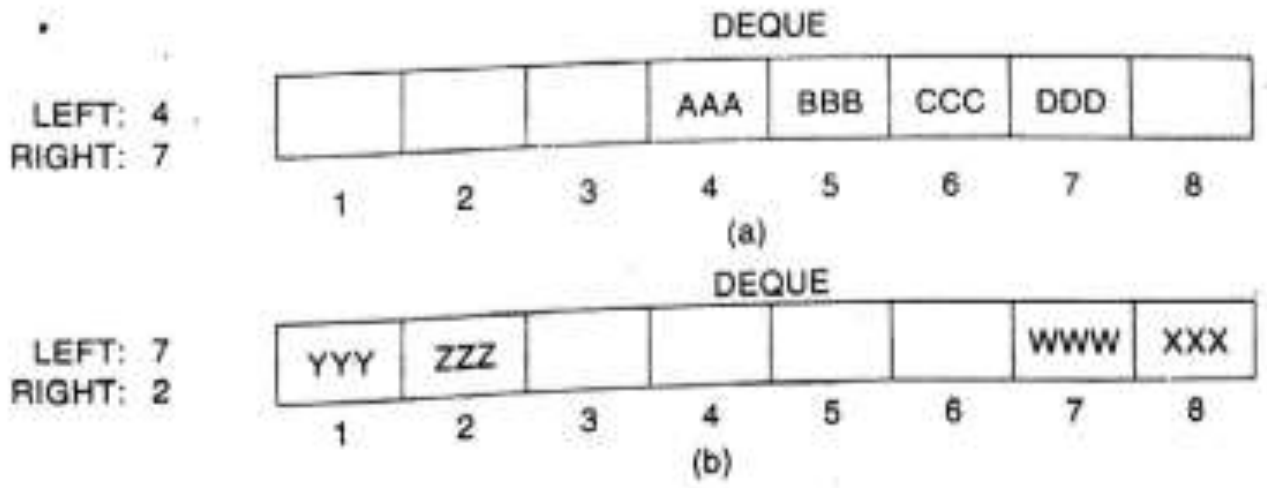


QDELETE(QUEUE, N, FRONT, REAR, ITEM)
This procedure deletes an element from a queue and assigns it to the variable ITEM.

```
1. [Queue already empty?]
   If FRONT := NULL, then: Write: UNDERFLOW, and Return.
2. Set ITEM := QUEUE[FRONT].
3. [Find new value of FRONT.]
     If FRONT = REAR, then: [Queue has only one element to start.]
           Set FRONT := NULL and REAR := NULL.
     Else if FRONT = N, then:
           Set FRONT := 1.
     Else:
           Set FRONT := FRONT + 1.
   [End of If structure.]
4. Return.
```

DEQUES

- A deque (or dequeue) is a linear list in which elements can be added or removed at either end but not in the middle.
- Deque means **D**ouble-**E**nded-**Que**ue



Variations of DEQUE

- There are two types of deque:
  - Input restricted deque
    - Allows insertions only at one end of the list but allows deletions at both the ends of the list.
  - Output restricted deque
    - Allows deletions at only one end of the list but allows insertions at both the ends of the list

Priority Queue

- A Priority queue is a collection of elements such that each element has been assigned a priority.
- The order of deletion or processing of elements depends upon the following:
  - Elements of higher priority are processed before lower priority elements
  - If elements are having same priority, they will be processed in the order in which they were added to queue.

