

3.1 Consider the following statements:

- First-in-first out types of computations are efficiently supported by STACKS.
 - Implementing LISTS on linked lists is more efficient than implementing LISTS on an array for almost all the basic LIST operations.
 - Implementing QUEUES on a circular array is more efficient than implementing QUEUES on a linear array with two indices.
 - Last-in-first-out type of computations are efficiency supported by QUEUES.
- (a) (i) & (iii) are true (b) (i) & (ii) are true
(c) (iii) & (iv) are true (d) (ii) & (iv) are true

[1996 : 1 M]

3.2 Which of the following is essential for converting an infix expression to the post fix form efficiently?

- (a) An operator stack
(b) An operand stack
(c) An operand stack and an operator stack
(d) A parse tree

[1997 : 1 M]

3.3 A priority queue Q is used to implement a stack that stores characters. PUSH (C) is implemented INSERT (Q, C, K) where K is an appropriate integer key chosen by the implementation. POP is implemented as DELETEMIN(Q). For a sequence of operations, the keys chosen are in

- non-increasing order
- non-decreasing order
- strictly increasing order
- strictly decreasing order

[1997 : 2 M]

3.4 Compute the postfix equivalent of the following expression:

$$3 * \log(x+1) - \frac{a}{2}$$

[1998 : 2 M]

3.5 What value would the following function return for the input $x = 95$?

```
Function fun (x:integer): integer;
Begin
  If  $x > 100$  then  $\text{fun} = x - 10$ 
  Else  $\text{fun} := \text{fun}(\text{fun}(x + 11))$ 
End;
```

(a) 89 (b) 90
(c) -91 (d) 92

[1998 : 2 M]

3.6 Let S be a stack of size $n \geq 1$. Starting with the empty stack, suppose we push the first n natural numbers in sequence, and then perform n natural operations. Assume that Push and Pop operation take X seconds each, and Y seconds elapse between the end of one such stack operation and the start of the next operation. For $m \geq 1$, define the stack-life of m is the time elapsed from the end of push(m) to the start of the pop operation that removes m from S. The average stack-life of an element of this stack is

- (a) $n(X + Y)$
(b) $3Y + 2X$
(c) $n(X + Y) - X$
(d) $Y + 2X$

[2003 : 2 M]

3.7 A single array A[1...MAXSIZE] is used to implement two stacks. The two stacks grow from opposite ends of the array. Variables top1 and top2 (top1 < top2) point to the location of the topmost element in each of the stacks. If the space is to be used efficiently, the condition for "stack full" is

- (a) (top1 = MAXSIZE/2) and (top2 = MAXSIZE/2 + 1)
(b) top1 + top2 = MAXSIZE
(c) (top1 = MAXSIZE/2) or (top2 = MAXSIZE/2)
(d) top1 = top2 - 1

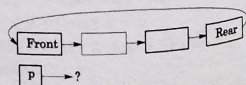
[2004 : 1 M]

3.8 The best data structure to check whether an arithmetic expression has balanced parentheses is a

- (a) queue (b) stack
(c) tree (d) list

[2004 : 1 M]

3.9 A circularly linked list is used to represent a Queue. A single variable p is used to access the Queue. To which node should p point such that both the operations enqueue and dequeue can be performed in constant time?



- (a) rear node
(b) front node
(c) not possible with a single pointer
(d) node next to front

[2004 : 2 M]

3.10 Assume that the operators +, -, * are left associative and ^ is right associative. The order of precedence (from highest to lowest) is ^, *, +, -. The postfix expression corresponding to the infix expression $a + b \times c - d \wedge e \wedge f$ is

(a) $abc \times + def \wedge \wedge -$
(b) $abc \times + de \wedge f \wedge$
(c) $ab \times c \times d - e \wedge f \wedge$
(d) $- + a \times bc \wedge \wedge def$

[2004 : 2 M]

3.11 A program attempts to generate as many permutations as possible of the string, 'abcd' by pushing the characters a, b, c, d in the same order onto a stack, but it may pop off the top character at any time. Which one of the following strings CANNOT be generated using this program?

- (a) abcd (b) dcba
(c) cbad (d) cabd

[2004 : 2 M]

3.12 A function f defined on stacks of integers satisfies the following properties. $f(\emptyset) = 0$ and $f(\text{push}(S, i)) = \max(f(S), 0) + i$ for all stacks S and integers i. If a stack S contains the integers 2, -3, 2, -1, 2 in order from bottom to top, what is $f(S)$?

