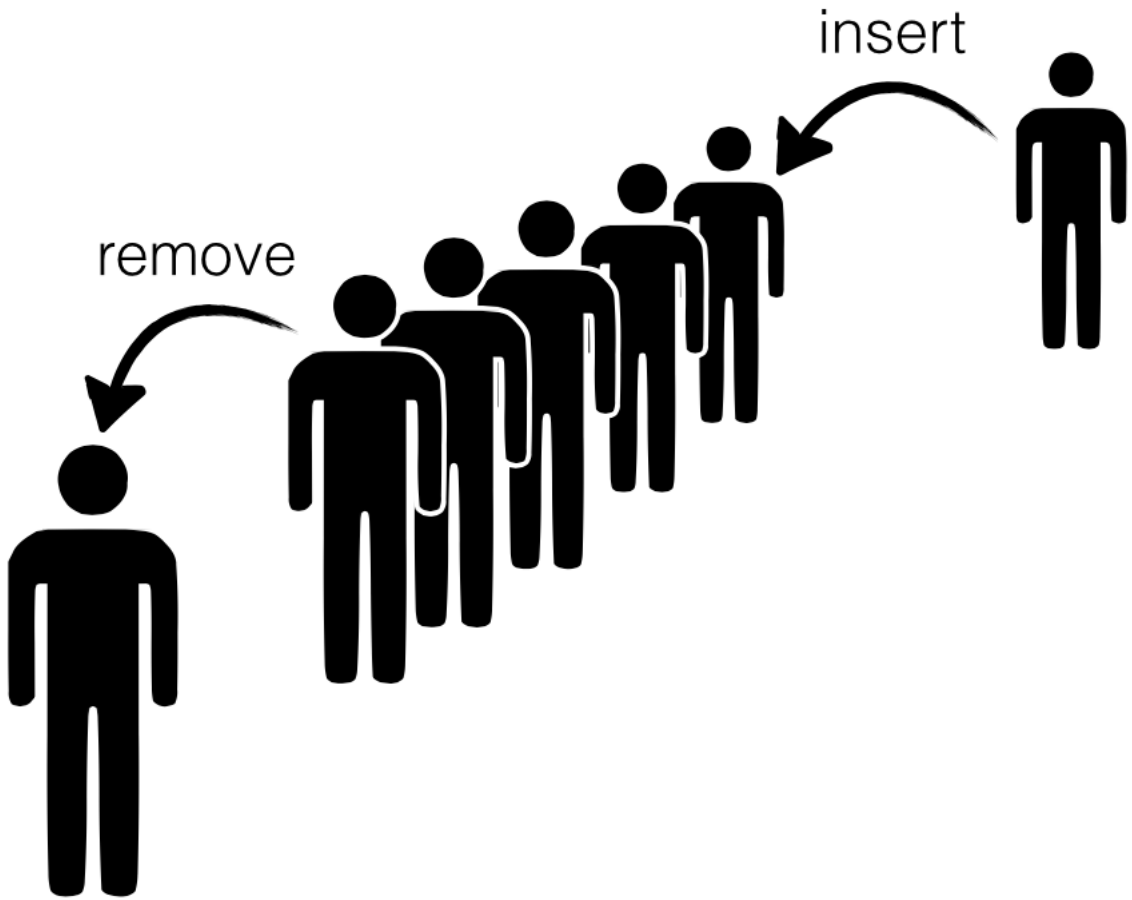
# Queue

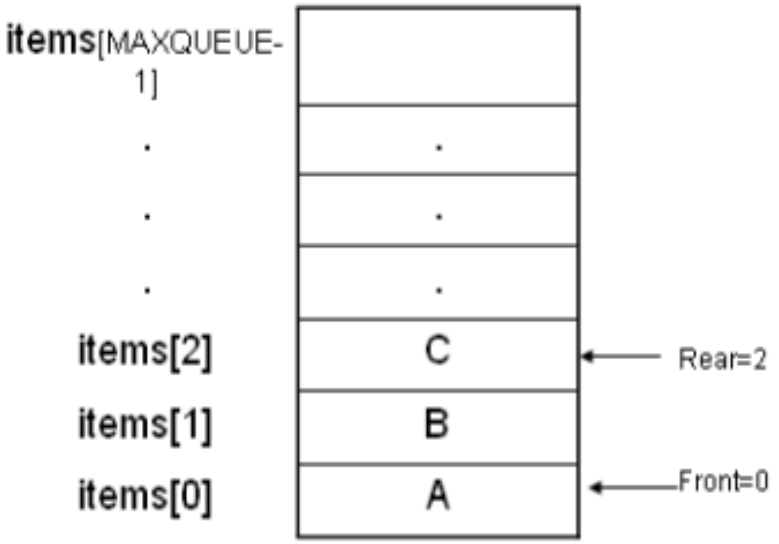
Unit 3



# Queue

- Concept and definition
- Queue as ADT
- Implementation of insert and delete operation of
  - Linear queue
  - Circular queue
- Concept of priority queue

# What is a queue?




A queue is a non primitive linear data structure. It is an **ordered collection of homogenious 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 (the **rear** of the queue).

The first element inserted into the queue is the first element to be removed. For this reason a queue is sometimes called a **fifo (first-in first-out)** list as opposed to the stack, which is a lifo (last-in first-out).

