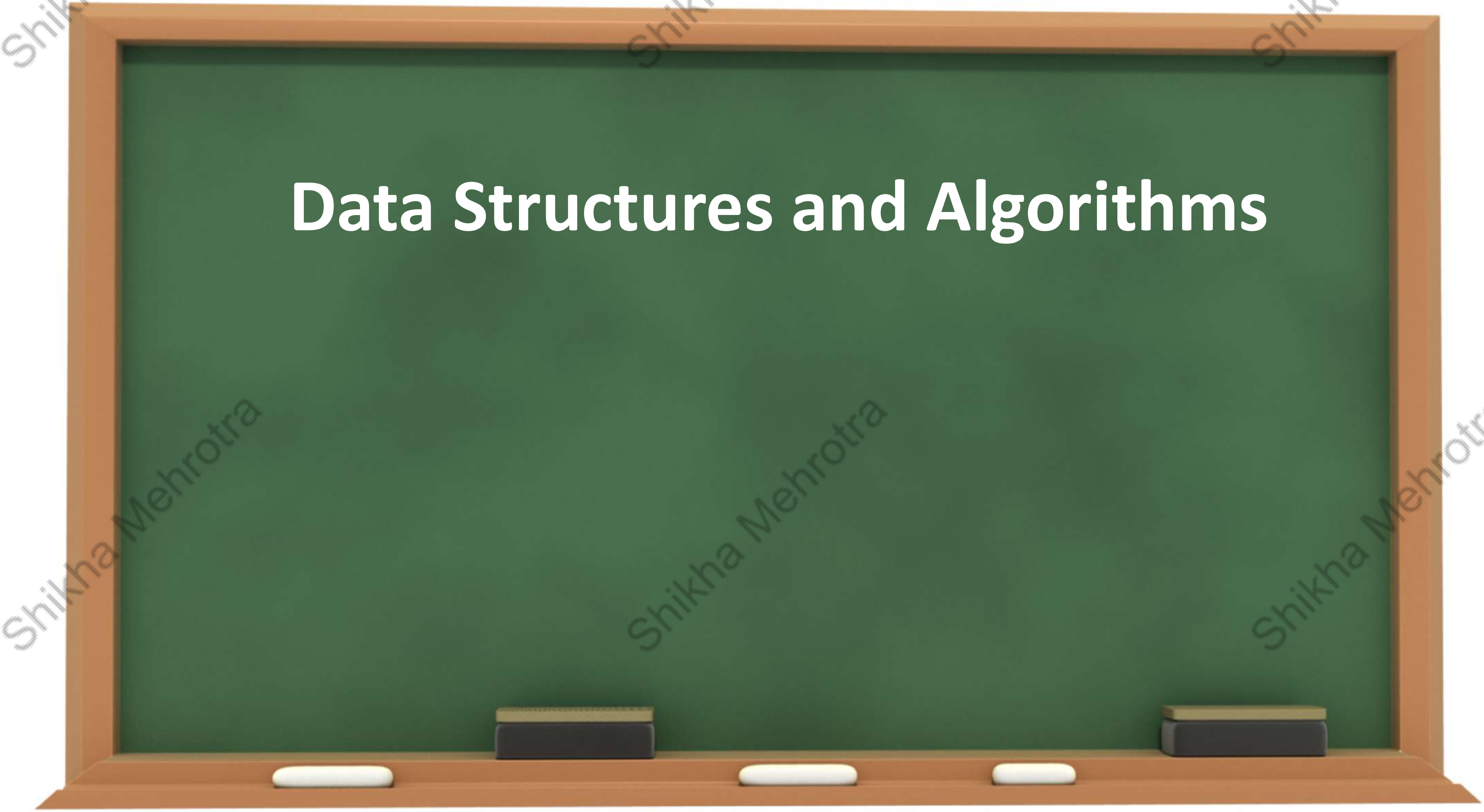
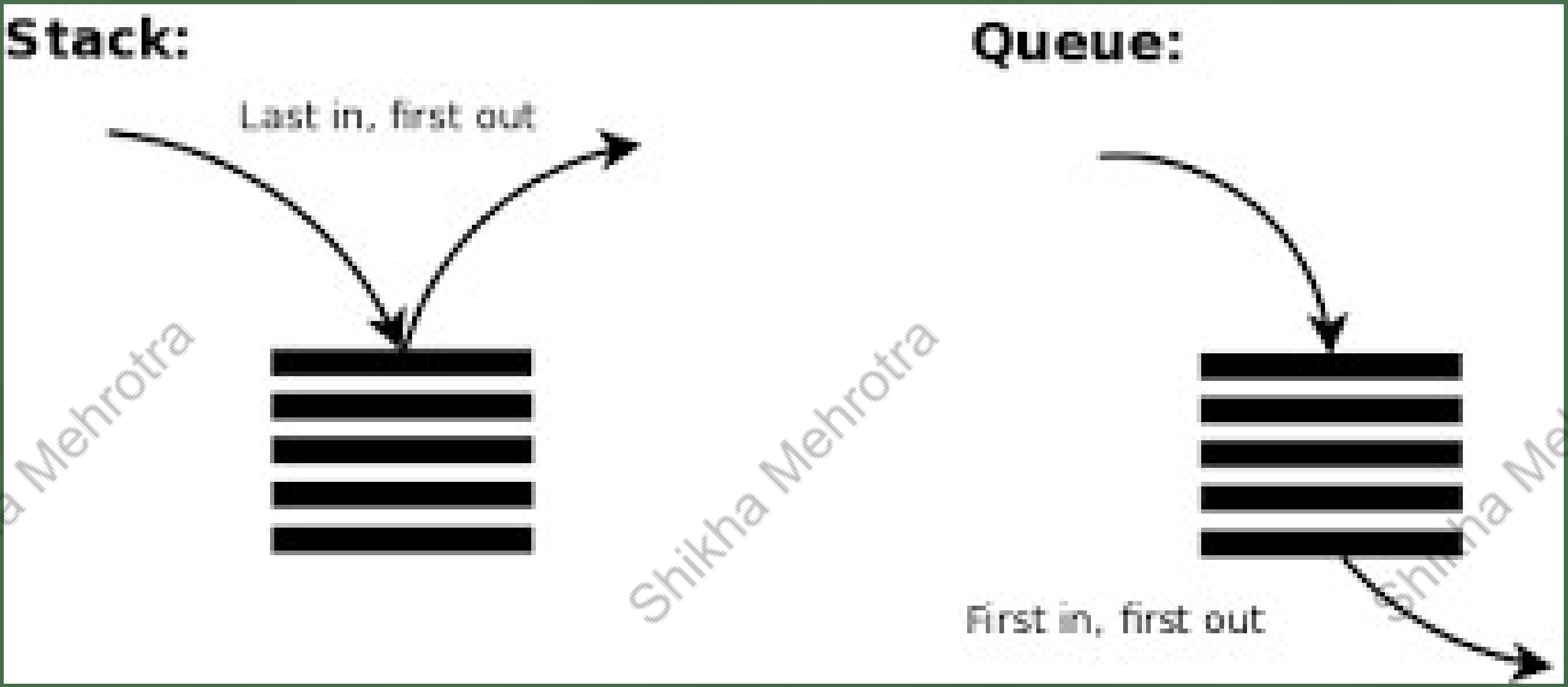


Data Structures and Algorithms

Shikha Mehrotra

The Queue

Shikha Mehrotra

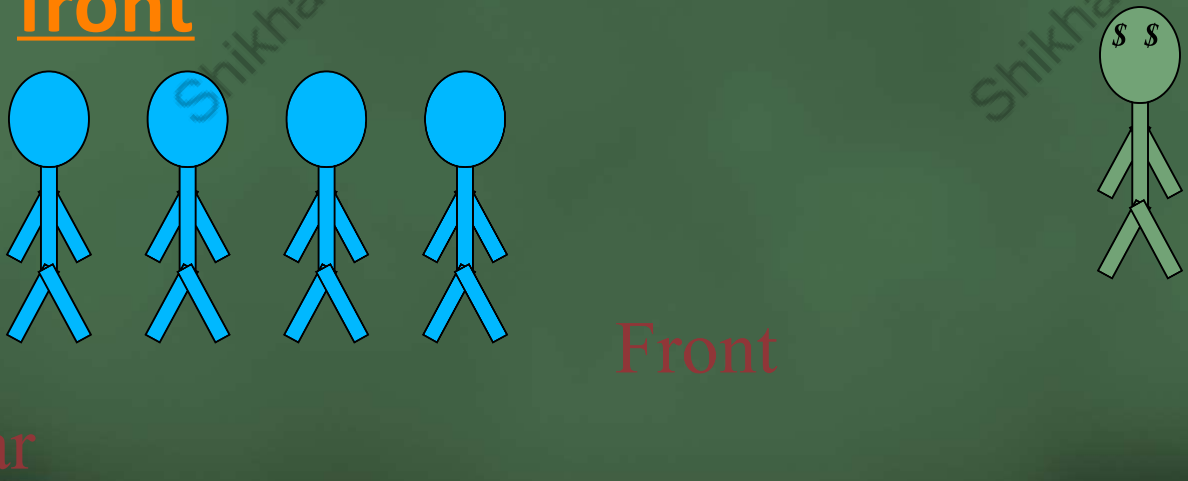

Stack:
Last in, first out
Queue:
First in, first out

The Queue ADT

A list of collection with the restriction that insertion can be performed at one end (rear) and deletion can be performed at other end.