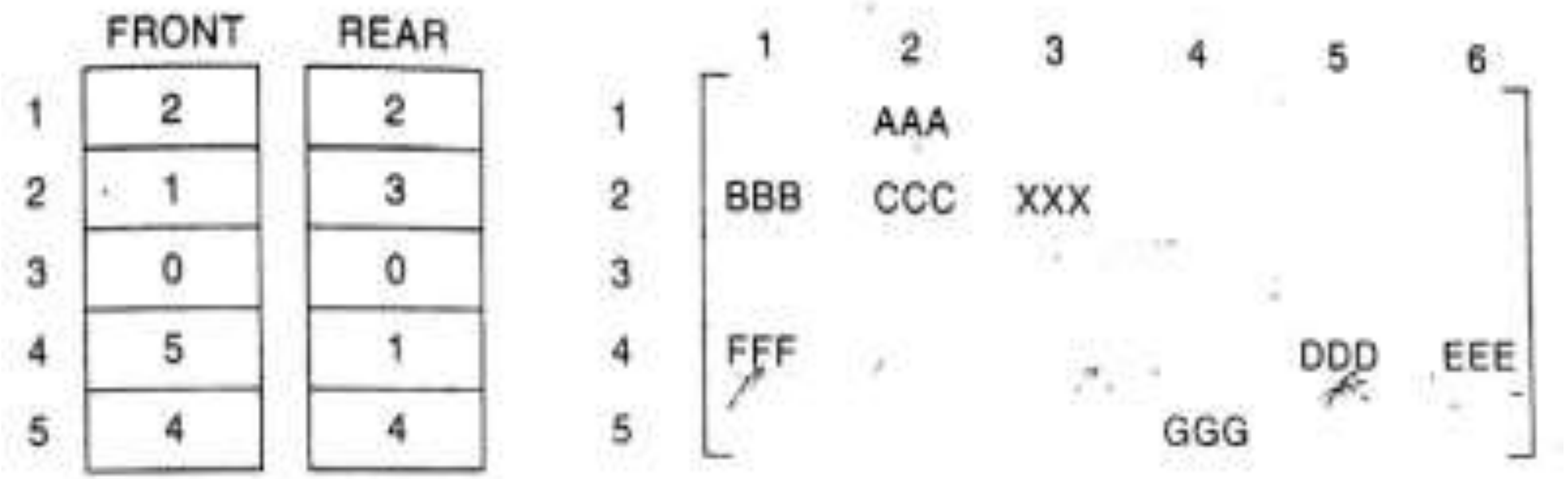
Representation of Priority Queue in memory

- Various ways of maintaining priority queue in memory are:
- One-way list representation
- Array representation

Array representation of priority queue

- A separate queue is used for each level of priority.
- Each such queue will behave as circular queue.
- Each queue has its own pair of pointers FRONT and REAR.
- Each queue is allocated same amount of space.
- A 2-D array QUEUE can be used instead of linear arrays.

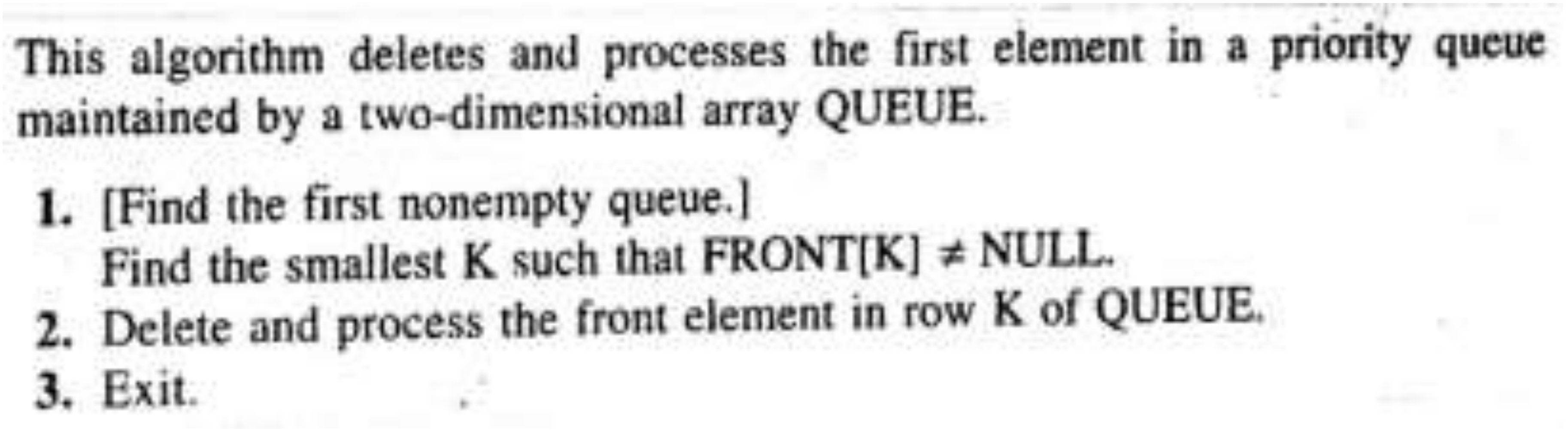
Array representation of priority queue



	FRONT	REAR
1	2	2
2	1	3
3	0	0
4	5	1
5	4	4

$$\begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & & \text{AAA} & & & & \\ 2 & \text{BBB} & \text{CCC} & \text{XXX} & & & \\ 3 & & & & & & \\ 4 & \text{FFF} & & & & \text{DDD} & \text{EEE} \\ 5 & & & & \text{GGG} & & \end{array}$$