# Operations in Queue

**MakeEmpty(q):** To make q as an empty queue

**Enqueue(q, x):** To insert an item x at the rear of the queue, this is also called by names add, insert.

**Dequeue(q):** To delete an item from the front of the queue q. this is also known as Delete, Remove.

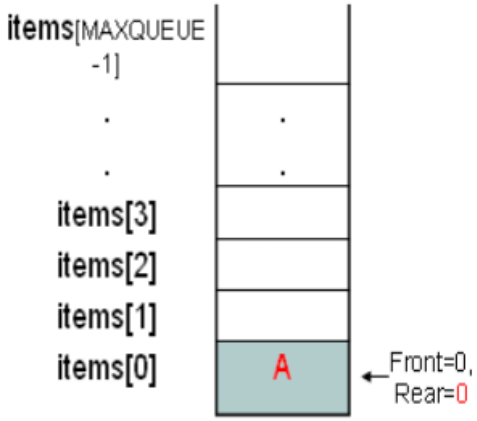
**IsFull(q):** To check whether the queue q is full.

**IsEmpty(q):**To check whether the queue q is empty

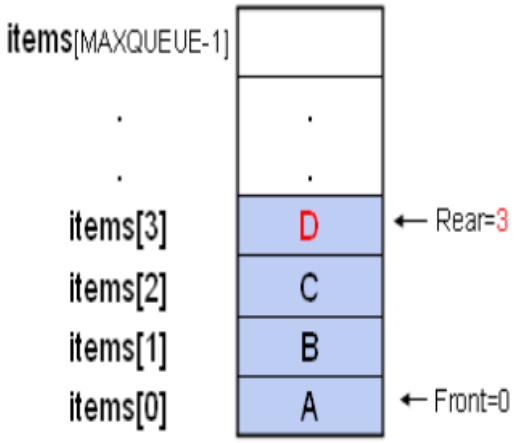
**Traverse (q):**To read entire queue that is display the content of the queue.

# Example Enqueue/Dequeue

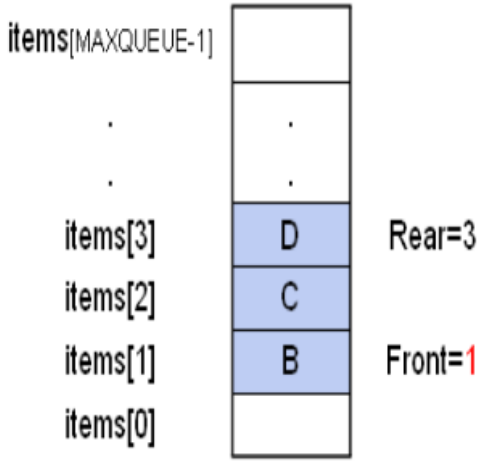
**Enqueue(A):**



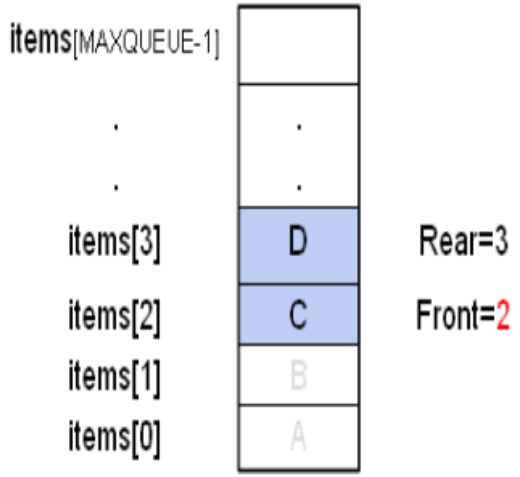

**Enqueue(B,C,D):**




Dequeue(A):

