UNIT-3

Queues:

Definition:

→ Queue is an ordered collection of homogeneous data elements. where the insertion and deletion must be done at different ends of Queue.

→ The insertion is done at rear end of the Queue

→ The deletion is done at front end of the Queue.

→ Queue is defined as collection of one (or) more elements in which the insertion and deletion is performed at different ends of Queue.

→ Some of the Queue examples are:.

1. Queuing in front of a Counter.
2. Traffic control at a turning point.
3. Process Synchronization in multi-user Environment.
4. Resource Sharing in a Computer centre.

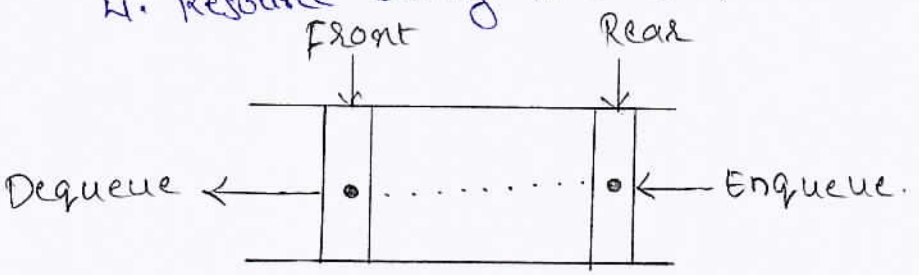


Fig:- Structure of a Queue:.

→ The insertion operation in the Queue is also called as Enqueue.

→ The Deletion operation in the Queue is also called as Dequeue.

→ Queue follows FIFO Principle (or) Structure

→ FIFO can be abrivated as FIRST IN FIRST OUT.

Operations on Queues:

There are two operations that can be performed on Queues.

(1) ENQUEUE (Insertion)

(2) DEQUEUE (Deletion)

REPRESENTATION OF QUEUES:

There are two - ways to represent a Queue in memory:

(1) Queue representation with arrays.

(2) Queue representation with linked list.

(1) Representation of a Queue using an Array:

A one - dimensional array is used to represent a Queue. In this representation two pointers, namely FRONT and REAR are used to indicate the two ends of the Queue.

For Inserting an Element into the Queue the pointer REAR is used.

For Deleting an Element from the Queue the pointer FRONT is used.

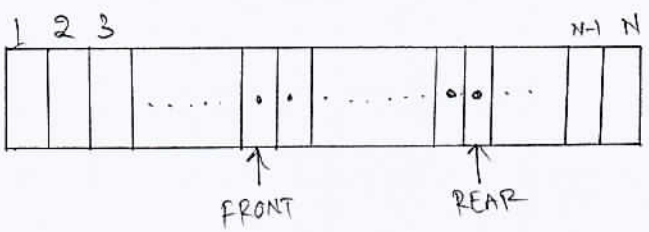


Fig: Array representation of a Queue.

Three States of a Queue with this representation are:

(1) Queue is Empty:

FRONT = 0
REAR = 0

(2) Queue is Full

REAR = N
FRONT = 1

(3) Queue contains Elements ≥ 1

FRONT ≤ REAR

Number of Elements = REAR - FRONT + 1

1. The Insertion operation (ENQUEUE) to insert an Element into a Queue.

Algorithm Enqueue : Array:

Steps:

1. If (REAR = N) then
2. print "Queue is full".
3. Exit
4. Else
5. if (REAR = 0) and (FRONT = 0) then
6. FRONT = 1
7. Endif
8. REAR = REAR+1
9. Q[REAR] = ITEM
10. Endif.
11. STOP

2. The Deletion operation (DEQUEUE) to delete an element.

Algorithm Dequeue : Array:

Steps:

1. if (FRONT = 0) then
2. print "Queue is empty".
3. Exit
4. Else
5. ITEM = Q[FRONT]
6. if (FRONT = REAR)
7. REAR = 0
8. FRONT = 0
9. Else
10. FRONT = FRONT +1
11. Endif
12. Endif
13. Stop.

2) Representation of a Queue using a Linked-List:

In Double-linked list representation of a Queue. The pointers FRONT and REAR point the first node and the last node in the list.