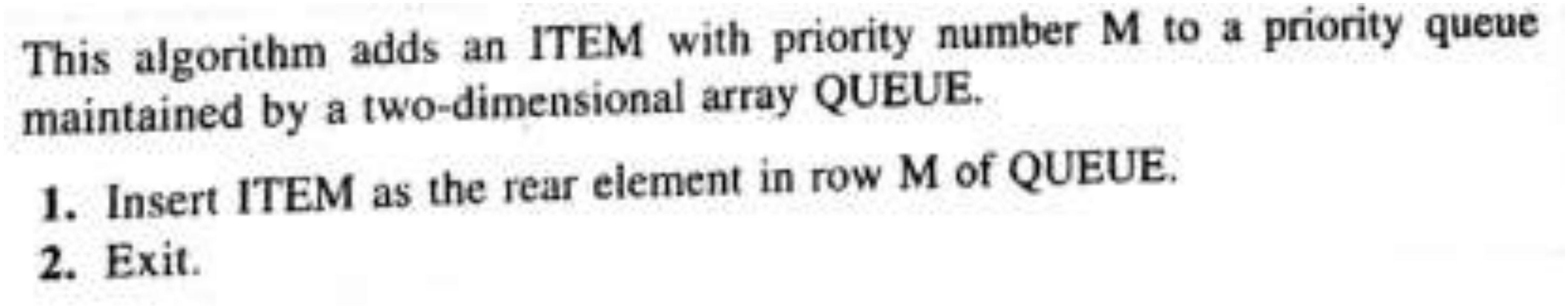
Algorithm to delete element from Queue



This algorithm deletes and processes the first element in a priority queue maintained by a two-dimensional array QUEUE.

```
1. [Find the first nonempty queue.]
   Find the smallest K such that FRONT[K] ≠ NULL.
2. Delete and process the front element in row K of QUEUE.
3. Exit.
```