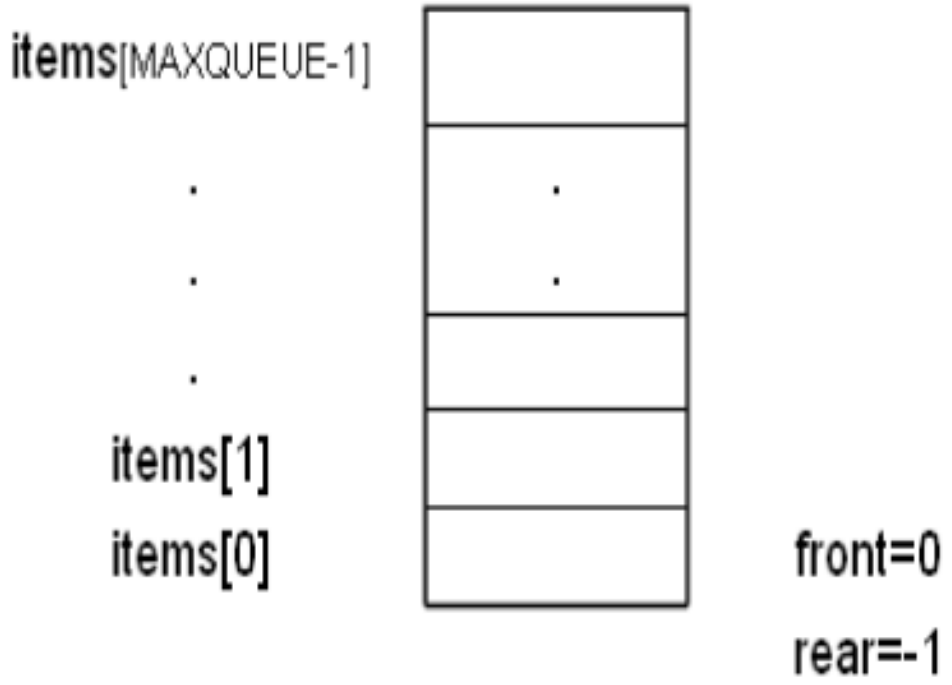



Dequeue(B):



# Initialization of queue:

- The queue is initialized by having the rear set to -1, and front set to 0.