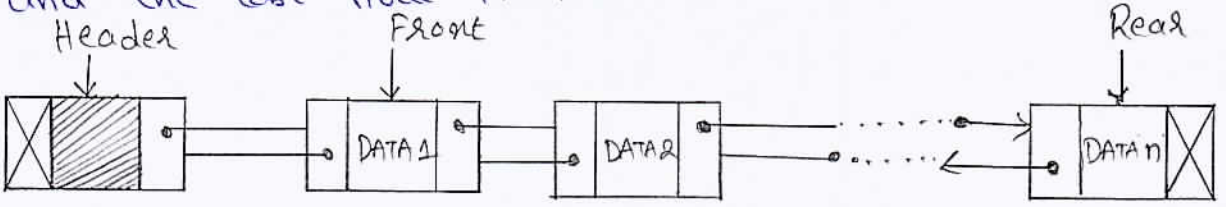


fig:- A double linked list representation of a Queue.

** The two states of the Queue, either Empty or Containing some elements, can be known by the following tests.

Queue is Empty:

FRONT = REAR = HEADER

HEADER → RLINK = NULL

Queue contains at least one element:

HEADER → RLINK ≠ NULL

** NOTE: The Insertion operation in the Queue is same as in the Algorithm InsertEnd_DL for Enqueue.

The Deletion Operation in the Queue is same as in the Algorithm DeleteFront_DL for Dequeue.

VARIOUS QUEUE STRUCTURES:

The various types of Queue Structures are.

1. Circular Queue
2. Deque
3. Priority Queue.

1. Circular Queue:

In a Queue represented using an array. When the REAR pointer reaches the End, Insertion is not possible even if room is available at the front. To avoid this a Circular array (or) Circular Queue is used.

A Circular array is the same as an ordinary array, say A[1.....N]. But it implies that A[1] comes after A[N] (or) after A[N], A[1] appears.

fig:. Logical and physical views of a circular Queue.

NOTE: Both pointers will move in a clockwise direction. This is Controlled by the MOD operation:

The two states of the circular Queue are:.

(1) Circular Queue is Empty:.

FRONT = 0

REAR = 0

(2) Circular Queue is full:.

FRONT = (REAR MOD LENGTH) + 1

Operations on Circular Queue:.

(1) Insertion (Enqueue)

(2) Deletion (Dequeue).

1. Algorithm Enqueue - CQ:.

Steps:.

1. If (FRONT = 0) then
2. FRONT = 1
3. REAR = 1
4. CQ [FRONT] = ITEM
5. Else
6. next = (REAR MOD LENGTH) + 1
7. if (next ≠ FRONT) then
8. REAR = next
9. CQ [REAR] = ITEM
10. Else
11. Print "Queue is full".
12. Endif
13. Endif
14. Stop.

2) Algorithm Dequeue - CQ:.

Steps:.

1. If (FRONT = 0) then
2. Print "Queue is Empty".
3. Exit
4. Else
5. ITEM = CQ[FRONT]
6. if (FRONT = REAR) then
7. FRONT = 0
8. REAR = 0
9. Else
10. FRONT = (FRONT MOD LENGTH + 1)
11. Endif
12. Endif
13. Stop.

DEQUE:

Deque: (1) Another variation of the Queue is known as deque.

(2) In Deque, both Insertion and deletion operations can be done at either end of the structure.

(3) The term Deque has originated from double ended Queue.

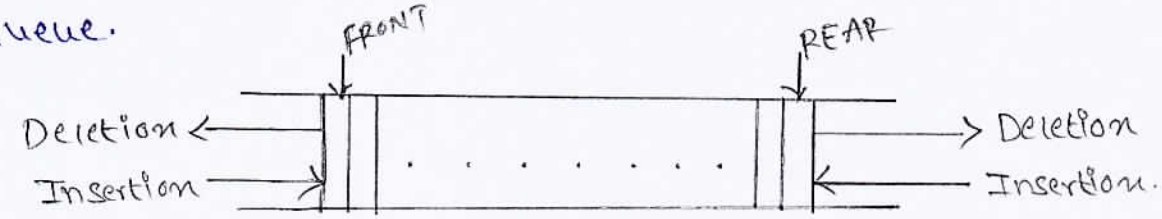


fig: A deque structure.

(4) A deque is a general representation of both Stack and Queue.

Operations on Deque:

The following four operations are possible on a deque.

1. PUSH_DQ(ITEM): To Insert ITEM at the FRONT end of a deque.
2. POP_DQ(): To Remove the FRONT item from a deque.
3. Inject (ITEM): To Insert ITEM at the REAR end of a deque.
4. Eject(): To remove the REAR-ITEM from a deque.