- (a) 6 (b) 4
(c) -3 (d) 2

[2005 : 1 M]

3.13 An implementation of a queue Q, using two stacks S1 and S2, is given below

```
void insert(Q, x)
{
    push(S1, x);
}

void delete(Q, x)
{
    if (stack-empty(S2)) then
        if (stack-empty(S1)) then
            print("Q is empty");
            return;
        }
    else while (!stack-empty(S1))
    {
        x = pop(S1);
        push(S2, x);
    }
}
```

```
}
x = pop(S2);
}
```

Let n insert and m ($\leq n$) delete operations be performed in an arbitrary order on an empty queue Q. Let x and y be the number of push and pop operations performed respectively in the processes. Which one of the following is true for all m and n ?

- (a) $n + m \leq x \leq 2n$ and $2m \leq y \leq n + m$
(b) $n + m \leq x < 2n$ and $2m \leq y \leq 2n$
(c) $2m \leq x < 2n$ and $2m \leq y \leq n + m$
(d) $2m \leq x < 2n$ and $2m \leq y \leq 2n$

[2006 : 2 M]

3.14 The following postfix expression with single digit operands in evaluated using a stack

$$8 \ 2 \ 3 \wedge / 2 \ 3^* + 5 \ 1^* -$$

Note that \wedge is the exponentiation operator. The top two elements of the stack after the first * is evaluated are

- (a) 6, 1 (b) 5, 7
(c) 3, 2 (d) 1, 5

[2007 : 2 M]

3.15 Suppose you are given an implementation of a queue of integers. The operations that can be performed on the queue are:

- isEmpty(Q) : returns true if the queue is empty, false otherwise.
- delete(Q) : deletes the element at the front of the queue and returns its value.
- insert(Q, i) : inserts the integer i at the rear of the queue.

Consider the following function:

```
void f(queue Q)
{
    int i;
    if (!isEmpty(Q))
    {
        i = delete(Q);
        f(Q);
        insert(Q, i);
    }
}
```

What operation is performed by the above function f?

- (a) Leaves the queue Q unchanged
(b) Reverses the order of the elements in the queue Q
(c) Deletes the element at the front of the queue Q and inserts it at the rear keeping the other elements in the same order
(d) Empties the queue Q

[2007 : 2 M]

- 3.16** Suppose a circular queue of capacity $(n - 1)$ elements is implemented with an array of n elements. Assume that the insertion and deletion operations are carried out using REAR and FRONT as array index variables, respectively. Initially, REAR = FRONT = 0. The conditions to detect queue full and queue empty are
- full : $(\text{REAR} + 1) \bmod n == \text{FRONT}$
empty : REAR == FRONT
 - full : $(\text{REAR} + 1) \bmod n == \text{FRONT}$
empty : $(\text{FRONT} + 1) \bmod n == \text{REAR}$
 - full : REAR == FRONT
empty : $(\text{REAR} + 1) \bmod n == \text{FRONT}$
 - full : $(\text{FRONT} + 1) \bmod n == \text{REAR}$
empty : REAR == FRONT

[2012 : 2 M]

- 3.17** Consider the following operation along with Enqueue and Dequeue operations on queues, where k is a global parameter.

```
Multi-Dequeue(Q) {
    m = k;
    while (Q is not empty) and (m > 0) {
        Dequeue(Q);
        m = m - 1;
    }
}
```

What is the worst case time complexity of a sequence of n queue operations on an initially empty queue?

- $\Theta(n)$
- $\Theta(n + k)$
- $\Theta(nk)$
- $\Theta(n^2)$

[2013 : 2 M]

- 3.18** Suppose a stack implementation supports an instruction REVERSE, which reverses the order of elements on the stack, in addition to the PUSH and POP instructions. Which one of the following statements is TRUE with respect to this modified stack?

- A queue cannot be implemented using this stack.
- A queue can be implemented where ENQUEUE takes a single instruction and DEQUEUE takes a sequence of two instructions.

☒ A queue can be implemented where ENQUEUE takes a sequence of three instructions and DEQUEUE takes a single instruction.

- A queue can be implemented where both ENQUEUE and DEQUEUE take a single instruction each.