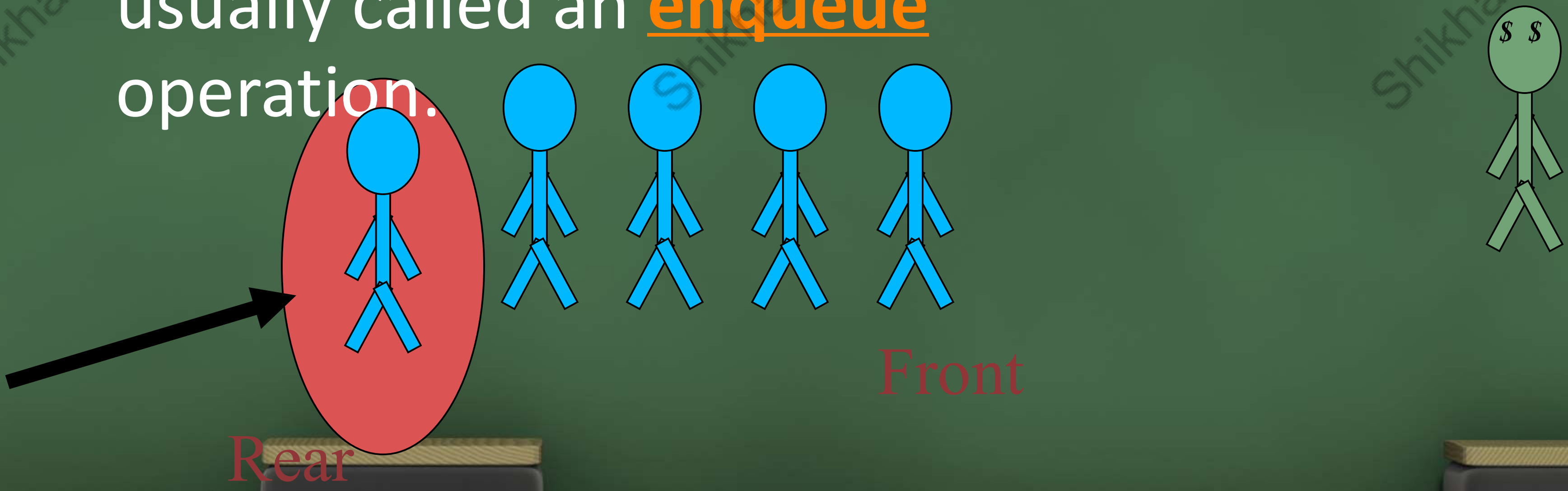
- **The Queue Operations**

  - A queue is like a line of people waiting for a bank teller. The queue has a **front** and a **rear**.



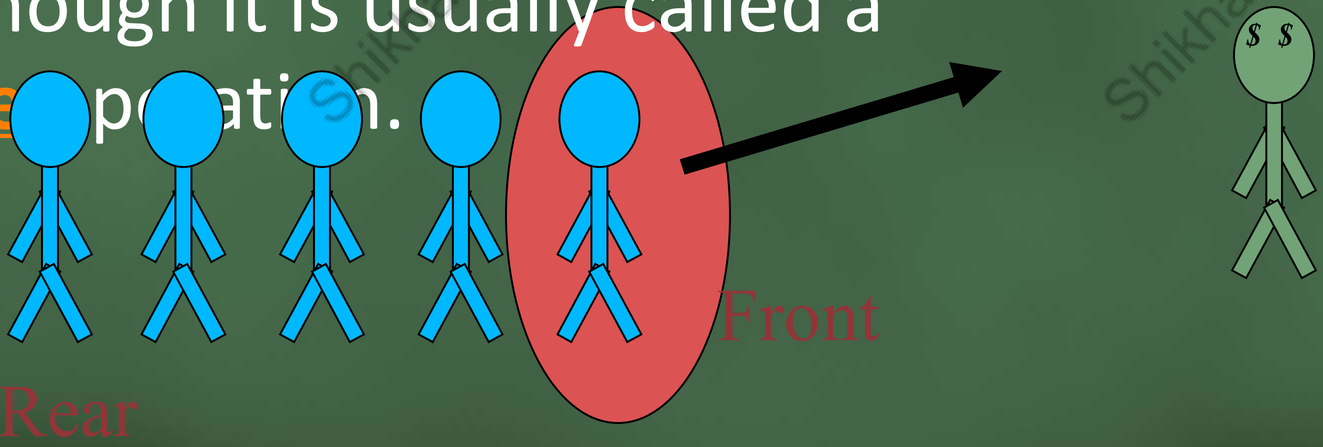
- **The Queue Operations**

  - New people must enter the queue at the rear. The C++ queue class calls this a **push**, although it is usually called an **enqueue** operation.



The Queue Operations

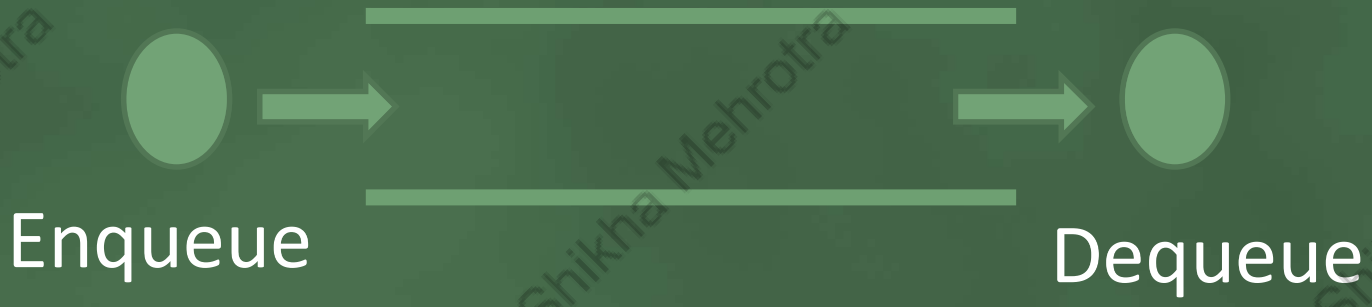
- When an item is taken from the queue, it always comes from the front. The C++ queue calls this a **pop**, although it is usually called a **dequeue** operation.

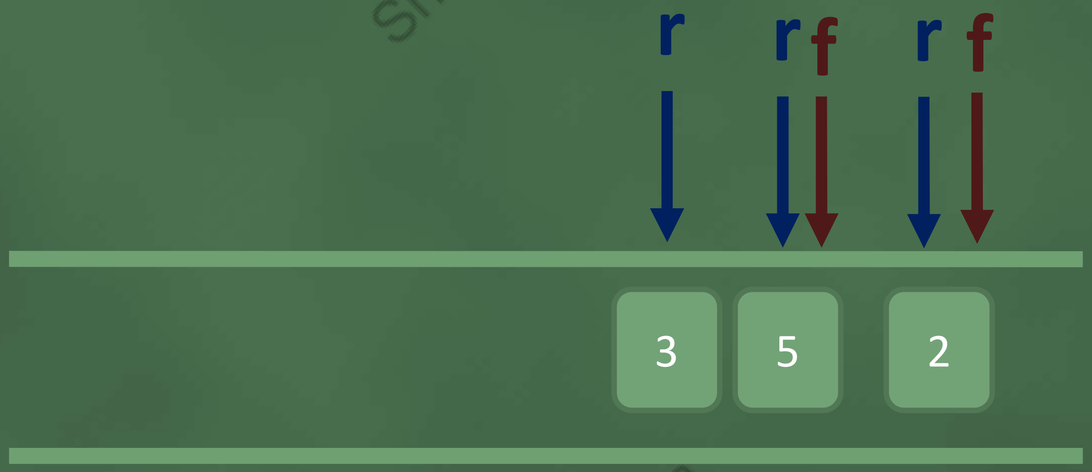


Queue ADT

```
AbstractDataType queue {
  instances
          ordered list of elements; one end is the front; the other is the rear;
  operations
          IsEmpty():          Return true if queue is empty, return false otherwise
          size():             Return the number of elements in the queue
          front():            Return the front element of queue
          dequeue():          Remove an element from the queue
          enqueue(x):         Add element x to the queue
```

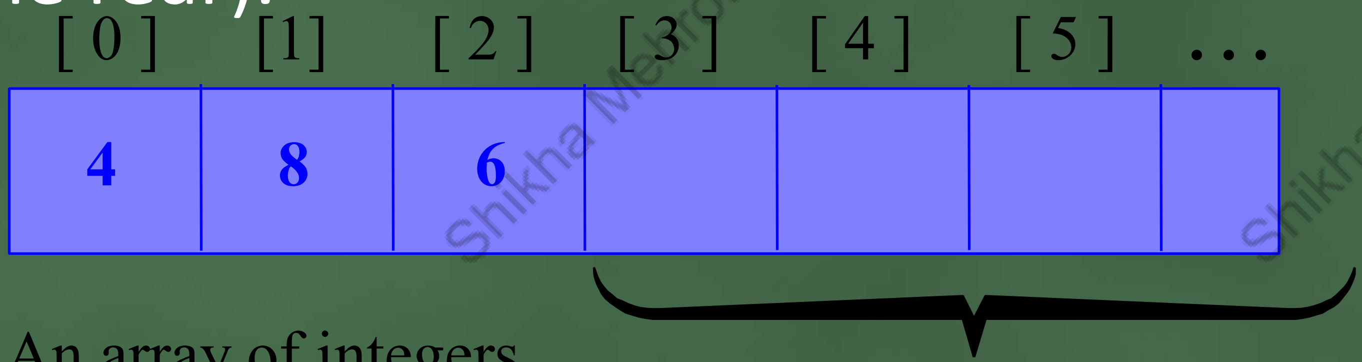
- Queue





Enqueue(2)
Enqueue(5)
Enqueue(3)
Dequeue()
Front()
IsEmpty()

- **Array Implementation**
  - A queue can be implemented with an array, as shown here. For example, this queue contains the integers 4 (at the front), 8 and 6 (at the rear).

