QUEUES

Unit 3

THE QUEUE AND ITS SEQUENTIAL REPRESENTATION

- A **queue** is an ordered collection of items from which items may be deleted at one end (called the ***front*** of the queue) and into which items may be inserted at the other end (called the ***rear*** of the queue).
- Fig 4.1.a illustrates a queue containing three elements A, B, C.
- A is at the **front** of the queue and C is at the **rear**.
- The first element inserted into a queue is the first element removed.
- It is called a **First-in First-out list**.

THE QUEUE AND ITS SEQUENTIAL REPRESENTATION

- A queue is sometimes called a ***fifo*** (first-in, first-out) list as opposed to a stack, which is a ***lifo*** (last-in, first-out) list.
- Many Examples of a queue exists in real world.

i) A line at a bank or at a bus stop,

ii) a group of cars waiting at a toll booth

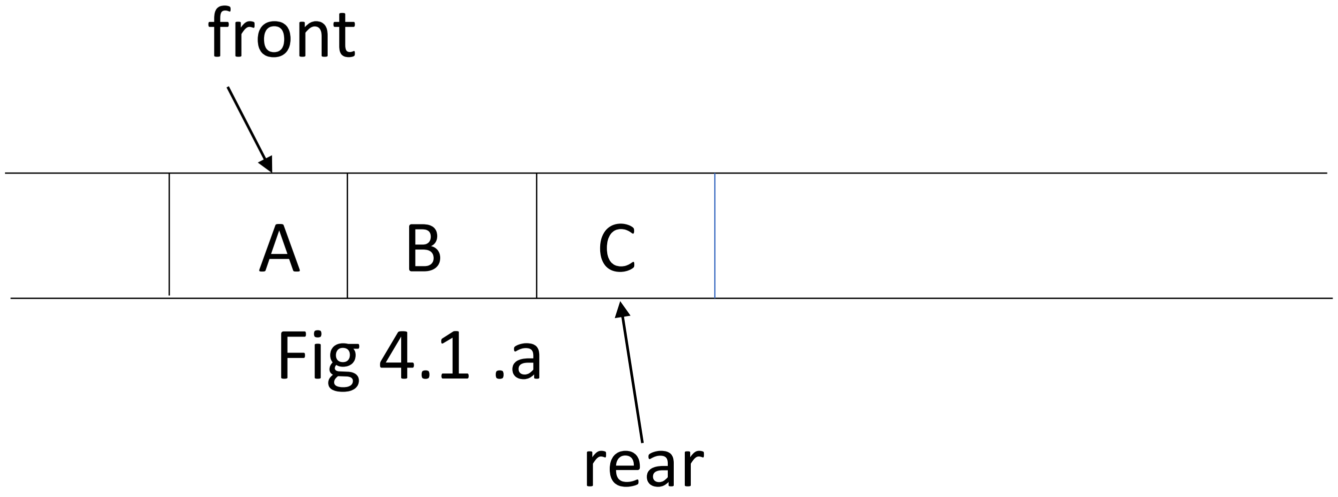


Fig 4.1 .a

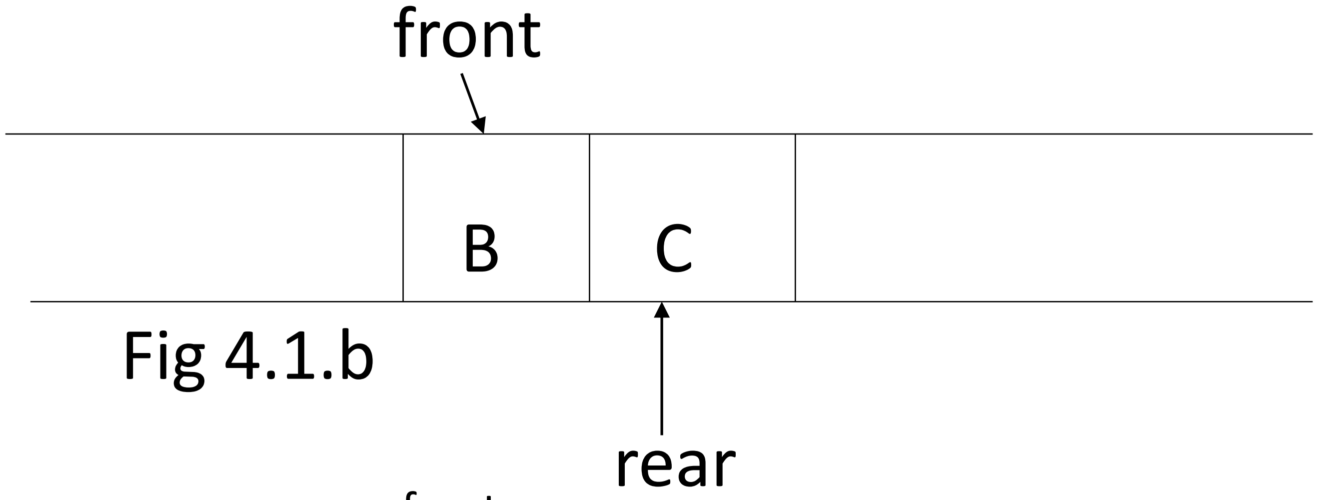


Fig 4.1.b

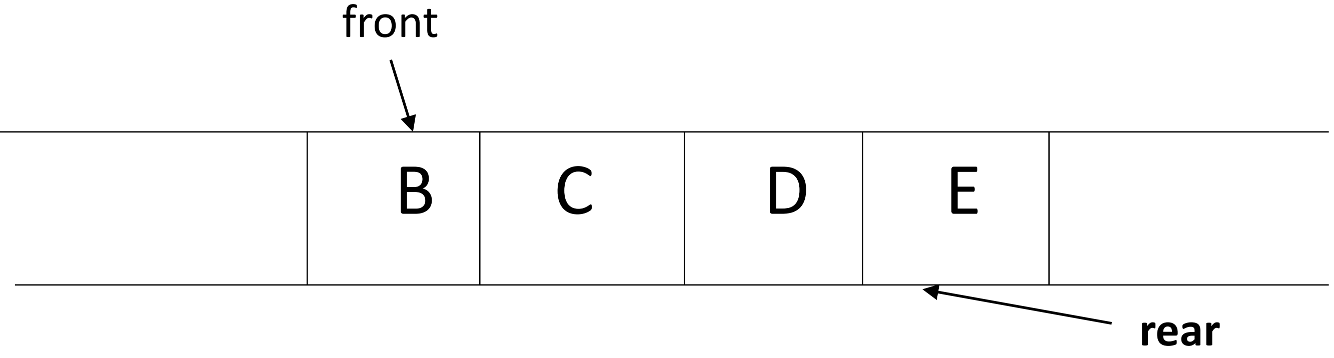


Fig 4.1.c

THE QUEUE AND ITS SEQUENTIAL REPRESENTATION

- **Three primitive operations** can be applied to a queue.
- The operation ***insert(q, x)*** inserts item ***x*** at the rear of the queue q.
- The operation ***x*** **= remove(*q*)** deletes the front element from the queue ***q*** and the sets x to its contents.
- The third operation, ***empty(q)*** returns ***false*** or ***true*** depending on whether or not the queue contains any elements.
- The ***remove*** operation, however, can be applied only if the queue is nonempty.
- The result of an illegal attempt to remove an element from an empty queue is called ***underflow.***

THE QUEUE AS AN ABSTRACT DATA TYPE

```
abstract  typedef<<eltype>> QUEUE(eltype);

abstract empty(q)
QUEUE (eltype) q;
postcondition    empty == (len(q) == 0);

abstract eltype remove(q)
QUEUE(eltype) q;
precondition    empty(q) == FALSE;
postcondition   remove == first(q’);
                   q == sub(q’, 1, len(q’)-1);

abstract  insert(q, elt)
QUEUE (eltype) q;
eltype elt;
postcondition   q == q’ + <elt>;
```

C IMPLEMENTATION OF QUEUES

```
#define  MAXQUEUE 100
struct queue {
 int items [MAXQUEUE];
 int front, rear;
}q;
```

Using an array to hold a queue introduces the possibility of ***overflow.***

The operation ***Insert (q, x)*** could be implemented by

```
 q. items [++ q. rear] = x;
```