[2014 (Set-2) : 2 M]

- 3.19** Consider the C program below:

```
#include <stdio.h>
int *A, stkTop;
int stkFunc(int opcode, int val)
{
    static int size=0, stkTop=0;
    switch(opcode)
    {
        case -1: size = val; break;
        case 0: if (stkTop < size)
                A[stkTop++] = val; break;
        default: if (stkTop)
                return A[--stkTop];
    }
    return -1;
}
```

```
int main()
{
    int B[20]; A=B; stkTop=-1;
    stkFunc(-1, 10);
    stkFunc(0, 5);
    stkFunc(0, 10);
    printf("%d\n", stkFunc(1, 0)+stkFunc(1, 0));
}
```

The value printed by the above program is 15.

[2015 (Set-2) : 2 M]

- 3.20** The result evaluating the postfix expression $10 \ 5 \ 60 \ 6 \ / \ * \ 8$ is

- 284
- 213
- 142
- 71

[2015 (Set-3) : 1 M]

- 3.21** A queue is implemented using an array such that ENQUEUE and DEQUEUE operations are performed efficiently. Which one of the following statements is CORRECT (n refers to the number of items in the queue)?

- Both operations can be performed in $O(1)$ time
- At most one operation can be performed in $O(1)$ time but the worst case time for the other operation will be $\Omega(n)$
- The worst case time complexity for both operations will be $\Omega(n)$
- Worst case time complexity for both operations will be $\Omega(\log n)$

[2016 (Set-1) : 1 M]

- 3.22** Let Q denote a queue containing sixteen numbers and S be an empty stack. $\text{Head}(Q)$ returns the element at the head of the queue Q without removing it from Q . Similarly $\text{Top}(S)$ returns the element at the top of S without removing it from S . Consider the algorithm given below.

```
while Q is not Empty do
if S is Empty OR Top(S) ≤ Head(Q) then
    x := Dequeue(Q);
    Push(S, x);
else
    x := Pop(S);
    Enqueue(Q, x);
end
```

The maximum possible number of iterations of the while loop in the algorithm is 256.

[2016 (Set-1) : 2 M]

- 3.23** Consider the following New-order strategy for traversing a binary tree:

- Visit the root;
- Visit the right subtree using New-order;
- Visit the left subtree using New-order;

The New-order traversal of the expression tree corresponding to the reverse polish expression $3 \ 4 \ * \ 5 \ - \ 2 \ ^ \ 6 \ 7 \ * \ 1 \ + \ -$ is given by

- $+ \ - \ 1 \ 6 \ 7 \ * \ 2 \ ^ \ 5 \ - \ 3 \ 4 \ *$
- $- \ + \ 1 \ * \ 6 \ 7 \ ^ \ 2 \ ^ \ 5 \ * \ 3 \ 4$
- $4 \ - \ + \ 1 \ * \ 7 \ 6 \ ^ \ 2 \ - \ 5 \ * \ 4 \ 3$
- $1 \ 7 \ 6 \ * \ + \ 2 \ 5 \ 4 \ 3 \ * \ - \ -$

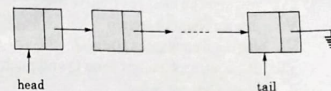
[2016 (Set-2) : 2 M]

- 3.24** A circular queue has been implemented using a singly linked list where each node consists of a value and a single pointer pointing to the next node. We maintain exactly two external pointers FRONT and REAR pointing to the front node and the rear node of the queue, respectively. Which of the following statements is/are CORRECT for such a circular queue, so that insertion and deletion operations can be performed in $O(1)$ time?

- Next pointer of front node points to the rear node.
 - Next pointer of rear node points to the front node.
- I only
 - II only
 - Both I and II
 - Neither I nor II

[2017 (Set-2) : 1 M]

- 3.25** A queue is implemented using a non-circular singly linked list. The queue has a head pointer and tail pointer, as shown in the figure. Let n denote number of nodes in the queue. Let enqueue be implemented by inserting a new node at the head and dequeue be implemented by deletion of a node from the tail.



Which one of the following is the time complexity of the most time-efficient implementation of enqueue and dequeue, respectively, for this data structure?