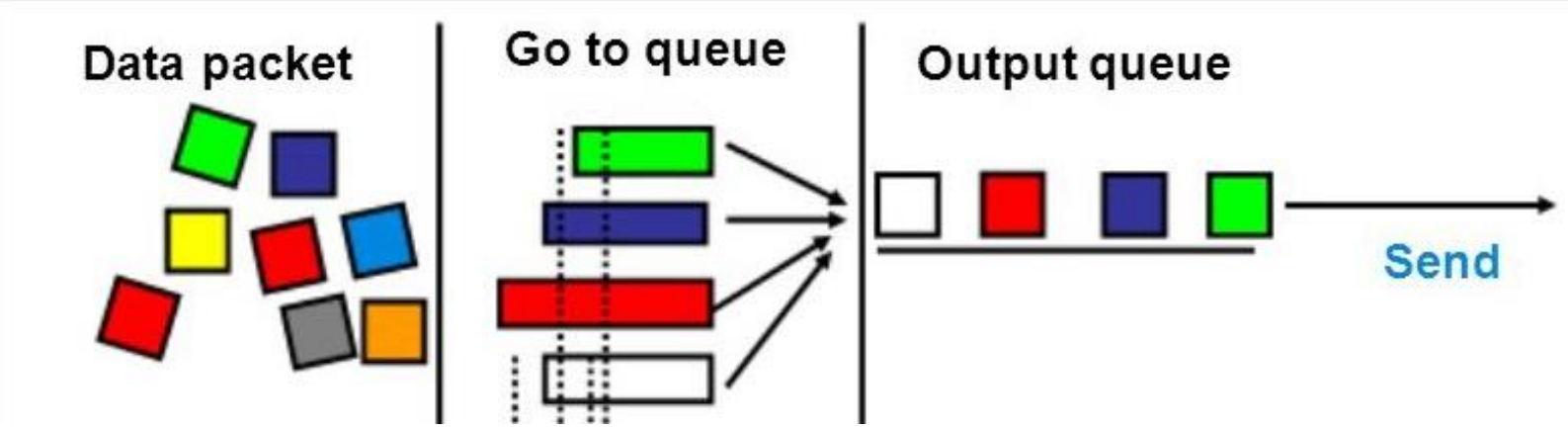


# Real World Example of Queue

- People waiting on a queue to get on a bus.
- People on an Escalator
- Cars at gas station

# Applications of Queue

- Task waiting for the printing
- Time sharing system for use of CPU
- For access to disk storage
- Task scheduling in operating system
- Queue of network data packets to send



# Queue as an ADT

A queue q of type T is a finite sequence of elements with the operations

- MakeEmpty(q): To make q as an empty queue
- IsEmpty(q): To check whether the queue q is empty. Return true if q is empty, return false otherwise.
- IsFull(q): To check whether the queue q is full. Return true in q is full, return false otherwise.
- Enqueue(q, x): To insert an item x at the rear of the queue, if and only if q is not full.
- Dequeue(q): To delete an item from the front of the queue q. if and only if q is not empty.
- Traverse (q): To read entire queue that is display the content of the queue

# Implementation of queue:

- There are two techniques for implementing the queue:
  - Array implementation of queue(static memory allocation)
  - Linked list implementation of queue(dynamic memory allocation)

# Array implementation of queue:

- **an array is used** to store the data elements.
- We must be sure about the exact number of elements we want to store in the queue.
  - Because we have to declare the size of the array at design time or before the processing starts.
- Total number of elements present in the queue =

  queue.rear – queue.front + 1

- If queue.rear < queue.front
  - Then there will be no elements in the queue
  - Or queue is empty

# Array Implementation of Queue:

- Array implementation is also further classified into two types:
- **Linear array implementation:**
  - A linear array with two indices always increasing that is rear and front.
  - Linear array implementation is also called linear queue.

- **Circular array implementation:**
  - This is also called circular queue.

# Linear queue:

- Algorithm for insertion an item in queue:

1. Initialize front=0 and rear=-1

        if rear>=MAXSIZE-1

                print “queue overflow” and return

        else

                set rear=rear+1

        queue[rear]=item

2. end

# Linear queue:

- Algorithm to delete an element from the queue:

1. if rear<front
        print “queue is empty” and return
   else
        item=queue[front++]
2. end

# Declaration of a Queue:

```
# define MAXQUEUE 50 /* size of the queue items*/
struct queue
{
        int front;
        int rear;
        int items[MAXQUEUE];
        };
typedef struct queue qt;
```

# Queue Operations: The MakeEmpty function:

```
void makeEmpty(qt *q)
{
        q->rear=-1;
        q->front=0;
}
```

# Queue Operations: The IsEmpty function:

```
int IsEmpty(qt *q)
{
        if(q->rear < q->front)
                return 1;
        else
                return 0;
}
```

# Queue Operations: The Isfull function:

```
int IsFull(qt *q)
{
        if(q->rear == MAXQUEUEZIZE-1)
                return 1;
        else
                return 0;
}
```

# Queue Operations: The Enqueue function:

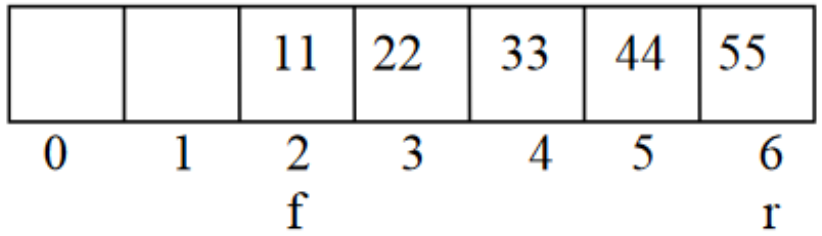
```
void Enqueue(qt *q, int newitem)
{
        if(IsFull(q))
        {
                printf("queue is full");
                exit(1);
        }
        else
        {
                q->rear++;
                q->items[q->rear] = newitem;
        }
}
```

# Queue Operations: The Dequeue function

```
int Dequeue(qt *q)
{
        if(IsEmpty(q))
        {
                printf("queue is Empty");
                exit(1);
        }
        else
        {
                return(q->items[q->front]);
                q->front++;
        }
}
```

# Problems with Linear queue implementation:

- Both rear and front indices are increased but never decreased.
- As items are removed from the queue, the storage space at the beginning of the array is discarded and never used again.
  - **Wastage of the space** is the main problem with linear queue.