C IMPLEMENTATION OF QUEUES

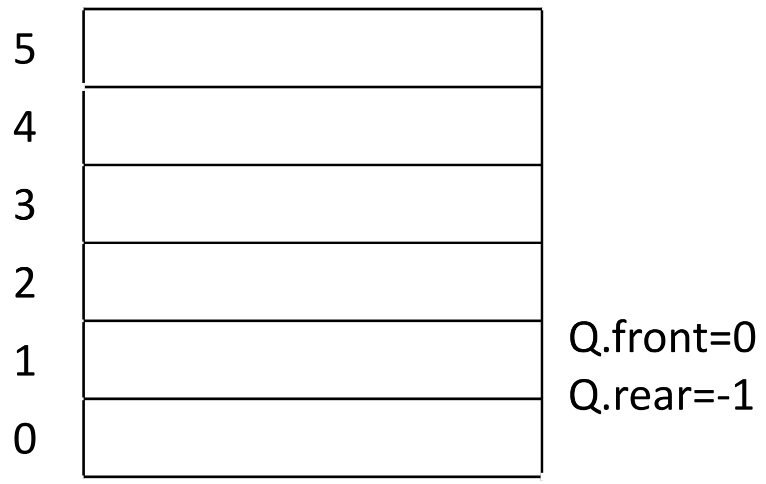
The operation **x = remove(q)** could be implemented by

x = q. items [q. front++]

- Initially, **q. rear is set to -1, and q. front is set to 0.**
- The queue **is empty whenever q.rear < q.front**.
- The number of elements in the queue at any time is equal to the value of **q.rear – q.front + 1.**

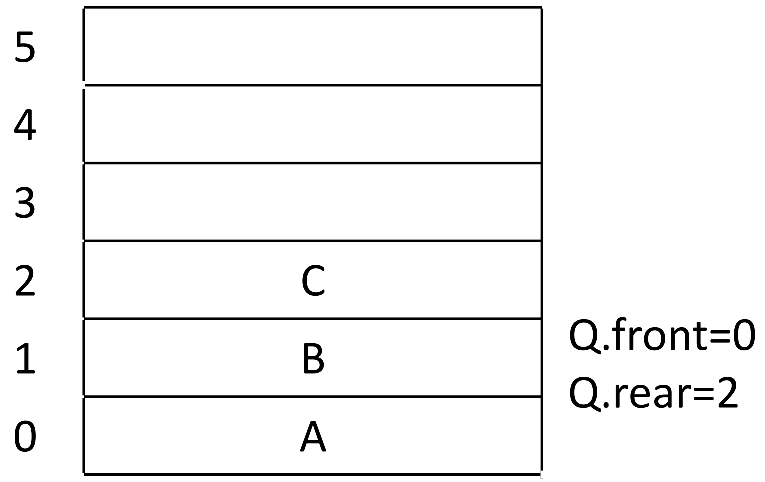
C Implementation of Queue – Contd..

- An array of six elements is used to represent a queue.
- The queue is empty.
- Therefore ***q.rear = -1*** and ***q.front = 0***.



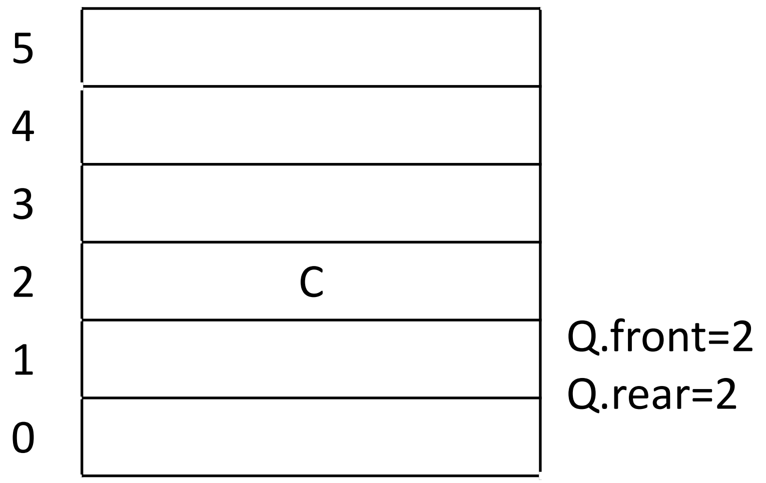
C Implementation of Queues – Contd..