1. Algorithm PUSH_DQ:

Steps:

1. If (FRONT = 1) then
2. ahead = LENGTH
3. else
4. if (FRONT = LENGTH) OR (FRONT = 0) then
5. ahead = 1
6. else
7. ahead = FRONT - 1

8. Endif
9. if (ahead = REAR) then
10. Print "Deque is full".
11. Exit.
12. Else
13. FRONT = ahead.
14. DQ[FRONT] = ITEM.
15. Endif
16. Endif
17. STOP.

Algorithm Eject-DQ:.

Steps:-

1. if (FRONT = 0) then
2. Print "Deque is Empty".
3. Exit
4. Else
5. if (FRONT = REAR) then
6. ITEM = DQ(REAR)
7. FRONT = REAR = 0
8. Else
9. if (REAR = 1) then
10. ITEM = DQ[REAR]
11. REAR = LENGTH
12. Else.
13. if (REAR = LENGTH) then
14. ITEM = DQ(REAR)
15. REAR = 1
16. Else
17. ITEM = DQ(REAR)
18. REAR = REAR - 1
19. Endif
20. Endif
21. Endif
22. Endif
23. Stop.

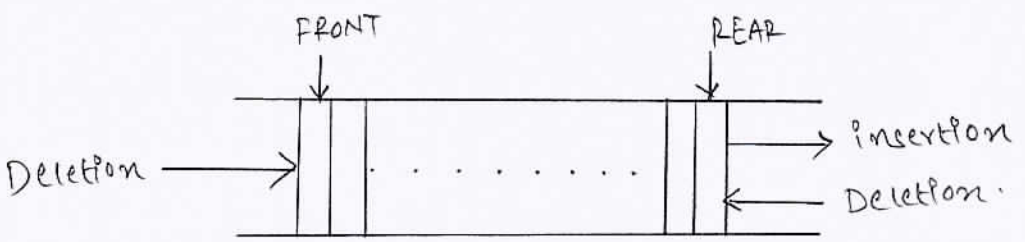
Variations of Deque:

There are two variations of Deque:.

1. Input - restricted deque.
2. Output - restricted deque.

1. Input- Restricted Deque:

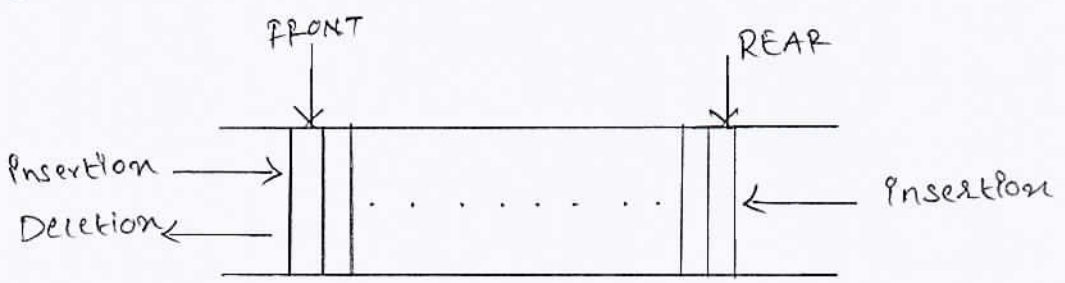
An Input- restricted deque is a deque which allows Insertions at one- end i.e only at REAR end and it allows deletions at both ends.



(1) Input- Restricted - Deque.

2. Output - restricted Deque:.

An Output - restricted - deque is a deque which allows Deletions at one- End i.e only at FRONT end and it allows Insertions at both ends.



(2) Output- Restricted - Deque.

PRIORITY QUEUE:

* A priority queue is another variable of queue structure. Here, each element has been assigned a value, called the priority of the element, and an element can be inserted or deleted not only at the ends but at any position on the queue.