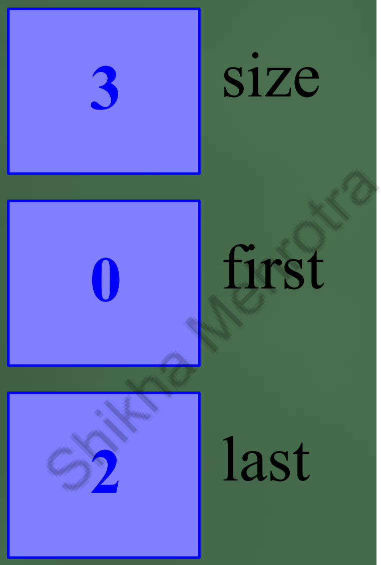


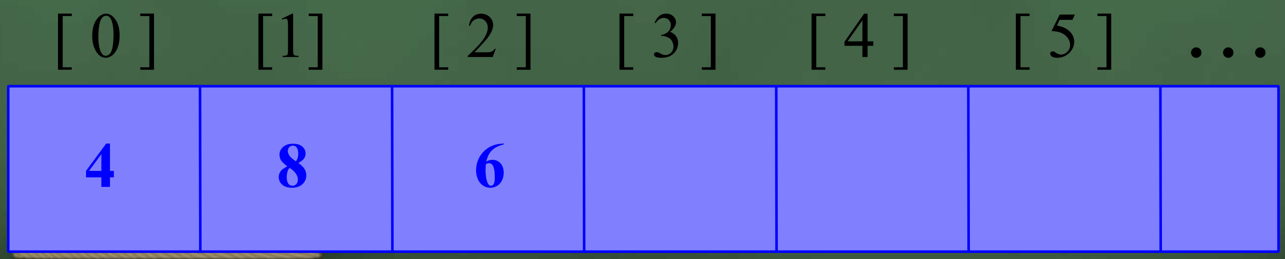
An array of integers to implement a queue of integers

We don't care what's in this part of the array.

Array Implementation

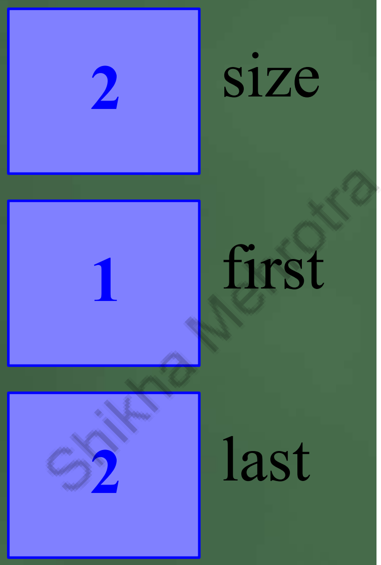
- The easiest implementation also keeps track of the number of items in the queue and the index of the first element (at the front of the queue), the last element (at the rear).

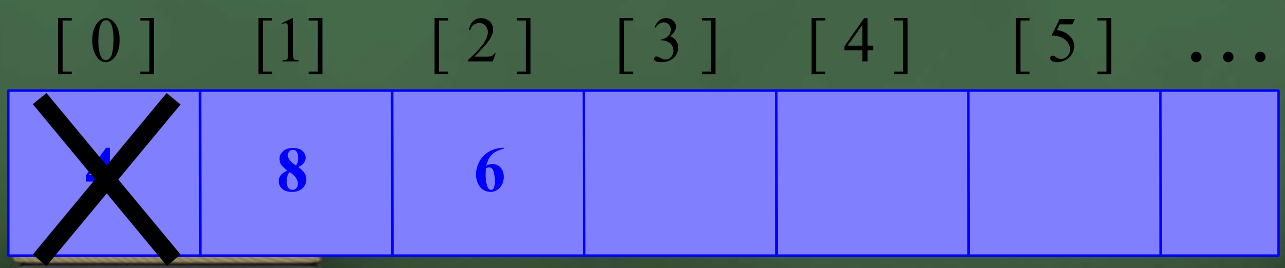




A Dequeue Operation

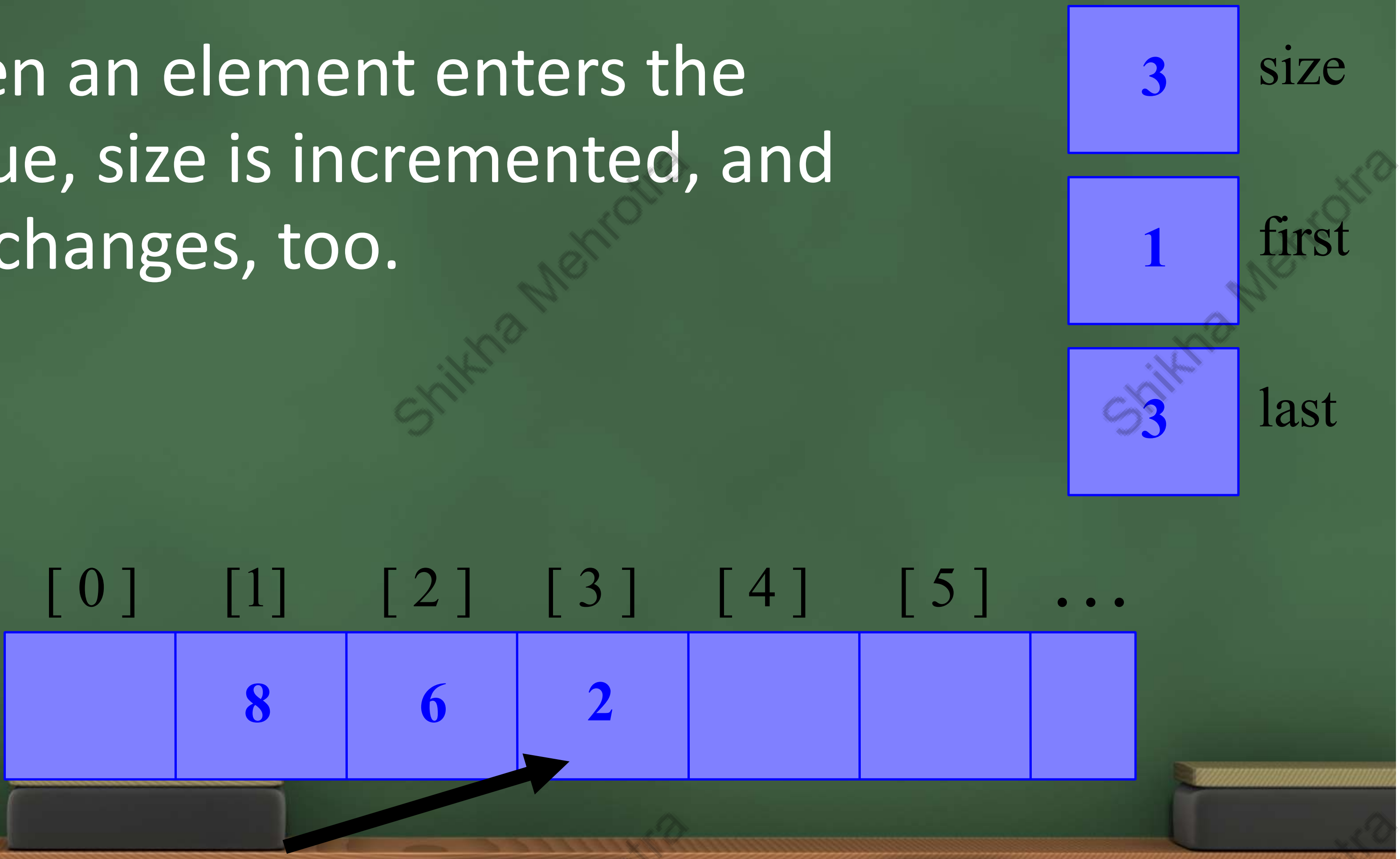
- When an element leaves the queue, size is decremented, and first changes, too.





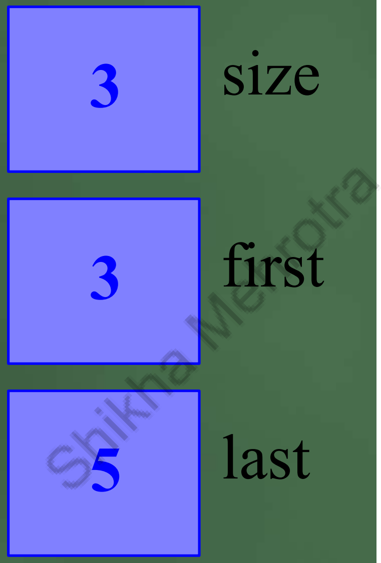
- **An Enqueue Operation**

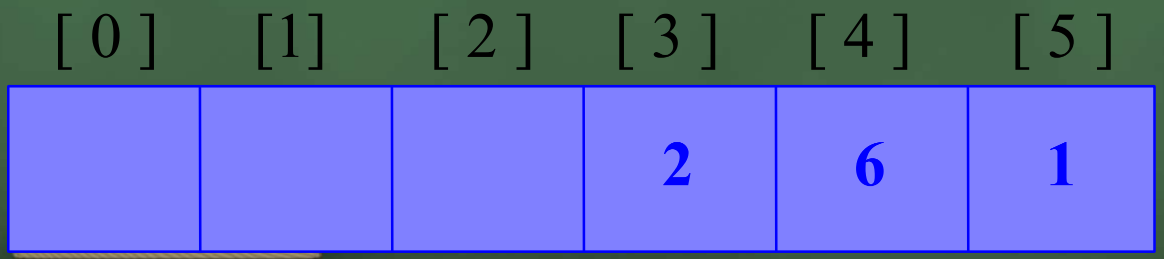
  - When an element enters the queue, size is incremented, and last changes, too.