- Items ***A***,***B***,***C*** have been inserted.
- Since elements are inserted at the rear end ***q.rear=2*** and ***q.front=0*** , since no deletion has taken place.



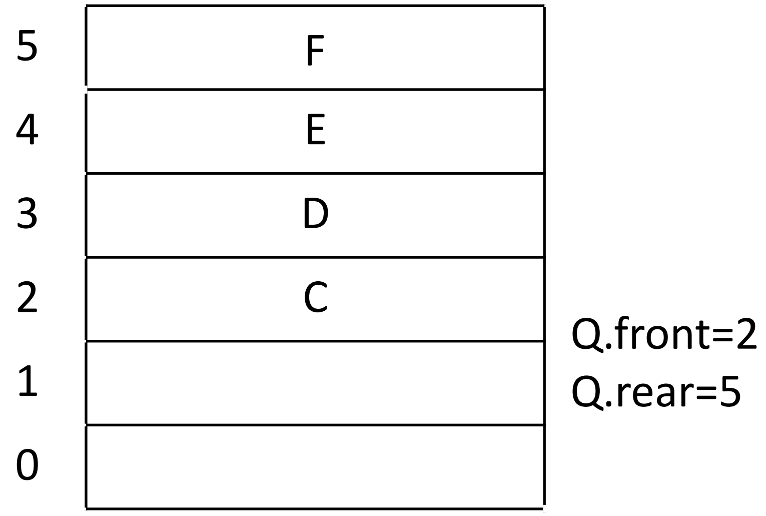
C Implementation of Queues – Contd..

- Items ***A***, ***B*** are deleted from the queue.
- Since deletion takes place at the front end front gets incremented twice.
- Therefore ***q.front = q.rear = 2***.



C Implementation of Queues – Contd..

- Three new items ***D***, ***E***, ***F*** are inserted into the queue. Since elements are inserted at the rear end, the rear value is incremented thrice.
- Therefore ***q.rear = 5*** and ***q.front = 2***.



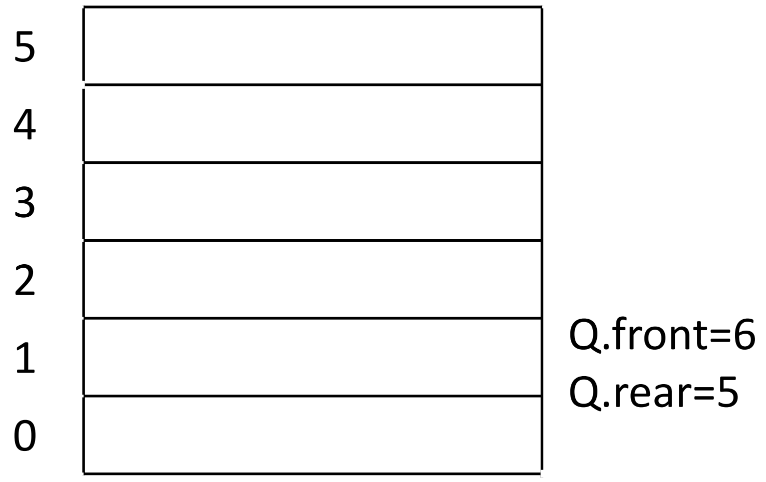
C Implementation of Queues – Contd..

- To insert new elements into the queue ***que.rear*** must be increased to ***6***.
- This is not possible since this is an array of only five elements.
- The insertion cannot be made.
- This situation is also possible when the Whole queue is empty.
- In the previous diagram perform remove operation 4 times

C Implementation of Queues – Contd..

- Since the elements are deleted at the front end, value of front gets incremented four times.
- Therefore ***q.rear = 5 and q.front = 6***.
- Queue empty

q.rear < q.front



C Implementation of Queues – Contd..