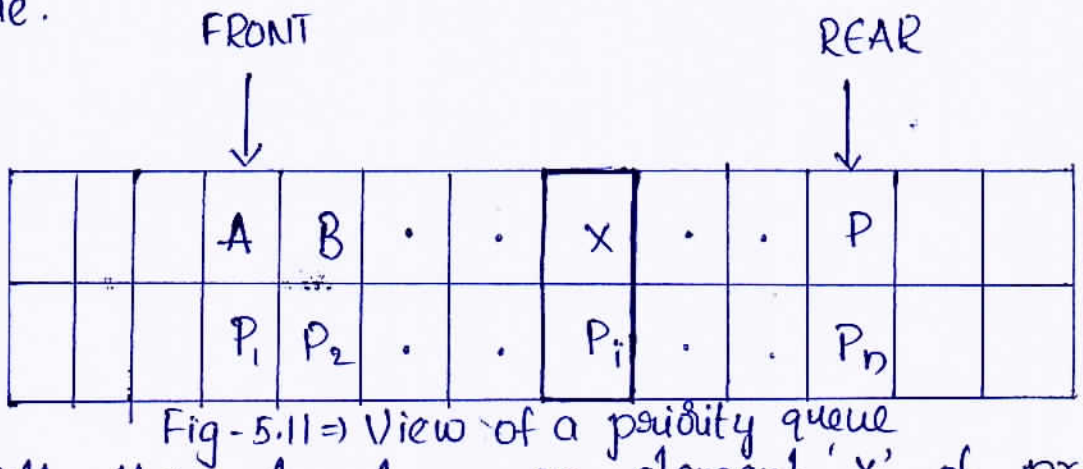


Fig-5.11 ⇒ View of a priority queue

* With this structure, an element 'X' of priority P_i may be deleted before an element which is at FRONT. Similarly, insertion of an element is based on its priority, that is, instead of adding it after the REAR it may be inserted at an intermediate position dictated by its priority value.

* A priority queue does not strictly follow the first-in-first-out (FIFO) principle which is the basic principle of a queue.

* An element of higher priority is processed before any element of lower priority.

* Two elements with the same priority are

processed according to the order in which they were added to the queue.

There are various ways of implementing the structure of a priority queue. They are: 1) using a simple/circuit circular array
2) Multi-queue implementation
3) using a double linked list
4) using help tree

priority queue using an array:

With this representation, an array can be maintained to hold the item and its priority value. The element will be inserted at the REAR end as usual. The deletion operation will then be performed in either of the two following ways:

a) starting from the FRONT pointer, transverse the array for an element of the highest priority. Delete this element from the queue. If this is not the front-most element, shift all its trailing elements after the deleted element one stroke each to fill up the vacant position (see figure-5.12).

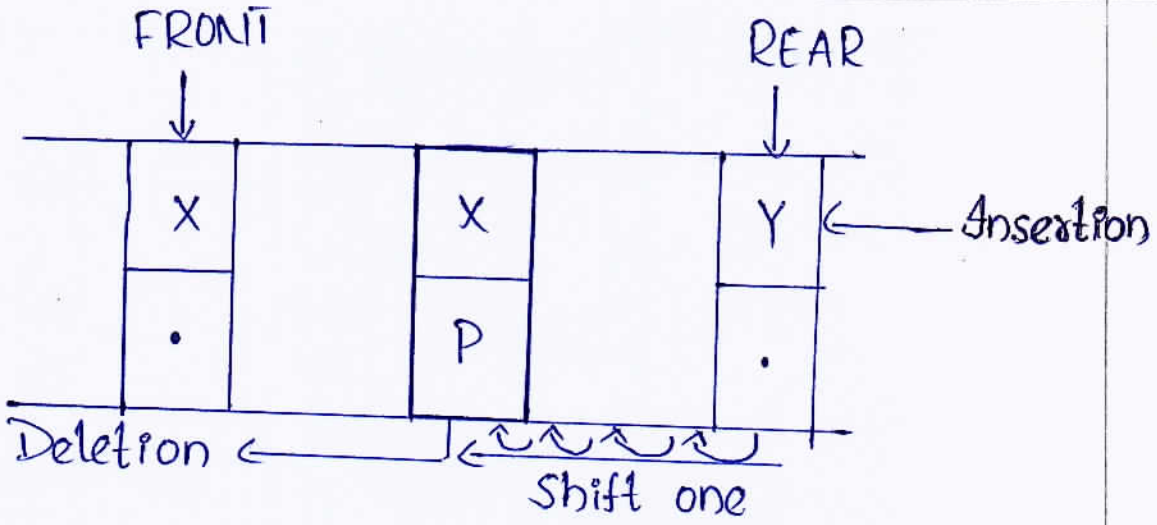


Figure-5.12 :- Deletion operation in an array representation of a priority queue.