- **At the End of the Array**

  - There is special behavior at the end of the array. For example, suppose we want to add a new element to this queue, where the last index is [5]:

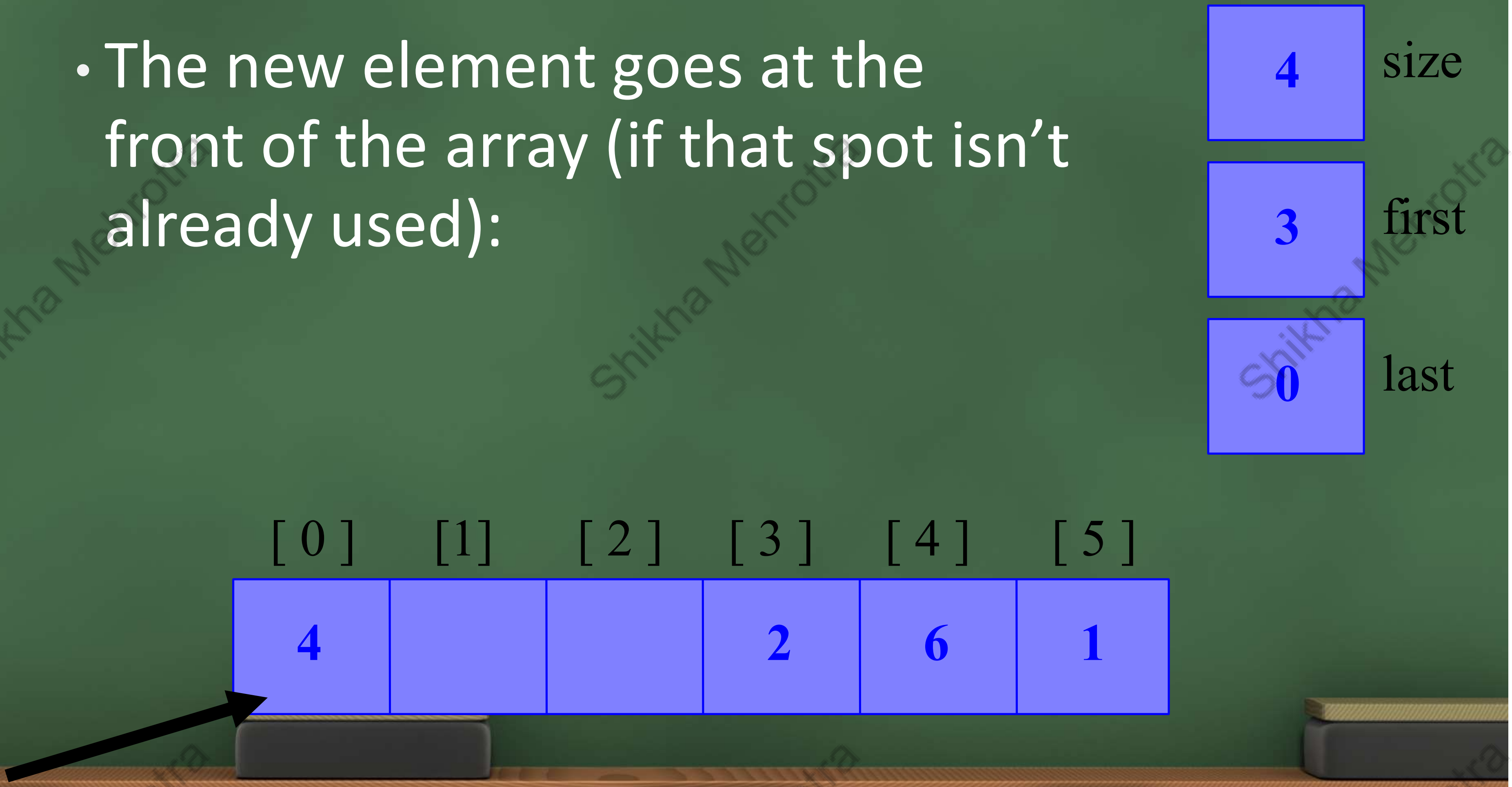




[0]	[1]	[2]	[3]	[4]	[5]
			2	6	1

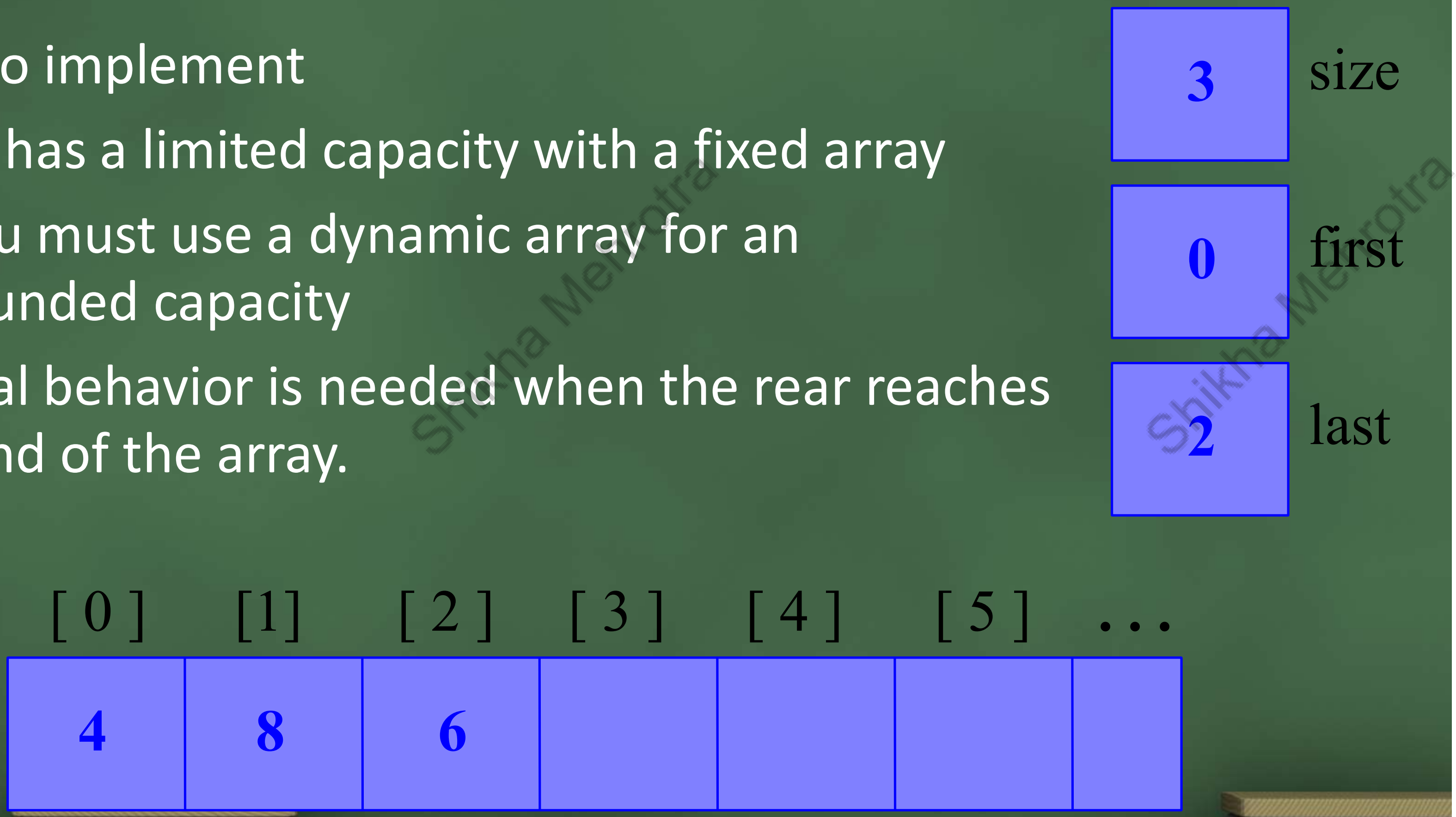
• At the End of the Array

- The new element goes at the front of the array (if that spot isn't already used):



• Array Implementation

- Easy to implement
- But it has a limited capacity with a fixed array
- Or you must use a dynamic array for an unbounded capacity
- Special behavior is needed when the rear reaches the end of the array.