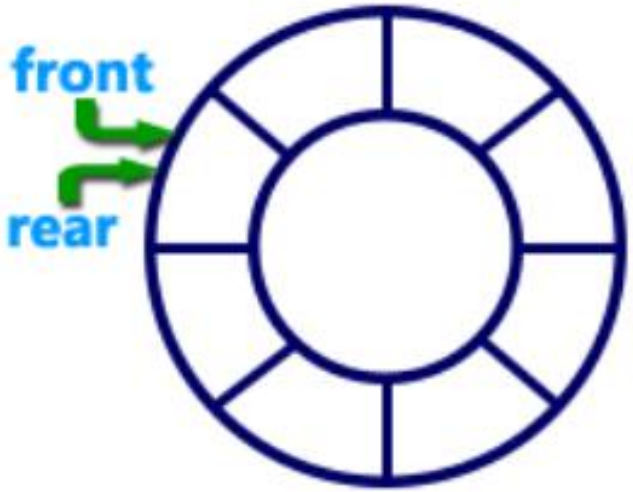

front=2, rear=6

This queue is considered full, even though the space at beginning is vacant.

# Circular Queue

Queue is Full


Queue is Full (Even three elements are deleted)

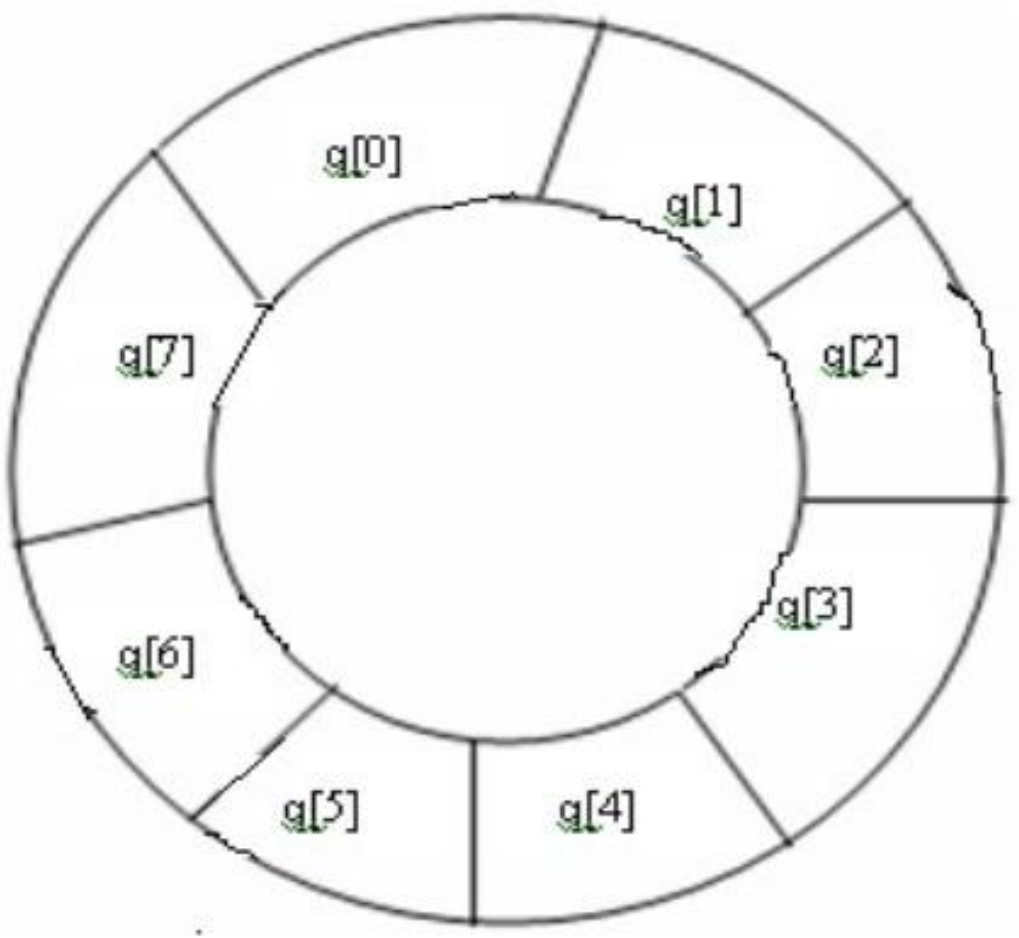
# Circular queue:

- the insertion of a new element is done at very first location of the queue if the last location of the queue is full.
- A circular queue is one in which the first element comes just after the last element.
- overcomes the problem of unutilized space in linear queue implementation as array.

# Initialization of Circular queue:

rear=-1

front= -1



# Algorithms for inserting an element in a circular queue:

This algorithm is assume that rear and front are initially set to -1

```
1. if (front==(rear+1)%MAXSIZE)
        print “Queue is full and exit”.
   else
        IF (Front == -1)
                set Front = Rear = 0;
        else
        rear=(rear+1)%MAXSIZE;
2. cqueue[rear]= value;
3. End if
```

# Algorithms for deleting an element from a circular queue:

1. if (Front=-1)
    print Queue is empty and exit
Else Item = Queue[front]
    If (front == Rear)
        set Front = -1;
        set Rear = -1;
    Else: Front = (Front + 1) % MAXSIZE
End if
2. Exit

# Declaration of a Circular Queue:

- #define MAXSIZE 50 /* size of the circular queue items*/

```
struct cqueue
{
        int front;
        int rear;
        int items[MAXSIZE];
};
typedef struct cqueue cq;
```

# Circular Queue Operations: The MakeEmpty function:

```
void makeEmpty(cq *q)
{
        q->rear=-1;
        q->front=-1;
}
```

# The IsEmpty function:

```
int IsEmpty(cq *q)
{
        if(q->front= -1)
                return 1;
        else
                return 0;
}
```

# The Isfull function:

```
int IsFull(cq *q)
{
        if(q->front==(q->rear+1)%MAXSIZE)
                return 1;
        else
                return 0;
}
```

# Priority queue:

- A priority queue is a collection of elements such that each element has been assigned a priority and the order in which elements are deleted and processed comes from the following rules:
  - An element of higher priority is processed before any element of lower priority.
  - If two elements has same priority then they are processed according to the order in which they were added to the queue.
- The best application of priority queue is observed in CPU scheduling.

✔ The jobs which have higher priority are processed first.

✔ If the priority of two jobs is same this jobs are processed according to their position in queue.

✔ A short job is given higher priority over the longer one.

# Types of priority queues:

- **Ascending priority queue(min priority queue):**
  - An ascending priority queue is a collection of items into which items can be inserted arbitrarily but from which **only the smallest item can be removed**.

- **Descending priority queue(max priority queue):**
  - An descending priority queue is a collection of items into which items can be inserted arbitrarily but from which **only the largest item can be removed**.

# Priority QUEUE Operations:

- **Insertion :** The insertion in Priority queues is the same as in non-priority queues.
- **Deletion :** Deletion requires a search for the element of highest priority and deletes the element with highest priority.
  - The following methods can be used for deletion/removal from a given Priority Queue:
  - An empty indicator replaces deleted elements.
  - After each deletion elements can be moved up in the array decrementing the rear.
  - The array in the queue can be maintained as an ordered circular array

# Priority Queue Declaration:

- Queue data type of Priority Queue is the same as the Non-priority Queue.

```
#define MAXQUEUE 10 /* size of the queue items*/
Struct pqueue
{
        int front;
        int rear;
        int items[MAXQUEUE];
};
Struct pqueue *pq;
```

# The priority queue ADT:

A ascending priority queue of elements of type T is a finite sequence of elements of T together with the operations:

- **MakeEmpty(p):** Create an empty priority queue p
- **Empty(p):** Determine if the priority queue p is empty or not
- **Insert(p,x):** Add element x on the priority queue p
- **DeleteMin(p):** If the priority queue p is not empty, remove the minimum element of the quque and return it.
- **FindMin(p):** Retrieve the minimum element of the priority queue p.