- $\Theta(1), \Theta(1)$
- $\Theta(1), \Theta(n)$
- $\Theta(n), \Theta(1)$
- $\Theta(n), \Theta(n)$

[2018 : 1 M]

- 3.26** Consider the following sequence of operations on an empty stack.

push(54); push(52); pop(); push(55);

push(62); s = pop();

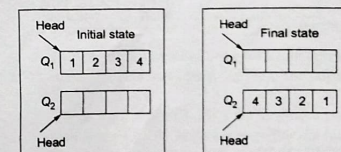
Consider the following sequence of operations on an empty queue.

enqueue(21); enqueue(24); dequeue(); enqueue(28); enqueue(32); q = dequeue();

The value of $s + q$ is 86.

[2021 (Set-1) : 1 M]

- 3.27** Consider the queues Q_1 containing four elements and Q_2 containing none (shown as the Initial State in the figure). The only operations allowed on these two queues are ENQUEUE(Q , element) and DEQUEUE(Q). The minimum number of ENQUEUE operations on Q_1 required to place the elements of Q_1 in Q_2 in reverse order (shown as the Final State in the figure) without using any additional storage is 0.



[2022 : 2 M]

- 3.28** Consider a sequence a of elements $a_0 = 1, a_1 = 5, a_2 = 7, a_3 = 8, a_4 = 9, a_5 = 2$. The following operations are performed on a stack S and a queue Q , both of which are initially empty.

- push the elements of a from a_0 to a_5 in that order into S .
- enqueue the elements of a from a_0 to a_5 in that order into Q .

- III: pop an element from S.
 IV: dequeue an element from Q.
 V: pop an element from S.
 VI: dequeue an element from Q.
 VII: dequeue an element from Q and push the same element into S.
 VIII: Repeat operation VII three times.
 IX: pop an element from S.
 X: pop an element from S.
 The top element of S after executing the above operations is S.

[2023 : 2 M]

- 3.29** Let S1 and S2 be two stacks. S1 has capacity of 4 elements. S2 has capacity of 2 elements. S1 already has 4 elements: 100, 200, 300, and 400, whereas S2 is empty, as shown below.

400 (Top)	
300	
200	
100	
Stack S1	Stack S2

Only the following three operations are available: PushToS2: Pop the top element from S1 and push it on S2.

PushToS1: Pop the top element from S2 and push it on S1.

GenerateOutput: Pop the top element from S1 and output it to the user.

Note that the pop operation is not allowed on an empty stack and the push operation is not allowed on a full stack.

Which of the following output sequences can be generated by using the above operations?

- (a) 400, 200, 100, 300
 (b) 200, 300, 400, 100
 (c) 100, 200, 400, 300
 (d) 300, 200, 400, 100

[2024 (Set-2) : 2 M]

- 3.30** Consider a stack data structure into which we can PUSH and POP records. Assume that each record pushed in the stack has a positive integer key and that all keys are distinct.

We wish to augment the stack data structure with an $O(1)$ time MIN operation that returns a pointer to the record with smallest key present in the stack

- without deleting the corresponding record, and
- without increasing the complexities of the standard stack operations.

Which one or more of the following approach(es) can achieve it?

- (a) Keep with every record in the stack, a pointer to the record with the smallest key below it.
 (b) Keep a pointer to the record with the smallest key in the stack.
 (c) Keep an auxiliary array in which the key values of the records in the stack are maintained in sorted order.
 (d) Keep a Min-Heap in which the key values of the records in the stack are maintained.

[2025 (Set-2) : 2 M]

Answers Stacks and Queues

- 3.1** (a) **3.2** (a) **3.3** (d) **3.4** Sol. **3.5** (c) **3.6** (b) **3.7** (d) **3.8** (b) **3.9** (c)
3.10 (a) **3.11** (d) **3.12** (c) **3.13** (a) **3.14** (a) **3.15** (b) **3.16** (a) **3.17** (a) **3.18** (c)
3.19 (15) **3.20** (c) **3.21** (a) **3.22** (256) **3.23** (c) **3.24** (b) **3.25** (b) **3.26** (86) **3.27** (0)
3.28 (8) **3.29** (a, b, d) **3.30** (a)

Explanations Stacks and Queues

3.1 (a)

Stack is LIFO system so FIFO not supported by stack hence option statement 1 is incorrect and queue is using FIFO so statement 4 is also wrong.

3.2 (a)

Operands never change its position during expression conversion so only operator stack is needed.