Applications of Queue

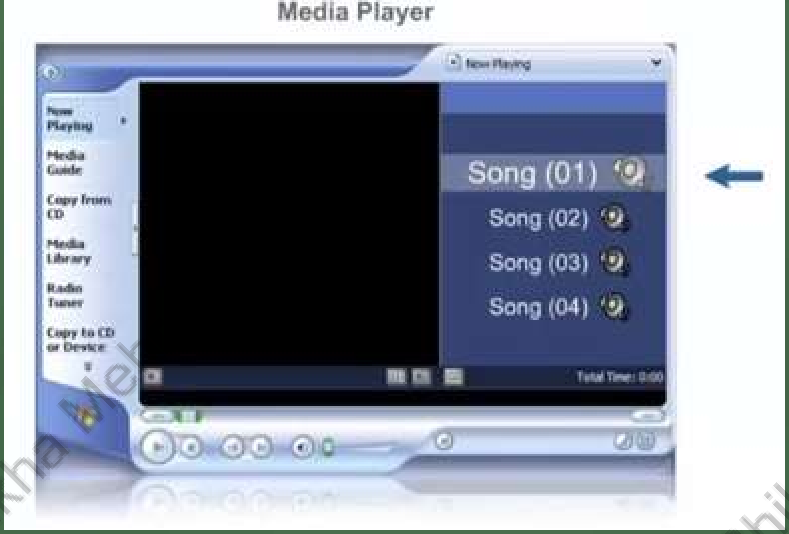
Rear of Queue
Front of Queue
next...
PAY HERE

DAC Feb 2017

Applications of Queue

Can’t You see?
I am busy

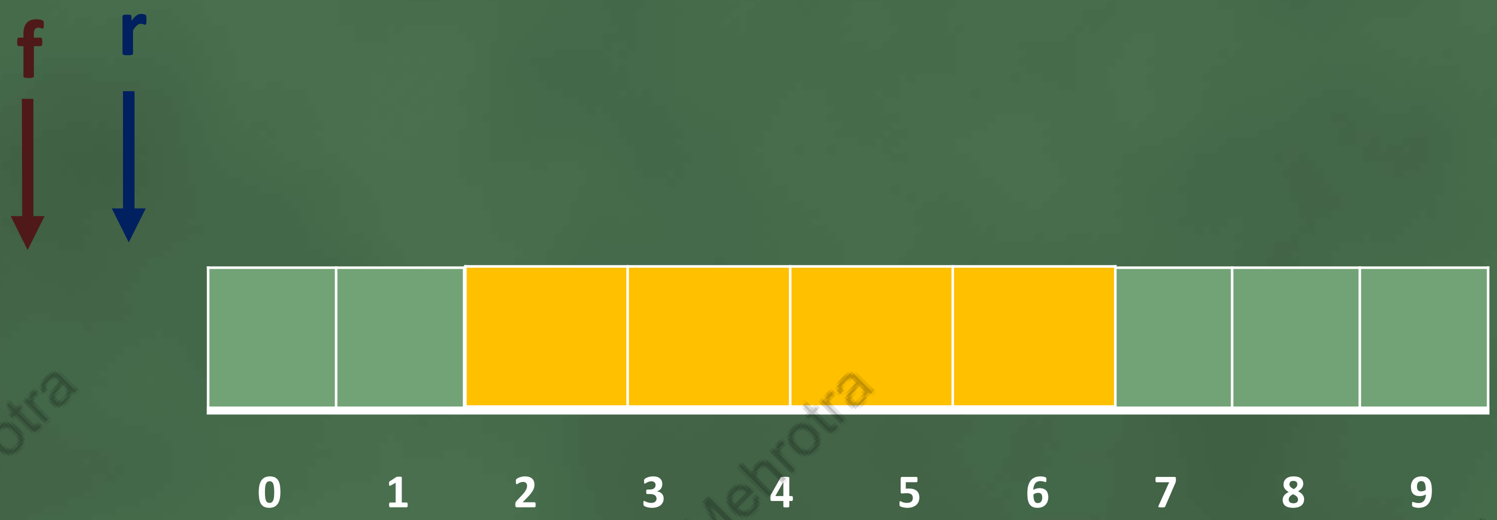
Applications of Queue



Rear of Queue
Front of Queue
next...
PAY HERE

- Serving requests of a single shared resource (printer, disk, CPU),

•Implementation of Queue

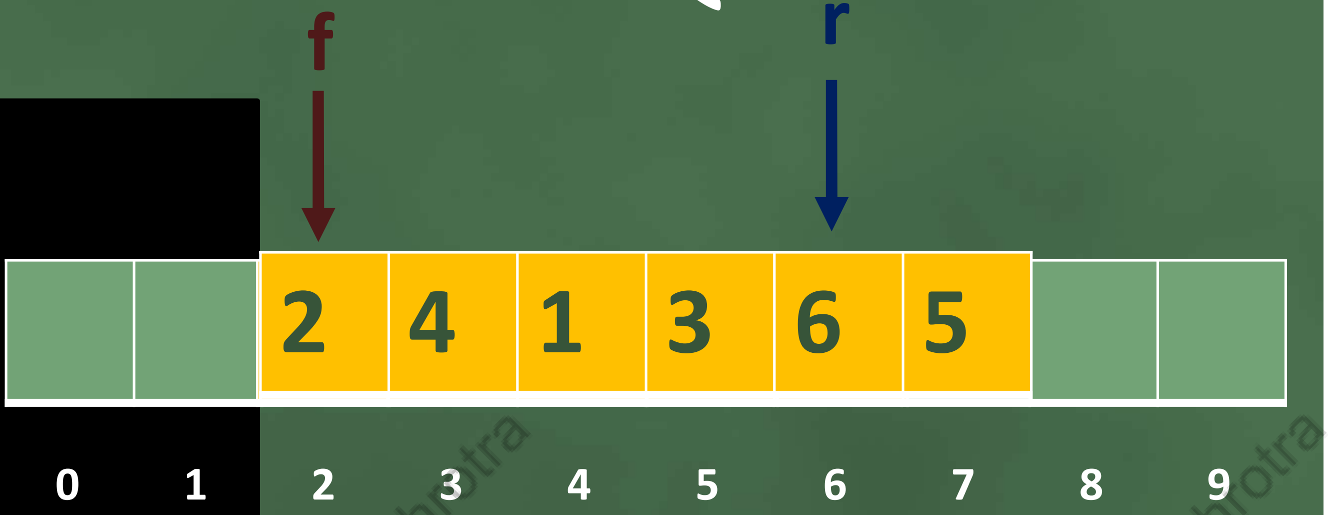


```
int A[10]
front=-1,
rear=-1
```

```
IsEmpty()
{
   if front ==-1 && rear ==-1
      return true
   else
       retrun false
}
```

•Implementation of Queue

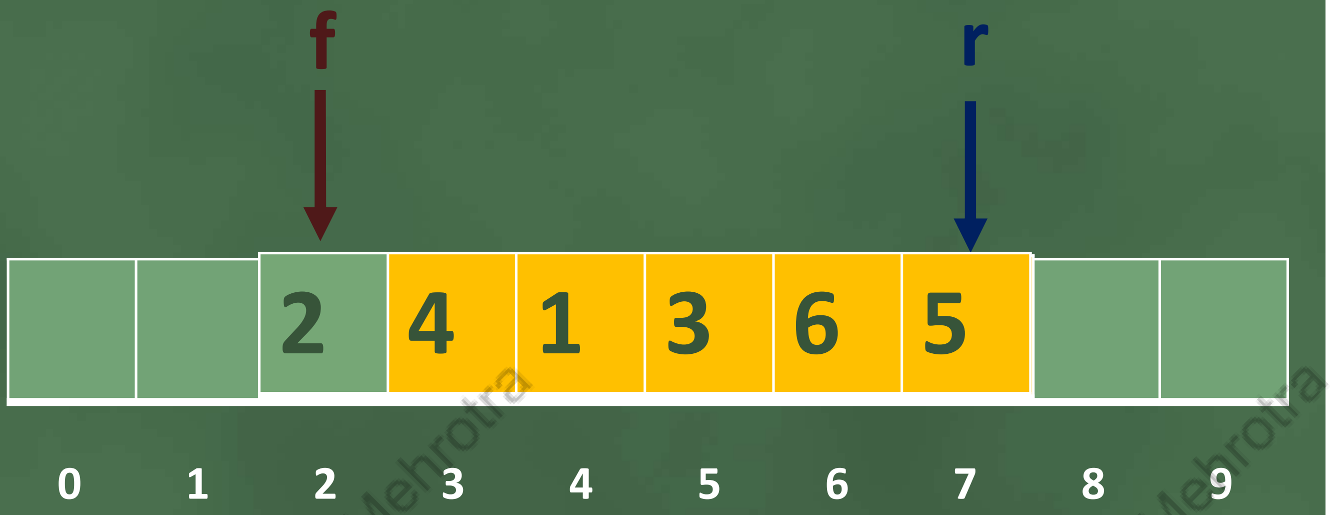
```
Enqueue(x)
{
   if IsFull()
         return

   else if IsEmpty()
         front=rear=0

    else
         rear=rear+1

  a[rear]=x

}
```



0	1	2	3	4	5	6	7	8	9
		2	4	1	3	6	5		

Enqueue(5)