# Array implementation of priority queue:

Unordered array implementation:

- To insert an item, insert it at the rear end of the queue.
- To delete an item, find the position of the minimum element and
  - Either mark it as deleted (lazy deletion) or
  - shift all elements past the deleted element and then decrement rear.

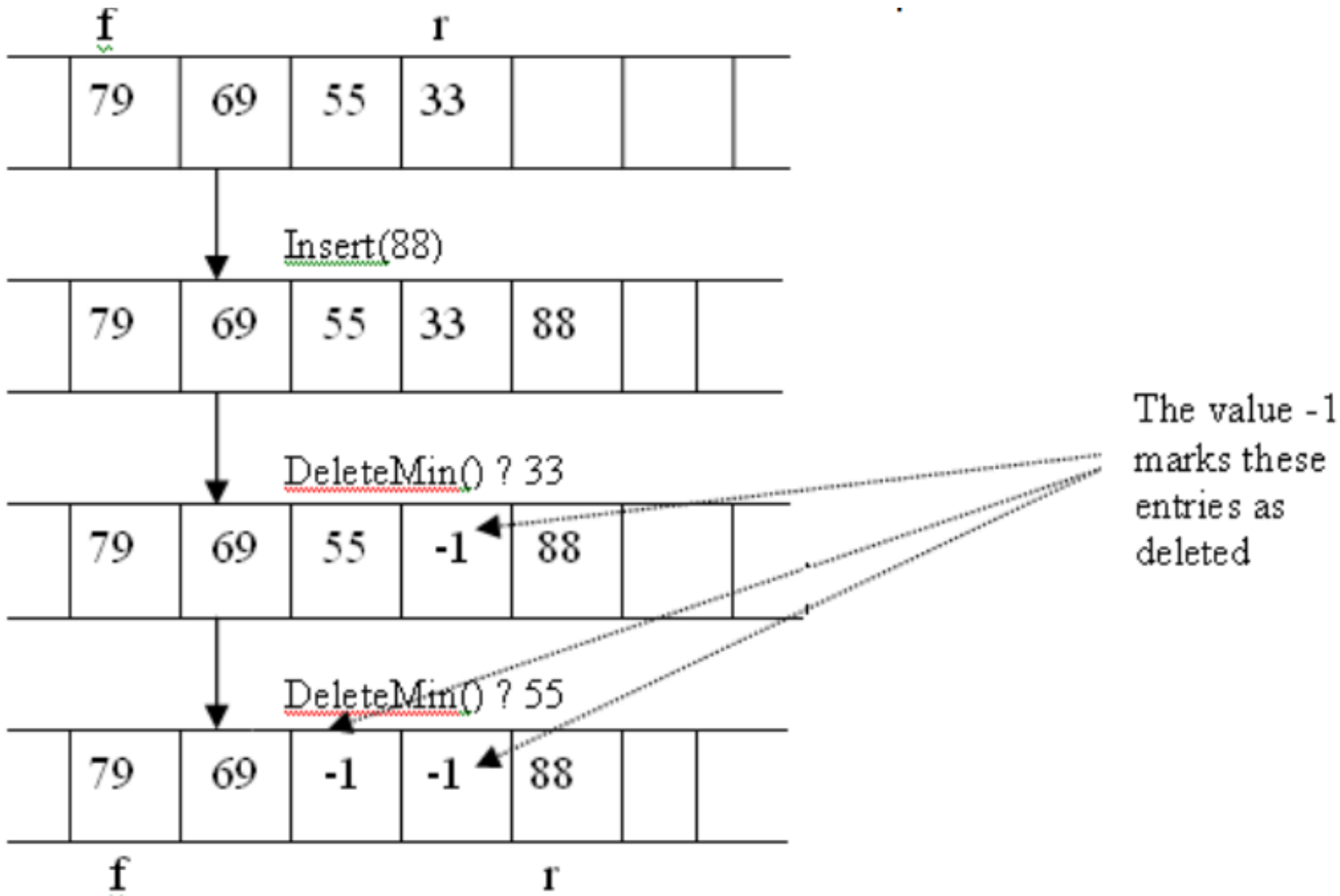



Fig Illustration of unordered array implementation

# Ordered array implementation:

- Set the front as the position of the smallest element and the rear as the position of the largest element.
- To insert an element, locate the proper position of the new element and shift preceding or succeeding elements by one position.
- To delete the minimum element, increment the front position.