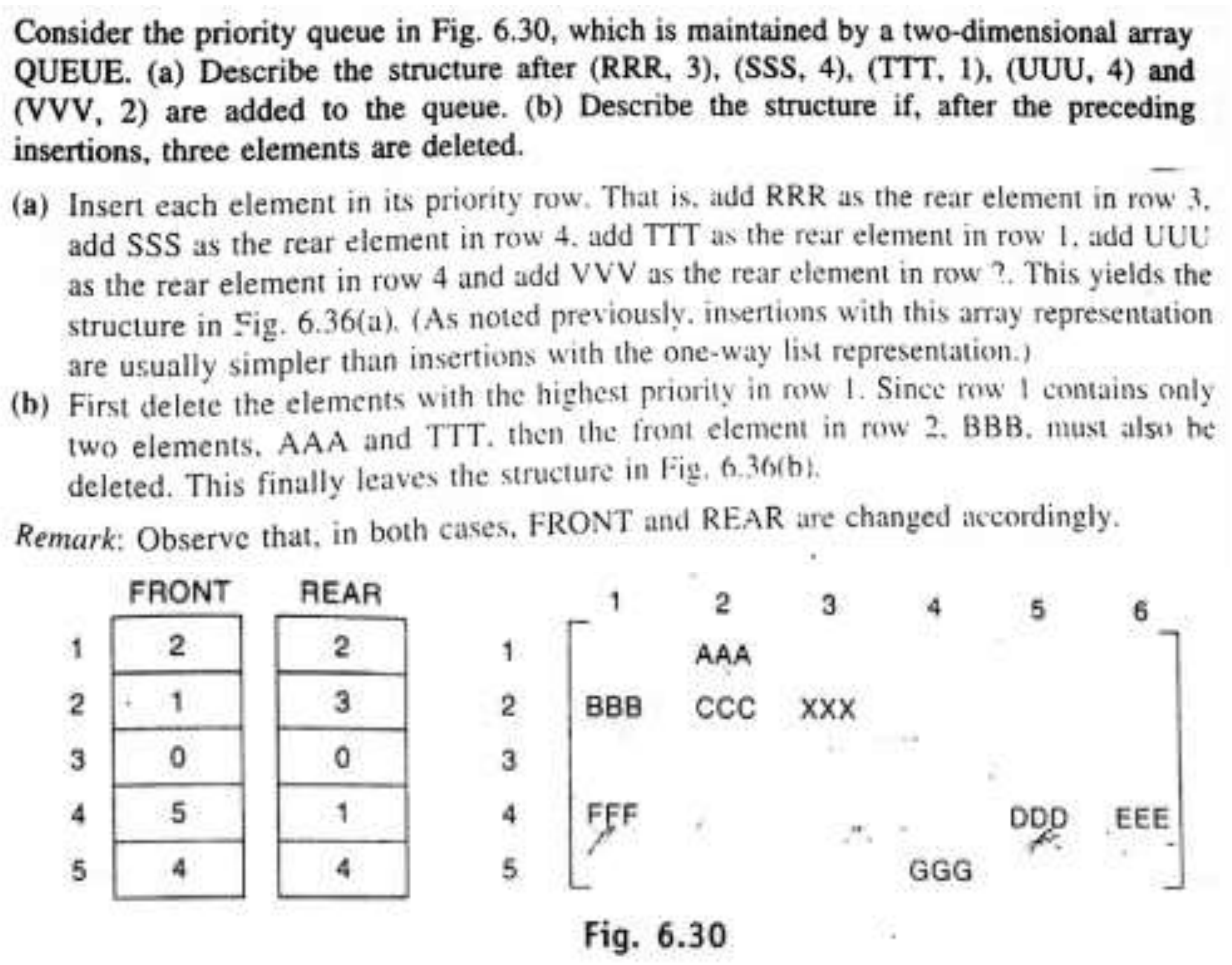
Algorithm to add element into Queue



This algorithm adds an ITEM with priority number M to a priority queue maintained by a two-dimensional array QUEUE.

1. Insert ITEM as the rear element in row M of QUEUE.
2. Exit.

Example



Consider the priority queue in Fig. 6.30, which is maintained by a two-dimensional array QUEUE. (a) Describe the structure after (RRR, 3), (SSS, 4), (TTT, 1), (UUU, 4) and (VVV, 2) are added to the queue. (b) Describe the structure if, after the preceding insertions, three elements are deleted.

(a) Insert each element in its priority row. That is, add RRR as the rear element in row 3, add SSS as the rear element in row 4, add TTT as the rear element in row 1, add UUU as the rear element in row 4 and add VVV as the rear element in row 2. This yields the structure in Fig. 6.36(a). (As noted previously, insertions with this array representation are usually simpler than insertions with the one-way list representation.)

(b) First delete the elements with the highest priority in row 1. Since row 1 contains only two elements, AAA and TTT, then the front element in row 2, BBB, must also be deleted. This finally leaves the structure in Fig. 6.36(b).

Remark: Observe that, in both cases, FRONT and REAR are changed accordingly.

	FRONT	REAR
1	2	2
2	1	3
3	0	0
4	5	1
5	4	4

$$\begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & & \text{AAA} & & & & \\ 2 & \text{BBB} & \text{CCC} & \text{XXX} & & & \\ 3 & & & & & & \\ 4 & \text{FFF} & & & & \text{DDD} & \text{EEE} \\ 5 & & & & \text{GGG} & & \end{array}$$

Fig. 6.30

	FRONT	REAR
1	2	3
2	1	4
3	1	1
4	5	3
5	4	4

QUEUE

$$\begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & & \text{AAA} & \text{TTT} & & & \\ 2 & \text{BBB} & \text{CCC} & \text{XXX} & \text{VVV} & & \\ 3 & \text{RRR} & & & & & \\ 4 & \text{FFF} & \text{SSS} & \text{UUU} & & \text{DDD} & \text{EEE} \\ 5 & & & & \text{GGG} & & \end{array}$$

(a)

	FRONT	REAR
1	0	0
2	2	4
3	1	1
4	5	3
5	4	4

QUEUE

$$\begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & & & & & & \\ 2 & & \text{CCC} & \text{XXX} & \text{VVV} & & \\ 3 & \text{RRR} & & & & & \\ 4 & \text{FFF} & \text{SSS} & \text{UUU} & & \text{DDD} & \text{EEE} \\ 5 & & & & \text{GGG} & & \end{array}$$

(b)