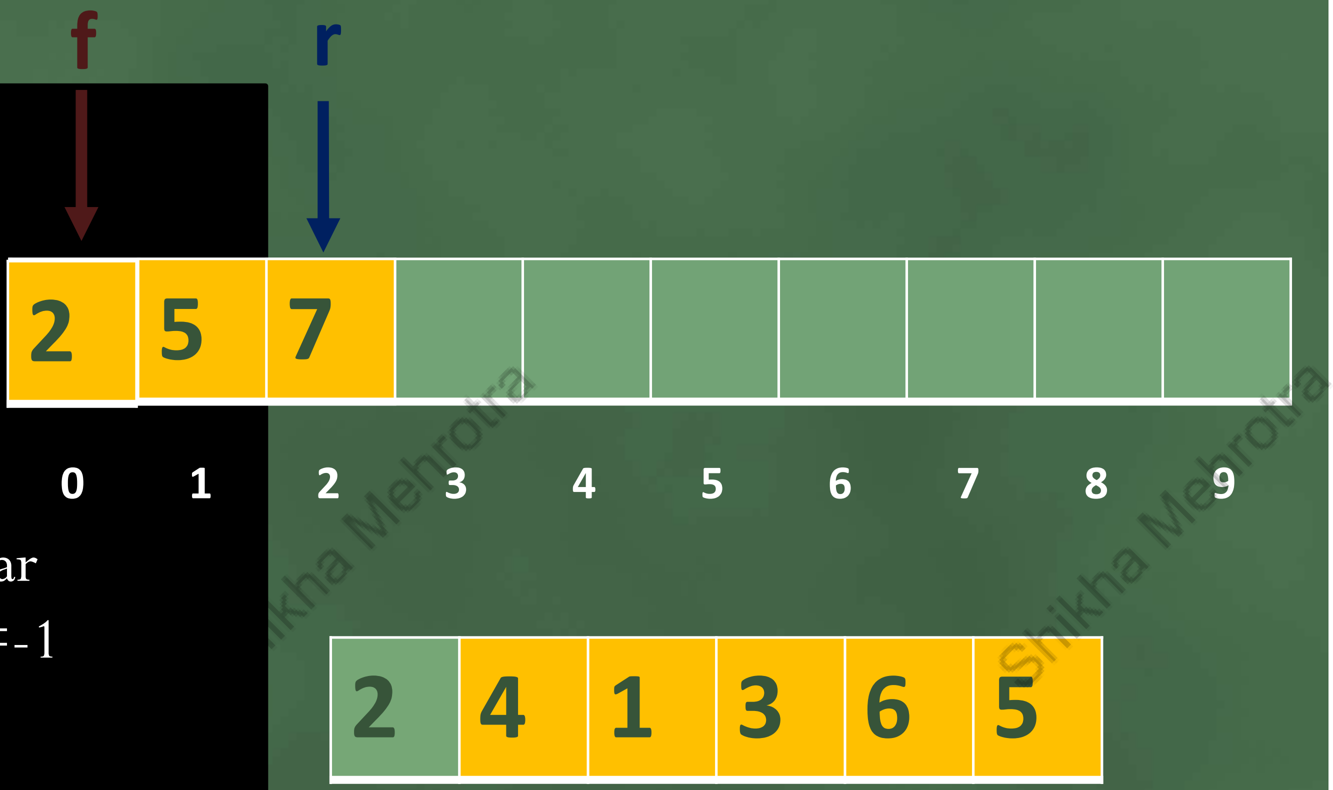
- Implementation of Queue



		f					r		
		2	4	1	3	6	5		
0	1	2	3	4	5	6	7	8	9

•Implementation of Queue

```
Deque(x)
```

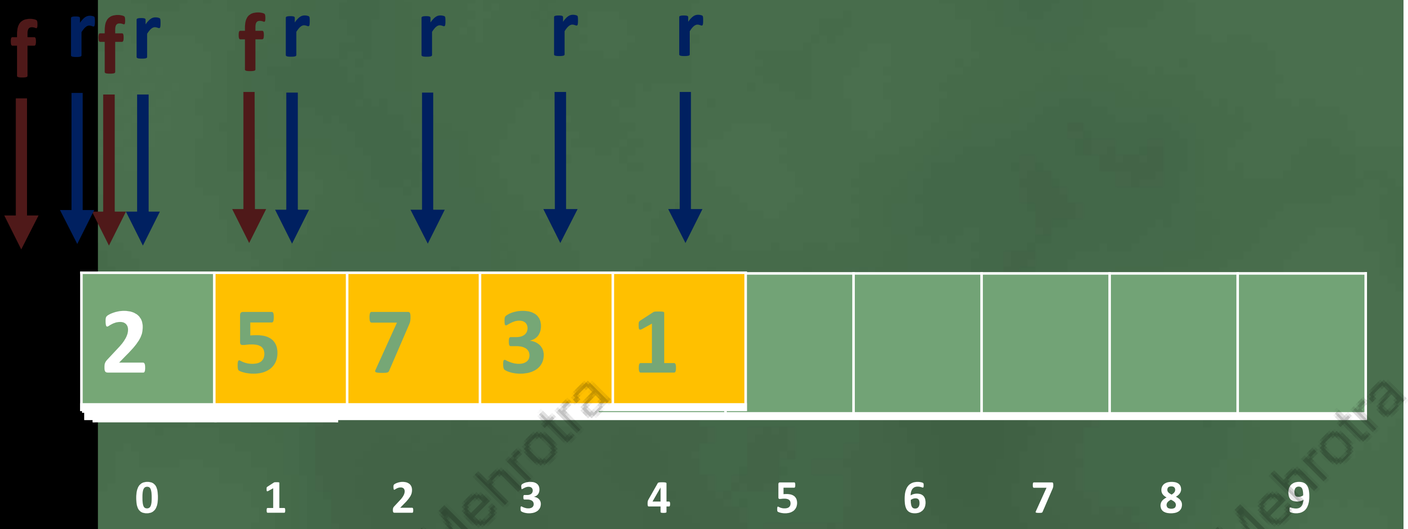


```
Dequeue(x)
{
    if IsEmpty()
            return

     else if front==rear
            front=rear=-1

      else
            front = front +1
}
```

•Implementation of Queue

```
Enqueue(x)
{
   if IsFull()
            return

   else if IsEmpty()
         front=rear=0

    else
            rear=rear+1

  a[rear]=x

}
```

```
Dequeue(x)
{
  if IsEmpty()
        return

   else if front==rear
          front=rear=-1

    else
          front = front +1
}
```



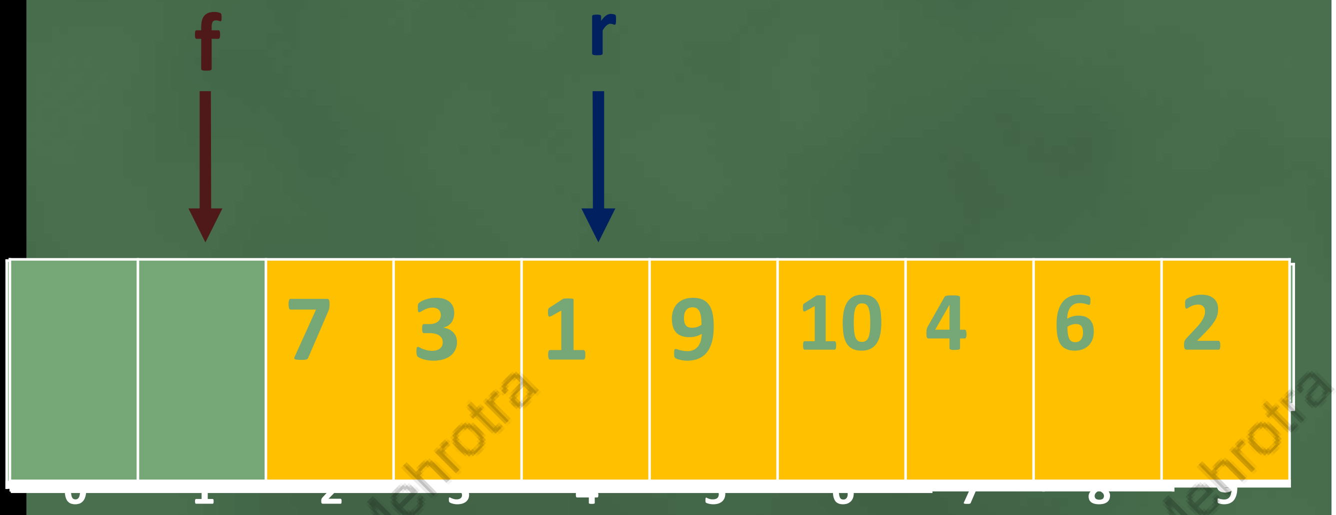
0	1	2	3	4	5	6	7	8	9
2	5	7	3	1					

Enqueue(2)
Enqueue(5)
Enqueue(7)
Dequeue()
Enqueue(3)
Enqueue(1)

•Implementation of Queue

```
Enqueue(x)
{
   if IsFull()
              return

   else if IsEmpty()
          front=rear=0

    else
              rear=rear+1

  a[rear]=x

}
```

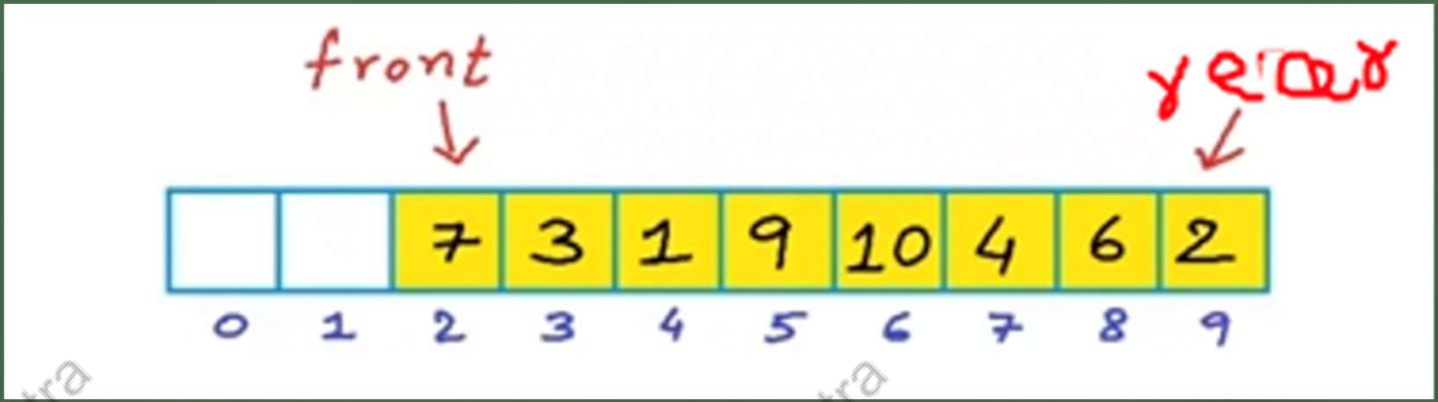
```
Dequeue(x)
{
   if IsEmpty()
          return

    else if front==rear
              front=rear=-1

     else
              front = front +1
}
```