3.3 (d)

In stack last element will be deleted first but according to question, it should be work like queue. So last element deleted at last, for that we give lower priority to last element. So all elements should be in strictly decreasing order.

3.4 Sol.

$$3 * \log(x+1) - \frac{a}{2}$$

Step by step conversion to postfix

$$= 3 * (x+1) \log - a/2 = 3 * x1 + \log - a/2 \\ = 3x1 + \log * - a/2 = 3x1 + \log * a/2 -$$

3.5 (c)

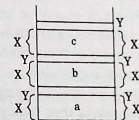
```

fun(95)
└─ fun(fun(106))
   └─ fun(96)
      └─ fun(fun(107))
         └─ fun(97)
            └─ fun(fun(108))
               └─ fun(98)
                  └─ fun(fun(fun(109)))
                     └─ fun(99)
                        └─ fun(fun(fun(200)))
                           └─ fun(100)
                              └─ fun(fun(fun(111)))
                                 └─ fun(101)
                                    └─ fun = 101 - 10 = 91
  
```

So the return value is 91.

3.6 (b)

Let we have $n = 3$ sized stack:



So, lifetime of c : Y

$$\text{lifetime of b : } Y + X + Y + X + Y \\ = 3Y + 2X$$

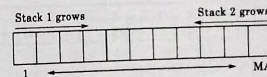
$$\text{lifetime of a : } Y + X + Y + X + Y + X + Y + X + Y \\ = 5Y + 4X$$

So, average lifetime of an element

$$= \frac{Y + 3Y + 2X + 5Y + 4X}{3} = \frac{9Y + 6X}{3} \\ = 3Y + 2X = (3Y + 3X) - X \\ = 3(Y + X) - X$$

So, in general : $n(Y + X) - X$

3.7 (d)



- When stack 1 contain 1 element and stack 2 contain 1 element then $\text{top1} = 1$ and $\text{top2} = \text{MAXSIZE} - 1$. Here $\text{top1} + \text{top2} = \text{MAXSIZE}$. Since both stack are not full. Hence no overflow. So false
- $(\text{top1} = \text{MAXSIZE}/2)$ and $(\text{top2} = \text{MAXSIZE}/2 + 1)$ will true when both stack has $n/2$ element but space is not effectively used i.e. when stack 1 has no element and stack 2 has $n - 1$ elements then it will not detect overflow. So false
- $\text{top1} = \text{top2} - 1$ is true, since it cover all cases when both stack are with full capacity, stack 1 is empty and stack 2 is full and vice-versa. Hence true

The stack is a data structure in which each push (insert) or pop (delete) operation takes constant $\theta(1)$ time. The best data structure to check whether an arithmetic expression has a balanced parentheses.

Step 1: Start with empty stack of currently open parentheses.

Step 2: Process each char of an expression string.

Step 3: If it is open, push it on the stack.

Step 4: If it is closed, check it against the top of stack element.

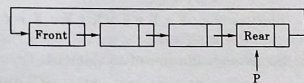
Step 4.1: If stack empty or not a matching parentheses then not balanced and return false.

Step 4.2: Otherwise it matches. pop the open parentheses then goto step 2.

Step 5: If stack is empty at the end, return true.

Step 6: If not empty then some parentheses is unmatched, return false.

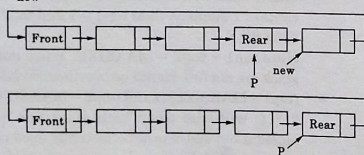
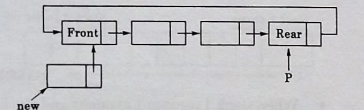
3.9 (c)



To add a new node:



new \rightarrow next = Rear \rightarrow next;
Rear \rightarrow next = new;
Rear = new;



It will take constant time.

Similarly delete node from Front code will be:

Front = Rear \rightarrow next;

Rear \rightarrow next = Front \rightarrow next;

free (Front);

It will also take constant time.