This implementation, however, is very inefficient as it involves searching the queue for the highest priority element and shifting the trailing elements after the deletion. A better implementation is as follows:

b)=> Add the elements at the REAR end as earlier. using a stable sorting algorithm*, sort the elements of the queue so that the highest priority element is at the front end. When the deletion is required, delete it from the FRONT end only.

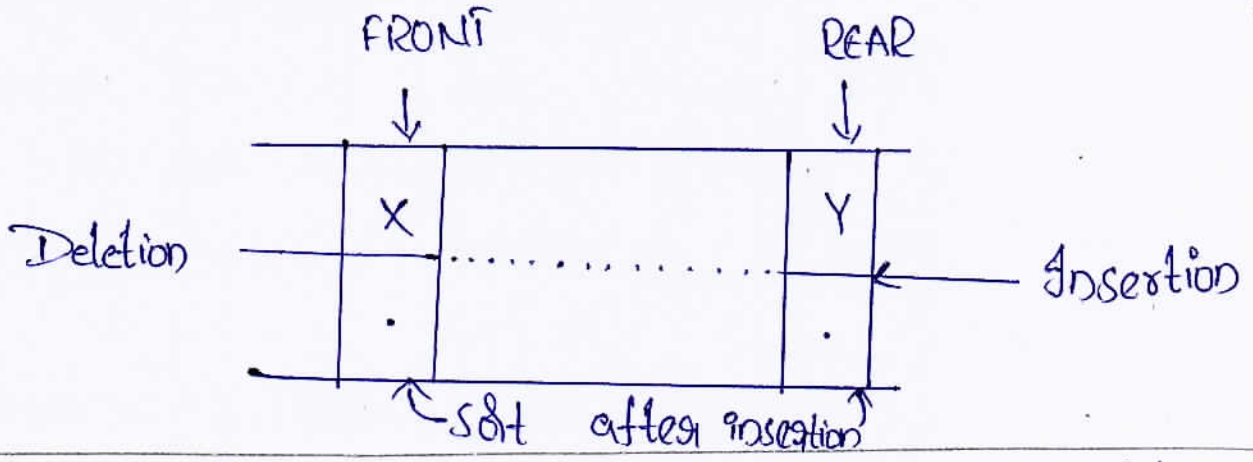


Fig 5.13 :- Another array implementation of a priority queue

Multi-queue implementation:

This implementation assumes 'N' different priority values. For each priority P_i there are two pointers F_i and R_i corresponding to the FRONT and REAR pointers respectively. The elements between F_i and R_i are all of equal priority value P_i. Figure 5.14 represents a view of such a structure.

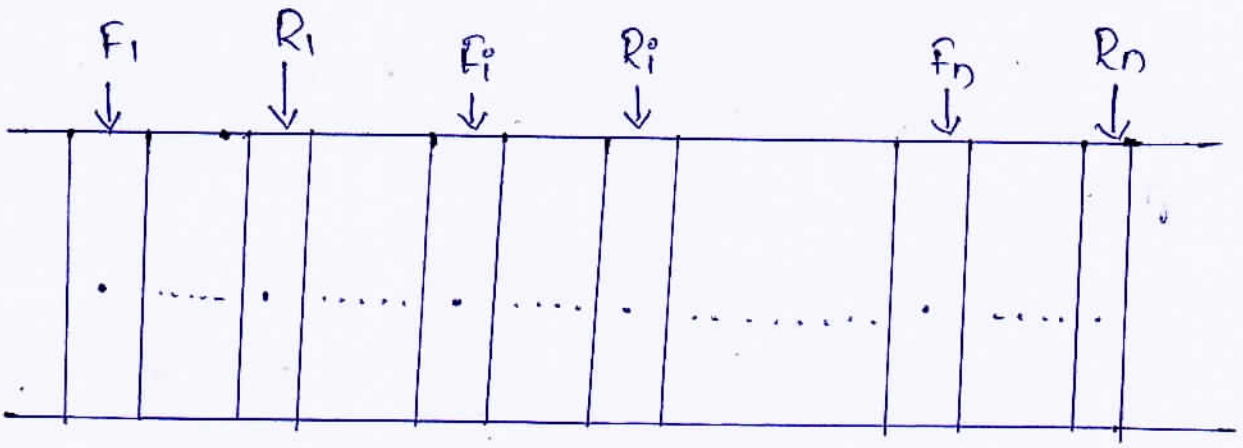


Fig 5.14: Multi-queue representation of a priority queue.