- Modify the ***remove*** operation.
- Whenever the item is deleted the entire queue is shifted to the beginning of the array.

```
 x = q. items[0];
for (i = 0; i < q. rear; i++)
q. items [i] = q. items[i + 1];
 q. rear --;
```

- Here **front is not required.** Because front element is **always at 0 position.** But **very inefficient method** as it involves lot of movements of other elements
- **Efficient alternative** – make **array as circular**

Pseudo Code for Normal Queue Operations

- IsEmpty() Operation –

ISEMPTY (FRONT, REAR)
Step 1: If Rear < Front
Return 1

Step 2: Return 0

IsFull() Operation –

ISFULL (REAR, N)

Step 1: If Rear = N – 1

Return 1

Step 2: Return 0

Pseudo Code for Normal Queue Operations

- Insert Operation –

QINSERT (Q, REAR, N, ITEM)

Step 1: [Overflow check]

If IsFull() Then write (‘**overflow** ‘) and Return.

Step 2: [Increment rear pointer] Rear = Rear + 1.

Step 3 : [Insert the item] Q[Rear] = Item

Step 4 : Return

Pseudo Code for Normal Queue Operations

- Delete Operation –

QDELETE (Q, FRONT, REAR, ITEM)

Step 1: [Underflow check]

If IsEmpty() , write (‘**Underflow** ‘) and Return.

Step 2: [Delete the item] Item = Q[Front]

Step 3: If Front = Rear [When there is only one item]

Front = 0 , Rear = -1

Else Front= Front+ 1

Step 4: Return Item

Pseudo Code for Normal Queue Operations

- Display Operation - QDISPLAY (Q, FRONT, REAR, ITEM)

Step 1: [Underflow check]

If Rear = Front – 1(or rear < front) , write (‘QUEUE is empty ‘) and Return.

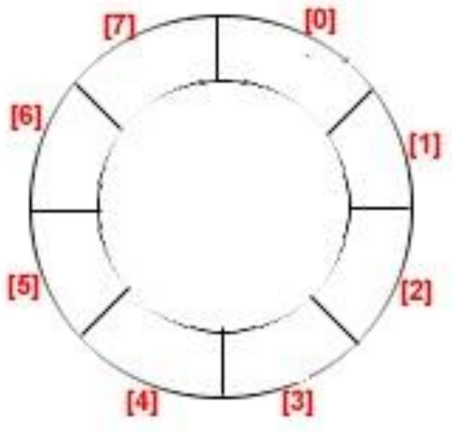
Step 2: [Display the items]

From I = Front to I <= Rear

Write Q [I]

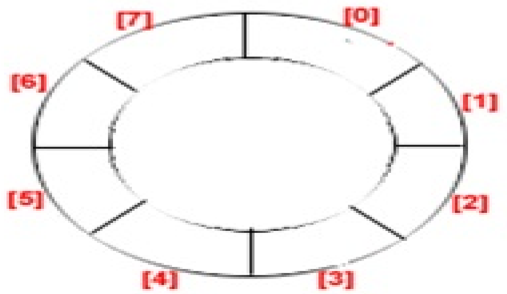
Step 3: Return.

Circular queues



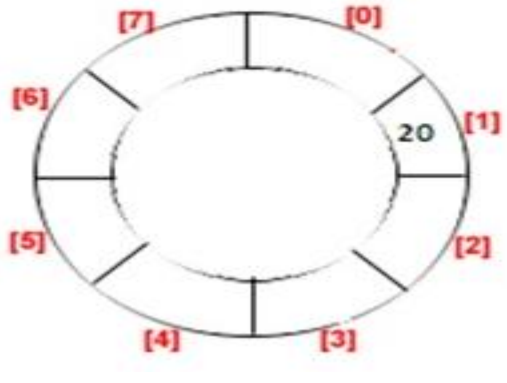
- Queue is considered as a circular queue.
- Each array position has a next and a previous position
- The position next to (maxsize -1) is 0.
- The first element i.e. position ***0*** follows the last element i.e. position (**maxsize -*1)*** immediately.
  - This implies that even if the last position is occupied, a new element can be accommodated at position ***0***.
  - The queue is considered as a circular queue when the positions 0 and MAX-1 are adjacent.
  - It is viewed as in figure