3.10 (a)

According to conditions given:

$$\Rightarrow [(a + (b \times c)) - (d \wedge (e \wedge f))]$$

$$\Rightarrow [(a + (b \times c)) - (d \wedge (e \wedge f))]$$

$$\Rightarrow [(abc \times +) - (def \wedge)] \Rightarrow abc \times + def \wedge$$

3.11 (d)

- For every element push an element and pop immediately i.e. push (a) then pop will give permutation abcd.
- For all elements push (a), push (b), push (c) and push (d) then pop, pop, pop, pop will give us dcba.
- Push (a), push (b), push (c) then pop, pop, pop will give us cba, then push (d) and pop will give us dcba.
- Option (d) is not possible because a cannot pop after c and before 'b'.

3.12 (c)

i : The element to be pushed, S : Stack
Initially f(S) = 0

f(S)	max(f(S), 0)	i	Updated_f(S) = max(f(S), 0) + i
0	0	2	2
2	2	-3	-1
-1	0	2	2
2	2	-1	1
1	1	2	3

3.13 (a)

Number of push operations

$$= n(\text{insert}) + m(\text{delete}) = n + m$$

So, $n + m \leq x$ but there are maximum $2n$ insert operations so $n + m \leq x \leq 2n$... (1)

Number of pop operations = $n + m$

But there are $2m$ delete operations which are less than no. of pop operations, hence

$$2m \leq n + m \quad \dots (2)$$

From (1) and (2): $n + m \leq x \leq 2n$ & $2m \leq n + m$

3.14 (a)

Given postfix expression is WW

$$8 \ 2 \ 3 \wedge / \ 23 \ * \ + \ 5 \ 1 \ *$$

Expression	Op1	Op2	Value	top S(RL)
8				8
2				8, 2
3				8, 2, 3
^	2	3	8	8, 8
/	8	8	1	1, 2
2				1, 2, 3
3				1, 6
*	2	3	6	

So the top two elements of the stack are 6, 1 after the first * is evaluated.

3.15 (b)

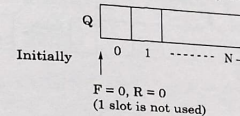
Case-1: When queue is empty then no change would happen on queue.

Case-2: 1st delete all elements one by one from queue then last deleted element enter at 1st position, 2nd last deleted element at 2nd position etc.

So finally we get reverse order of given (input queue).

3.16 (a)

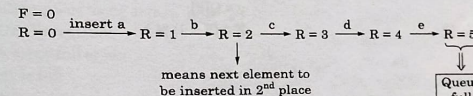
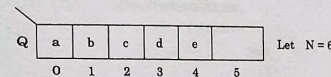
In actual implementation of Q of n-elements, we can store n elements in the array and initially $F = R = -1$. Here, we are implementing Q in $(n - 1)$ slots of n element array and given, initially $F = R = 0$



Initially $R = 0$ means R contains the position of next element to be inserted. (In actual implementation of Q, R contains the position of newly inserted element). (Also in actual implementation of Q, we increment R and then store the element) Here, firstly we will insert the element then R will get incremented, because initially $R = 0$.

i.e. $Q[R] = a \Rightarrow \text{Enqueue}$
 $(R++) \% N$

ex:



Insertion in $R = 5$ not possible because Q's capacity is $(6 - 1)$ i.e. 5 elements

So, queue will be full when:

$$(R + 1) \% 6 = F$$

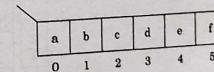
$$\{(5 + 1) \% 6 = 0\}$$

Now, for deleting:

$$y = x[F] \text{ (delete) / store}$$

$$(F++) \% N \text{ (increment)}$$

Return y

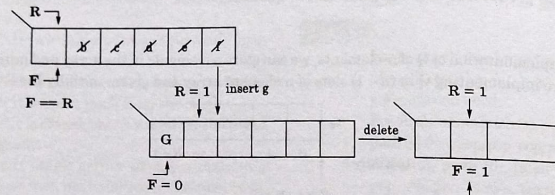


$$R = 5 \xrightarrow{-a} F = 1 \xrightarrow{+f} R = 0$$

$$\{ (R + 1) \% 6 = F \}$$

Delete element and increment F:

$F = 1 \xrightarrow{\text{del}} F = 2 \xrightarrow{\text{del}} F = 3 \xrightarrow{\text{del}} F = 4 \xrightarrow{\text{del}} F = 5 \xrightarrow{\text{del}} F = 0$
(and R is also 0)



So, at anywhere, if $F == R$ means Q is empty.