With this representation, an element with priority value P_i will consult F_i for deletion and R_i for insertion. But this implementation is associated with number of difficulties:

(i) It may lead to a huge shifting in order to

make room fd an item to be inserted.
(ii) A large number of pointers are involved when the range of priority value is large.

Linked list representation of a priority queue:

This representation assumes the node structure as shown in figure 5.16. LLINK and RLINK are two usual link fields, DATA is to store the actual content and p PRIORITY is to store the priority value of the item. we will consider FRONT and REAR as two pointers pointing the first and last nodes in the queue, respectively. Here all the nodes are in sorted order according to the priority values of the items in the nodes. The following is an instance of a priority queue.

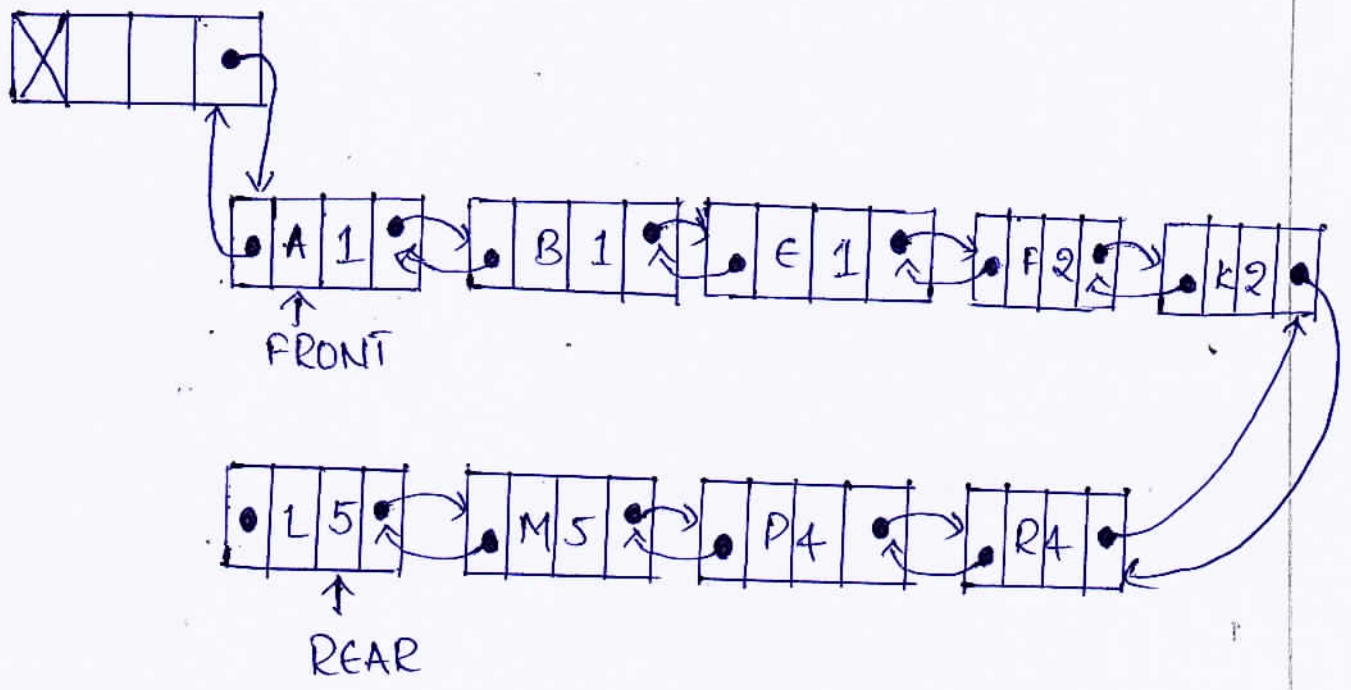


Fig-5.16: Linked list Representation of a priority queue

With this structure, to delete an item having priority p, the list will be searched starting from the node under pointer REAR and the first occuring node with PRIORITY = P will be deleted. Similarly, to insert a node containing an item with priority p, the search will begin from the node under the pointer FRONT and the node will be inserted before a node found first with priority value p, or if not found then before the node with the next priority value.