Circular Queue Implementation

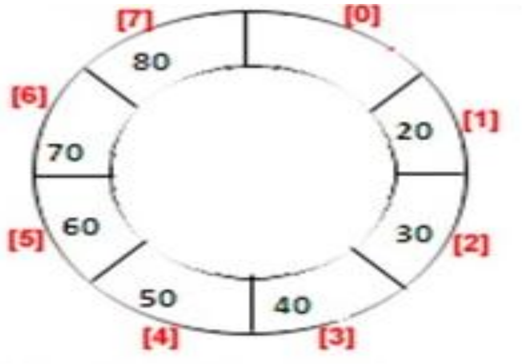


- Initially front = max-1, rear = max-1 = 7
- Insert 10 , 20, front =7, rear = 1
- Delete, front = 0, rear = 1 return 10

(Front is immediately preceding the first element)

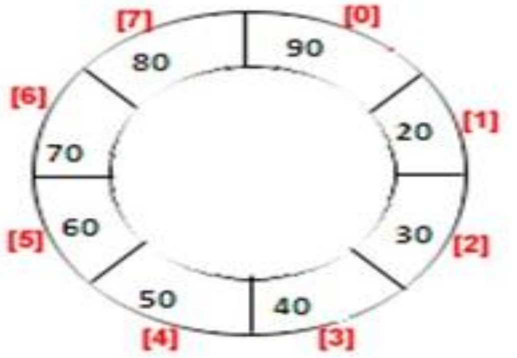


Circular Queue Implementation – Contd..



- Insert 30, 40, 50, 60, 70, 80
- Front = 0, rear = 7
- If it is a linear queue, then the status is
  - Full Queue
- Since it is a circular queue
  - 90 can be inserted , front =0, rear =0

  Front == rear then queue is full



Circular Queue Implementation –

- Remove(Front, Rear, Item,N)
- Step 1 : Start
- Step 2 : If IsEmpty () display Queue Underflow and return
- Step 3 : If front=Max-1 then front = 0
-             else front++
- Step 4 : Item = Q(front)
- Step 5 : return Item

Circular Queue Implementation –

QINSERT(Q, FRONT, REAR, N, ITEM)

- **Step 1 :** [Increment rear pointer]
- If (Rear = max - 1) Rear = 0.
- else Rear += 1
- **Step 2 :** if Rear == Front
  write (‘overflow ‘) and return
- **Step 3 :** [Insert the item] Q[Rear] = Item
- **Step 4 :** Return.

Circular Queue Implementation –

- QDISPLAY(Q, FRONT, REAR, N, ITEM)

- **Step 1:** [Underflow check]
- If IsEmpty() then write (‘QUEUE is empty ‘) and Return.
- **Step 2:** [Display the items]
- If (Front<= Rear) then
- For I = Front+1 to I <= Rear : Write Q [I].
- Else if (Front > Rear) then
- For I = Front+1 to I<=max-1 : Write Q [I].
- For I = 0 to I <= Rear : Write Q [I].
- **Step 3:** Return

Alternatively:
If (isEmpty)
Then write “Queue Empty”
Else
i= front
For j =0 to j <= count
Write Q [(i+1)%n]

Circular Queues

```
#define MAXQUEUE  100
 struct queue {
  int items [MAXQUEUE];
  int front, rear;
};
struct queue q;
 q. front = q. rear = MAXQUEUE-1; // initialized to last index of the array

int empty (struct queue *pq)
{
   return ((pq->front == pq->rear) ? 1 : 0);
}
```

```
if (empty(&q))
 /*queue is empty*/
 else
 /*queue is not empty*/
```

Circular Queues – remove operation

The operation ***remove***(***q***) may be coded as

```
 int remove(struct queue *cq)
 {
   if (empty(cq)) {
   printf(“queue underflow”);
   exit(1);
} /* end if */
If (cq->front == MAXQUEUE-1)
    cq->front = 0;
else
  (cq -> front)++;
return (cq -> items[cq –> front]);
} /* end remove */
```

Insert operation

```
 void insert (struct queue *cq, int x)
  {
    /* make room for new element */
  if (cq -> rear == MAXQUEUE-1)
      cq -> rear = 0;
  else
  (cq -> rear)++;
If (cq->rear == cq->front) // overflow checking
{
   printf(“ queue Overflow”);
     exit(1);
}
cq->items[cq->rear] = x;
return;
}
```

PRIORITY QUEUE

- It is a data structure which **has the intrinsic ordering of the elements**. These determine the results of the basic operations.
- It is a collection of elements such that **each element is assigned a priority** and there is an order in which the elements are deleted and processed.