So, in short initially $R == 0$, means no element present

So store the element then increment R and after insertion if $F == 0$ then delete the element and increment F 1 slot empty as queue capacity is $n - 1$.

Enqueue(x)

```
{
  if (R + 1 % N == F)
  {
    printf("Q is full, overflow");
    exit(1);
  }
  else
  {
    Q[R] = x;
    R = (R + 1) % N;
  }
}
```

Dequeue(x)

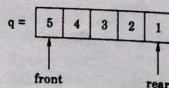
```
{
  if (F == R)
  {
    printf("underflow");
    exit(1);
  }
  else
  {
    y = Q[F];
    F = (F + 1) % N;
  }
}
```

3.17 (a)

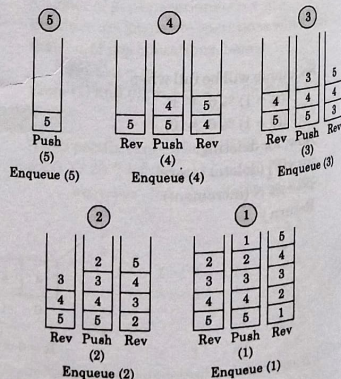
Since there are 3 queue operation i.e., enqueue, dequeue and Multi-dequeue 'n' enqueue operation will take $\Theta(n)$ time, 'n' dequeue operation will take $\Theta(n)$ time, 'n' Multi-dequeue operation will take $\Theta(n)$ time i.e., $\min(n, k)$ but for worst case when $n = k$ time complexity will be $\Theta(n)$.

3.18 (c)

Consider an queue with elements inserted in same order as element present in queue.



Now to implement stack same as queue:



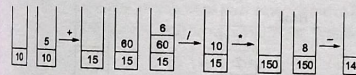
A queue can be implemented where ENQUEUE takes a sequence of three instructions and DEQUEUE takes a single instruction.

3.19 (15)

StkFunc (-1, 10); \Rightarrow size = 10; stkTop = 0
[case -1 executed]
StkFunc (0, 5); \Rightarrow A[0] sets to 5 and
stkTop = 1 [case 0 executed]
StkFunc (0, 10); \Rightarrow A[1] sets to 10 and stkTop = 2 [case 0 executed]
StkFunc (1, 0); \Rightarrow returns 10, stkTop = 1
[default case executed]
StkFunc (1, 0); \Rightarrow returns 5, stkTop = 0
[default case executed]
 \therefore Program prints 10 + 5 = 15

3.20 (c)

10 5 + 60 6 / * 8 -



Result is 142.

3.21 (a)

Implementing queue using array:

ENQUEUE Operation:

Check array full or not
if array is full
stop

else enter the element in the end of array;
which will take $O(1)$ time.

DEQUEUE Operation:

Check array empty or not
if array is empty
stop

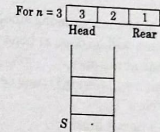
else delete the element from front of array and increment the head value (pointer to the starting element of array).
which will take $O(1)$ time.
So for array implementation of queue, ENQUEUE and DEQUEUE operation takes $O(1)$ time.

3.22 (256)

The minimum number of iterations of the while loop in algorithm when use take queue contain element in ascending order i.e., 1, 2, 3, 4, ..., 16 is 16.

The maximum number of iterations of while loop in algorithm when we take queue containing elements in descending order i.e., 16, 15, 14, ..., 1. First 16 will push into stack and then enqueue it in the end of the queue. This process do till we get 1 as head element. When head point to 1 then simple push the 1 in stack. In this manner we have to push all element in stack in ascending order, until queue is empty it will take 256 of iterations.

Example:



Sequence of operation with while loop execution.

1. dequeue (3)
2. pop (3)
3. dequeue (2)
4. enqueue (3)
5. dequeue (1)
6. enqueue (2)
7. push (1)
8. enqueue (2)
9. dequeue (3)
- push (2)

So for $n = 3$ it takes $3 \times 3 = 9$ iterations of while loop in algorithm.

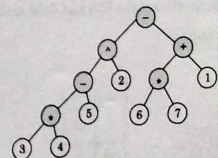
So, for $n = 16$ it will take $16 \times 16 = 256$ iterations of while loop.

3.23 (c)

The expression is given in reverse polish notation i.e., post order.