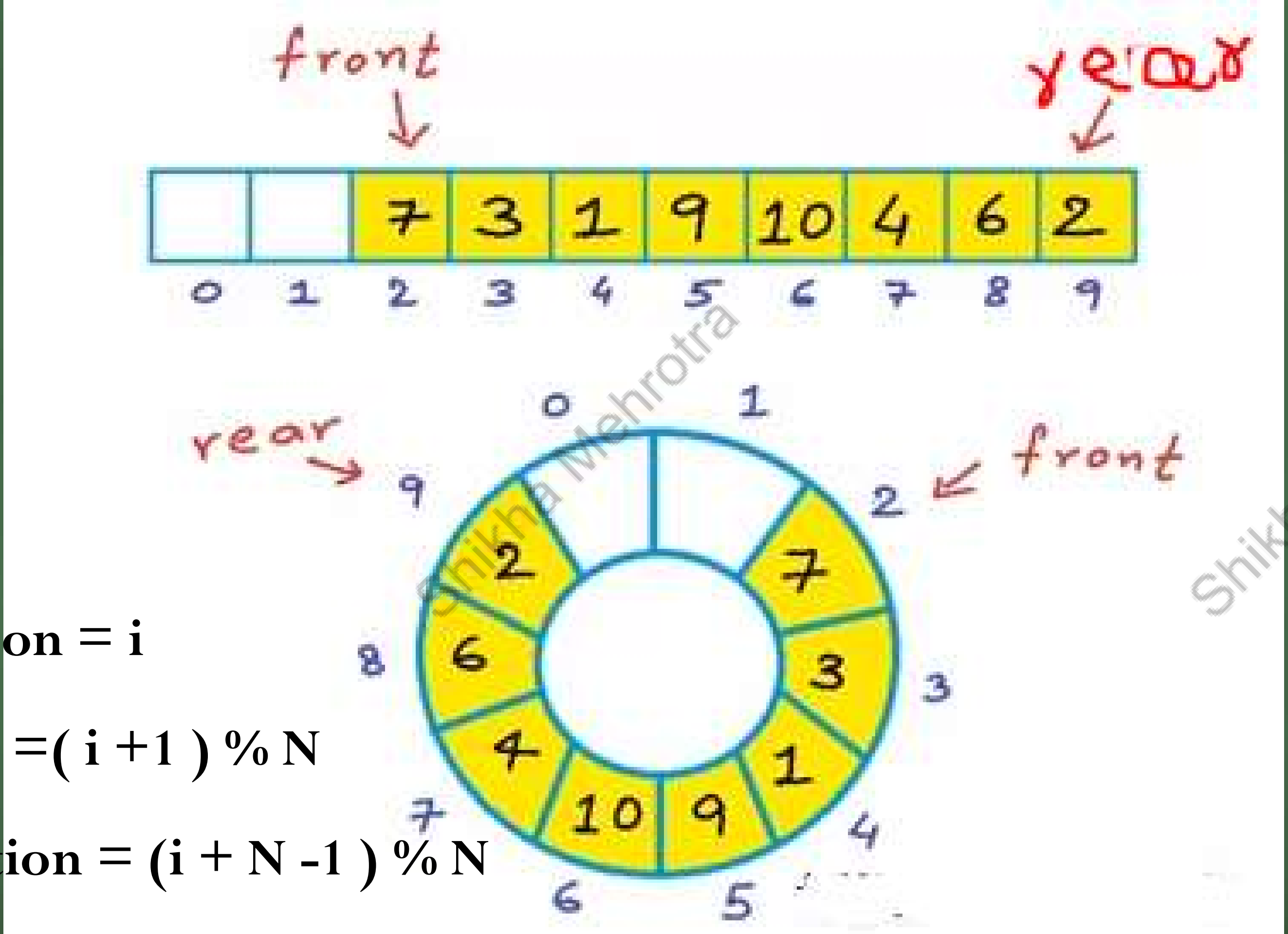
Enqueue(2)
Enqueue(5)
Enqueue(7)
Dequeue()
Enqueue(3)
Enqueue(1)

Enqueue(9)
Enqueue(10)
Enqueue(4)
Enqueue(6)
Dequeue()
Enqueue(2)

- **CIRCULAR QUEUE**


front
rear
7
3
1
9
10
4
6
2
0
1
2
3
4
5
6
7
8
9

• CIRCULAR QUEUE



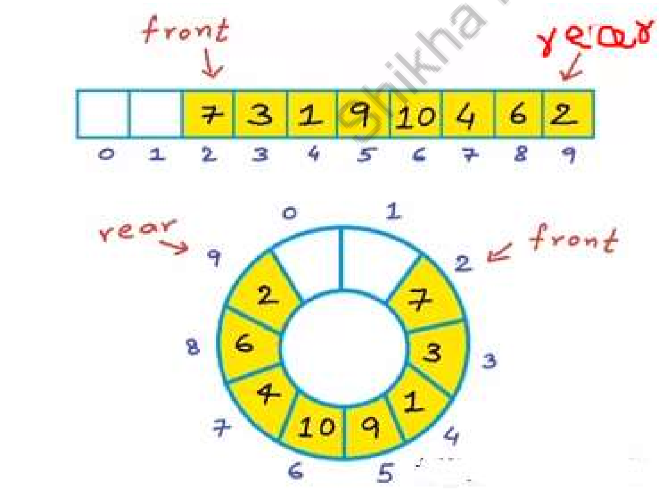
		7	3	1	9	10	4	6	2
0	1	2	3	4	5	6	7	8	9

Current Position = i

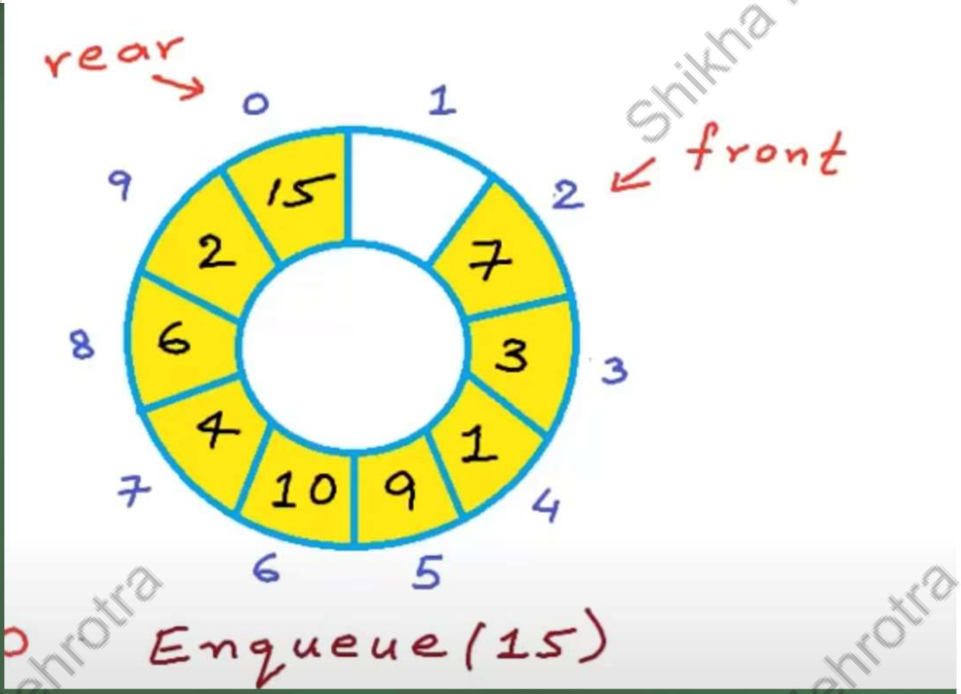
Next Position =(i +1) % N

Previous position = (i + N -1) % N

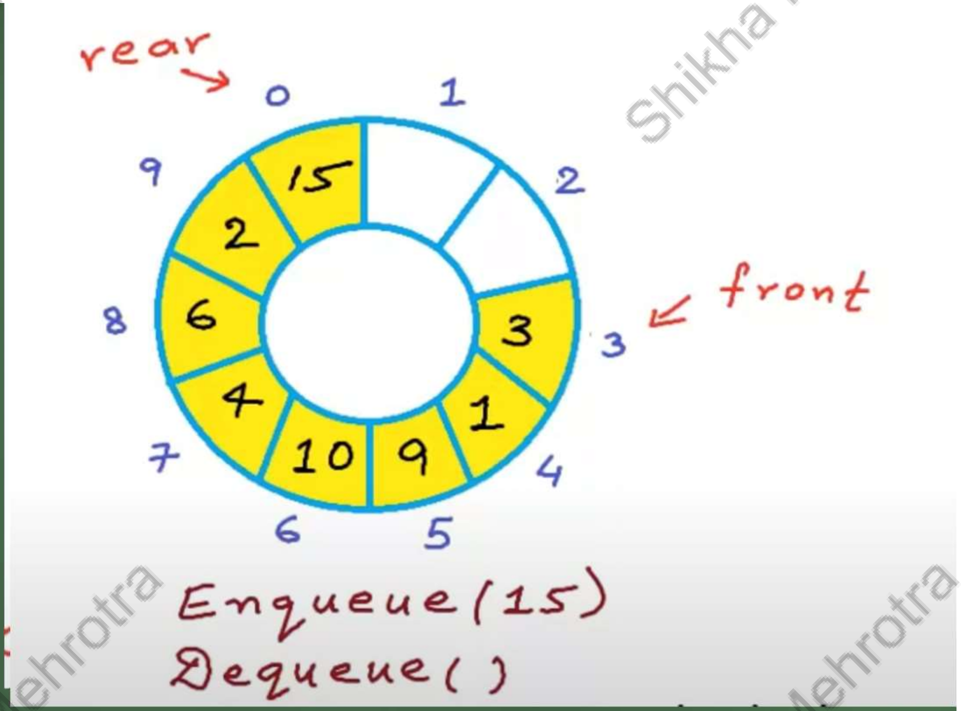
```
IsEmpty()
{
    if front ==-1 && rear ==-1
        return true
    else
        retrun false
}
```



```
Enqueue(x)
{
   if  ( rear + 1 ) % N == front


   else if IsEmpty()
            front=rear=0


    else
             rear= ( rear + 1 ) % N


  a[rear]=x


}
```