The rules are

- 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.

PRIORITY QUEUE

- There are two kinds of priority queues
  - ***Ascending priority queue***
  - ***Descending priority queue***.

Ascending Priority Queue

- **The elements can be inserted at any position in the queue but only the smallest element can be removed from the queue.**
- ***ascpqinsert(ascpq,x)*** inserts the element ***x*** into the queue ***ascpq***.
- ***ascpqdelete(ascpq)*** removes the smallest element from the queue ***ascpq*** and returns its value to the user.

Descending Priority Queue

- **The elements can be inserted at any position in the queue but only the largest element can be removed from the queue.**
- ***descpqinsert(descpq, x)*** inserts the element ***x*** into the queue ***descpq***.
- This is logically equivalent to the insert operation of ***ascpqinsert(ascpq, x).***
- ***descpqdelete(descpq)*** removes the largest element from the queue ***descpq*** and returns its value to the user.
- **Delete operation** in both ascending and descending priority queues can be applied only to **non empty queues**

Priority Queue

- The elements of the Priority queue need not be numbers or characters
- It could be any field
- For eg: A **stack** could be viewed as a **Descending Priority Queue** whose elements are **ordered by time of insertion.**
- For eg: A **Queue** could be viewed as a **Ascending Priority Queue** whose elements are **ordered by time of insertion.**
- In both cases time of insertion is not a part of element

Implementation of Priority Queue

- **Several different methods:**
- Maintain a single array, insertion – at the end of the queue and deletion – locate the smallest/largest element and then delete
- This is more costly
- While deleting – just mark it for deletion

  - shift all the subsequent elements
- Maintain array as an ordered list – insertion becomes costly but deletion becomes easy

Array Implementation of a Priority Queue

- A separate queue for each level of priority is maintained.
- Each such queue will appear in its own array and will have its own pair of pointers ***Front*** and ***Rear***.
- First specify the priority of the item and the item itself to be inserted.
- If the queue of the given priority is ***full*** display the condition ***Queue Overflow***.
- If the queue is not full insert the elements at the rear end of the given priority queue.

Priority Queue – Insertion()

Step 1 : Start

Step 2: //Overflow Condition

If rear[priority]=N-1 then display “Queue Overflow” and return

Step 3 : //Increment rear

rear[priority]=rear[priority]+1

Step 4 : //Assign item to the queue at rear position

queue[priority][rear[priority]]=item

Step 5 : Stop

Priority Queue – Delete()

- Delete operation **does not require any priority checking.**
- It deletes the element at the front position in the first queue.
- If that queue is empty, the element at the front position in the second queue is deleted and so on.

Priority Queue – Delete()

Step 1 : Start

Step 2 : For I from 0 to (number of queues)

a. if rear[i] < front[i] then display Queue I is empty

b. Else

i. Assign queue[i][front[i]]to item

ii. Increment front[i]

iii. Return item

Step 3 : Return

Note: Number of queues = no of priorities

Priority Queue – Display()

Step 1: Start

Step 2 : for I from 0 to 3(no of queues)

a. if rear[i]=front[i]-1 then display Queue I Underflow

b. else

i. Display Queue I

ii. for j from front[i] to rear[i] display queue[i][j]

Step 3 : Return

Other types of Queues

- DEQUE , ***double ended queue***, where insertions and deletions can be done from both ends
- It can also be maintained by a circular queue.
- There are two kinds
  - ***Input restricted Deque***
  - ***Output restricted Deque***

Other types of Queues

- **Input Restricted Deque** -It allows insertions only at one end of the queue but allows deletion at both the ends.
- The insertions are possible only at the ***rear*** end.
- **Output Restricted Deque** - It allows deletions only at one end of the queue but allows insertion at both the ends.
- The deletion is possible only at the ***front*** end.

Queue Operations - Terminology

- **Enqueue**
  - Add element to tail of queue
- **Dequeue**
  - Extract and use element at head of queue
- **Unqueue**
  - Remove and not use the element from queue;
- **Requeue**
  - Add again to the end of the queue an element previously extracted.

- Thank You