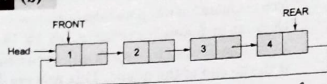
Expression is $3 \ 4 \ 5 \ 2 \ 6 \ 7 \ 1 \ +$

Expression tree for above post order expression is



From the above expression tree NEW ORDER traversal is $- + 1 * 7 6 \wedge 2 - 5 * 4 3$.

3.24 (b)

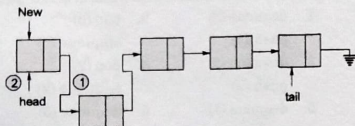


Since insertion in a queue are always from REAR and deletion is always from FRONT. Hence having the next pointer of REAR node pointing to the FRONT node will lead to both insertion and deletion operations in $O(1)$ time.

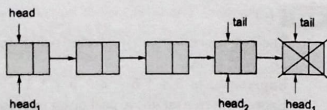
3.25 (b)

Queue is implemented using singly linked list. So, insertion will be done at head and deletion will be done at tail only.

- Insertion will take $\theta(1)$ time i.e. adding new node "New" by
New \rightarrow next = head
head = New

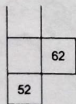


- Deletion will take $\theta(n)$ time i.e. we have to read one node before last node, otherwise we lose tail pointer i.e.
while ($\text{head}_1 \rightarrow \text{next} \neq \text{Null}$)
head₂ = head₁;
head₁ = head₁ \rightarrow next;
head₂ = Null;
free (head₁);
tail = head₂;



So, insertion take $\theta(1)$ and deletion take $\theta(n)$.

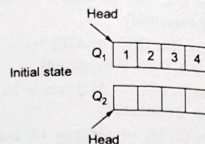
3.26 (86)



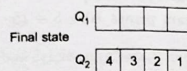
$$S = 62, R = 24$$

$$S + R = 86$$

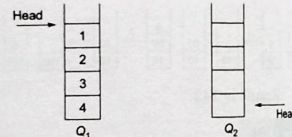
3.27 (0)



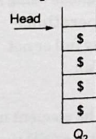
No additional storage.
Enqueues on Q_1 = ?



Consider Q_1 and Q_2 as the following



Now, Enqueue (Q_2 , \$) 4 times,

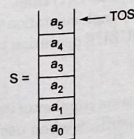


Now, one by one perform

- $x = \text{Dequeue}(Q_1)$
- Enqueue (Q_2 , x) until Q_1 is empty.

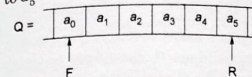
3.28 (8)

- S is stack and follows LIFO order. Thus after pushing a_0 to a_5 , a_5 will come at the top. Top of the stack points to a_5 .

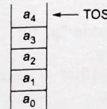


- Q is queue and follows FIFO order. Thus after enqueueing a_0 to a_5 order will persist.

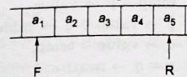
Front will point to a_0 and rear will point to a_5 .



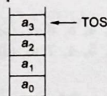
- Popping an element will remove top of the stack S i.e. a_5 will be removed. Now, TOS is a_4 .



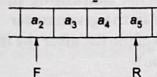
- Dequeuing an element will be from Front of the queue Q i.e. a_0 will be removed. Now, Front will point to a_1 .



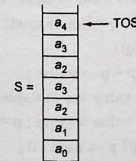
- Popping an element will remove top of the stack S i.e. a_4 will be removed. Now, TOS is a_3 .



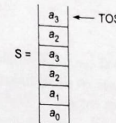
- Dequeuing an element will be from front of the queue Q i.e. a_1 will be removed. Now, front will point to a_2 .



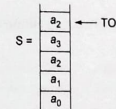
- 3 times dequeuing an element will be from front of the queue Q i.e. a_2 , a_3 , a_4 will be removed and pushed into stack S in the same order. So, TOS will be a_4 .



- Popping an element will remove top of the stack S i.e. a_4 will be removed. Now, TOS is a_3 .



- Popping an element will remove top of the stack S i.e. a_3 will be removed. Now, TOS is a_2 .



Hence, a_2 i.e. 8 will be at the top of the stack.

3.30 (a)

Maintaining a pointer to the smallest key below each element ensures that the MIN operation runs in $O(1)$ time. Push updates this pointer in $O(1)$ time, and pop removes the top element without affecting efficiency. Other approaches require $O(n)$ or $O(\log n)$ updates, making them unsuitable.

■■■■