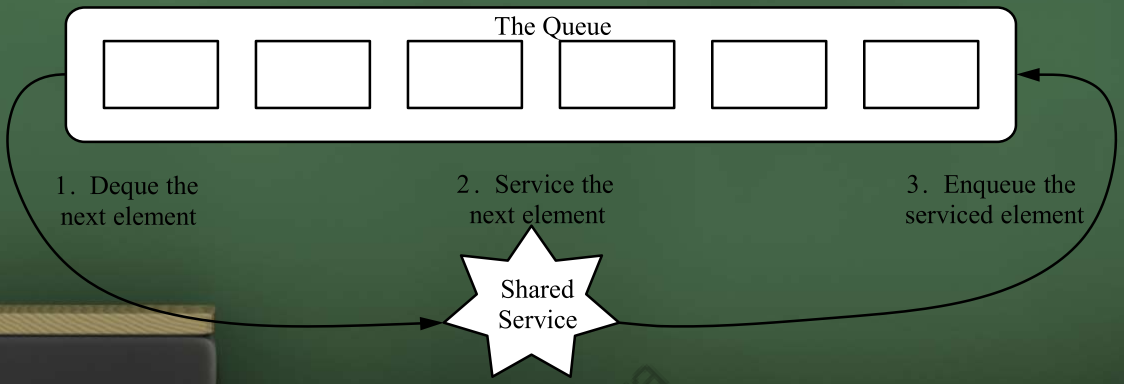
```
Dequeue(x)
{
   if IsEmpty()
         return

    else if front==rear
          front=rear=-1

    else
      front =(  front + 1 ) % N
}
```



```
#include<stdio.h>
#define n 5
int main()
{
   int queue[n],ch=1,front=0,rear=0,i,j=1,x=n;
   printf("Queue using Array");
   printf("\n1.Insertion \n2.Deletion \n3.Display
\n4.Exit");
   while(ch)
   {
      printf("\nEnter the Choice:");
      scanf("%d",&ch);
      switch(ch)
      {
      case 1:
         if(rear==x)
            printf("\n Queue is Full");
         else
         {
            printf("\n Enter no %d:",j++);
            scanf("%d",&queue[rear++]);
         }
         break;
case 2:
         if(front==rear)
         {
            printf("\n Queue is empty");
         }
         else
         {
            printf("\n Deleted Element is %d",queue[front++]);
            x++;
         }
         break;

case 3:
         printf("\nQueue Elements are:\n ");
         if(front==rear)
            printf("\n Queue is Empty");
         else
         {
            for(i=front; i<rear; i++)
            {
               printf("%d",queue[i]);
               printf("\n");
            }
            break;
         case 4:
            exit(0);
         default:
            printf("Wrong Choice: please see the options");
         }
      }
   }
   return 0;
}
```

Application: Round Robin Schedulers

- We can implement a round robin scheduler using a queue, *Q*, by repeatedly performing the following steps:
  1. *e* = *Q*.dequeue()
  2. Service element *e*
  3. *Q*.enqueue(*e*)



The Queue

1. Deque the next element

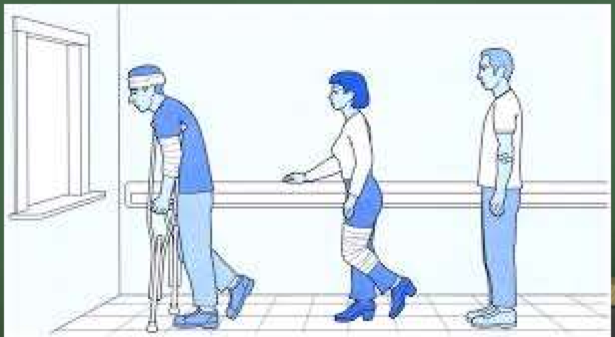
2. Service the next element

3. Enqueue the serviced element

Shared Service

• Priority Queue

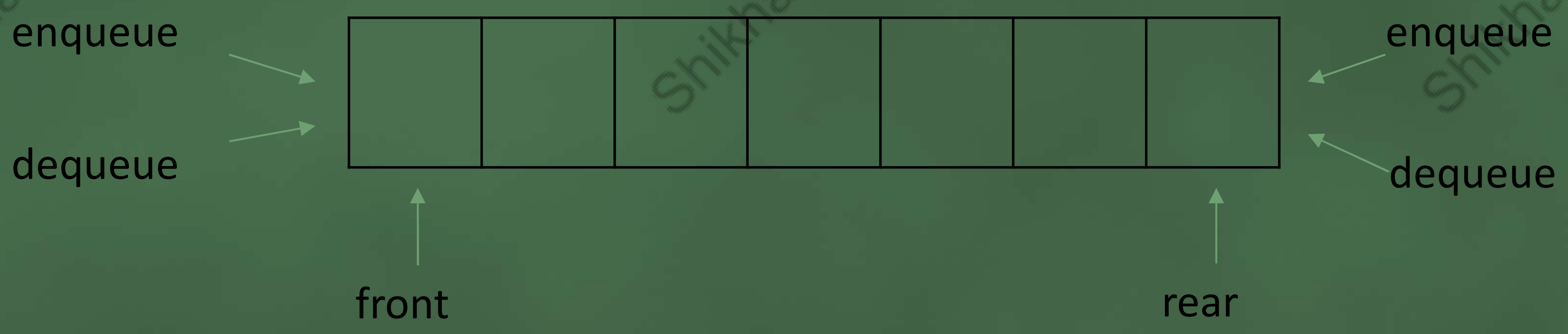
- In Priority queue items are ordered by key value so that item with the lowest value of key is at front and item with the highest value of key is at rear or vice versa.
- So we're assigned priority to item based on its key value.
- Lower the value, higher the priority.



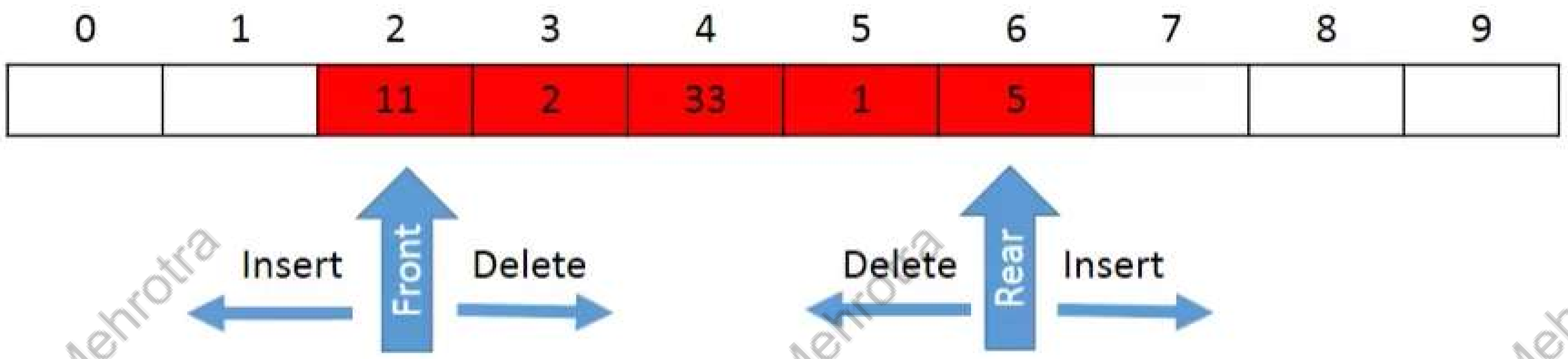
- **Priority Queue Abstract Data Type**
- Two fundamental methods:
  - enqueue
  - dequeue
- Supporting menthods
  - PeekMin
  - removeMin
  - isFull – check if queue is full.
  - isEmpty – check if queue is empty.

• Double Ended Queue (Deque)

Double Ended Queue is also a Queue data structure in which the insertion and deletion operations are performed at both the ends (front and rear).



• Functions of Double Ended Queue (Deque)



0	1	2	3	4	5	6	7	8	9
		11	2	33	1	5			

Four functions possible:

- **Insertion at front**
- **Deletion at front**
- **Insertion at Rear**
- **Deletion at Rear**

• Deque Abstract Data Type

- Deque: it creates a new deque that is empty. It needs no parameters and returns an empty deque.
- Fundamental Methods:
  - addFront(item) adds a new item to the front of the deque. It needs the item and returns nothing.
  - addRear(item) adds a new item to the rear of the deque. It needs the item and returns nothing.
  - removeFront() removes the front item from the deque. It needs no parameters and returns the item. The deque is modified.
  - removeRear() removes the rear item from the deque. It needs no parameters and returns the item. The deque is modified.
- Supporting Methods
  - isEmpty() tests to see whether the deque is empty. It needs no parameters and returns a boolean value.
  - size() returns the number of items in the deque. It needs no parameters and returns an integer.

• Deque Operations

Queue Operation	Queue Contents	Return Value
q.isEmpty()	[]	true
q.enqueue(4)	[4]	
q.enqueue(10)	[10][4]	
q.enqueue(5)	[5][10][4]	
q.size()	[5][10][4]	3
q.isEmpty()	[5][10][4]	false
q.enqueue(9)	[9][5][10][4]	
q.dequeue()	[9][5][10]	
q.dequeue()	[9][5]	
q.size()	[9][5]	2