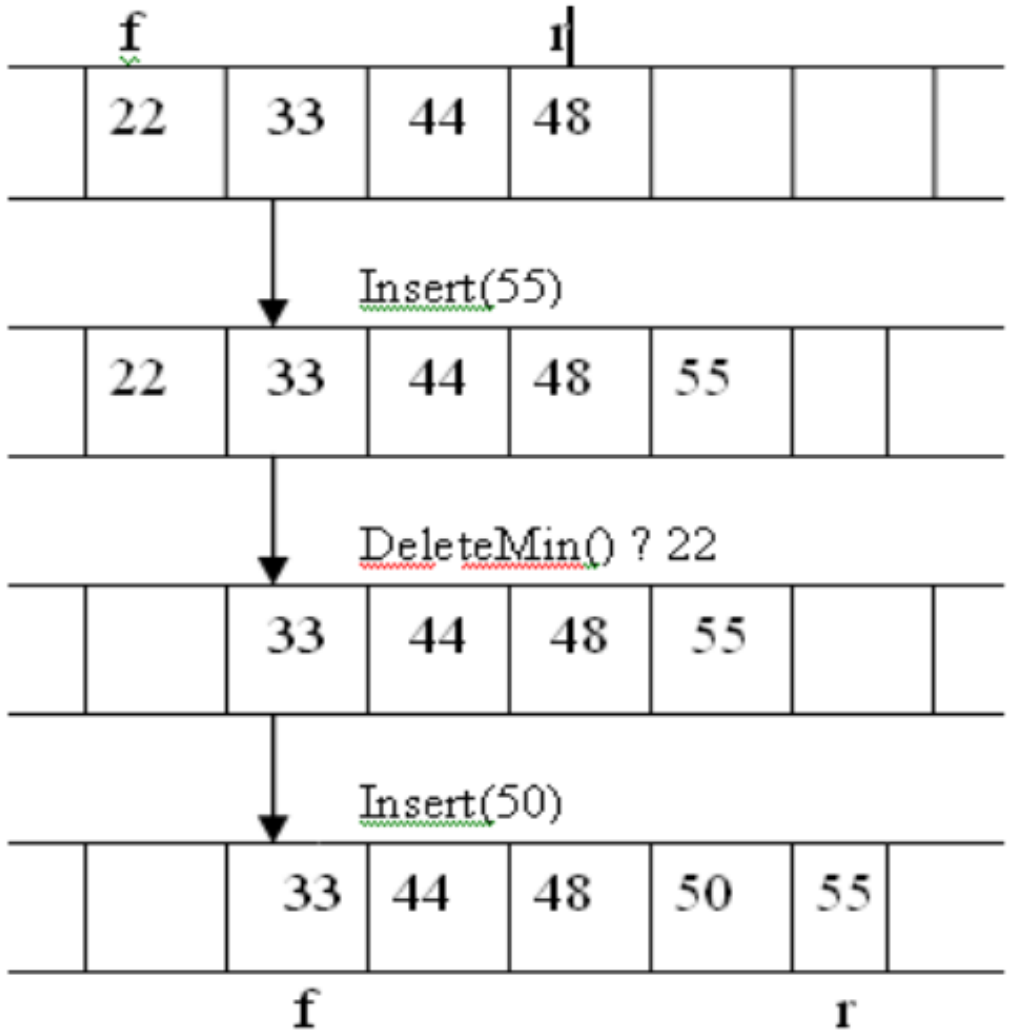



Fig Illustration of ordered array implementation

# Application of Priority queue:

- In a time-sharing computer system, a large number of tasks may be waiting for the CPU, some of these tasks have higher priority than others. The set of tasks waiting for the CPU forms a priority queue.

# double-ended queue (deque)

- elements can be added to or removed from either the front (head) or back (tail)
- Differs from FIFO queue
- possible sub-types:
  - Input Restricted Deque
    - deletion can be made from both ends, but insertion can be made at one end only.
  - Output-Restricted Deque
    - insertion can be made at both ends, but deletion can be made from one end only.

# double-ended queue (deque)

- Operations:
  - insert element at back
  - insert element at front
  - remove last element
  - remove first element
  - examine last element
  - examine first element