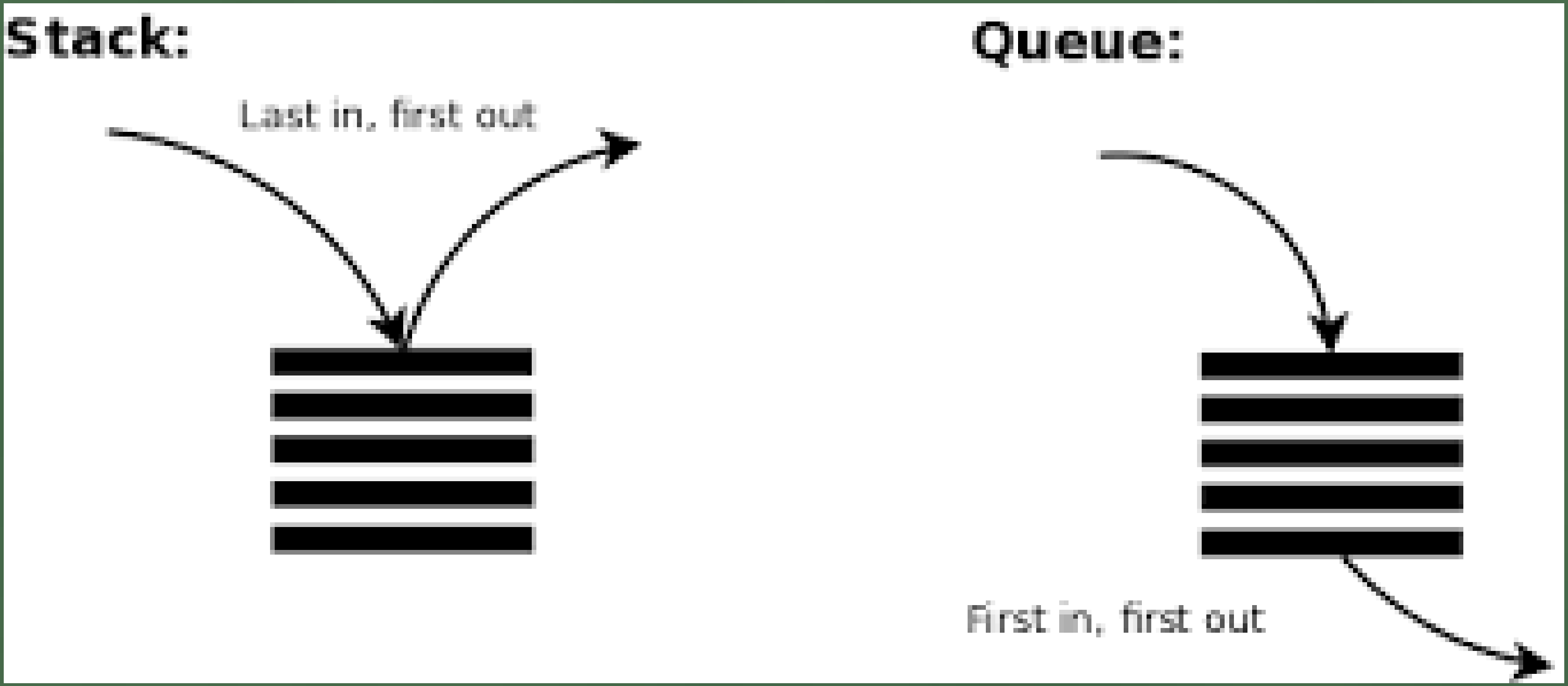
Data Structures and Algorithms

Shikha Mehrotra

The Queue

Shikha Mehrotra

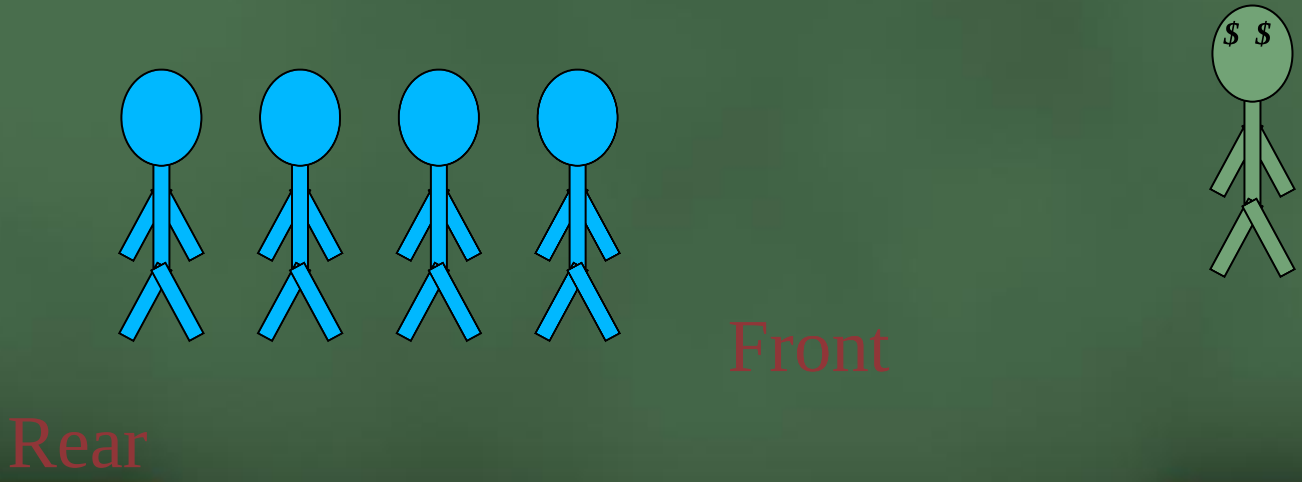

Stack:
Last in, first out
Queue:
First in, first out

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.

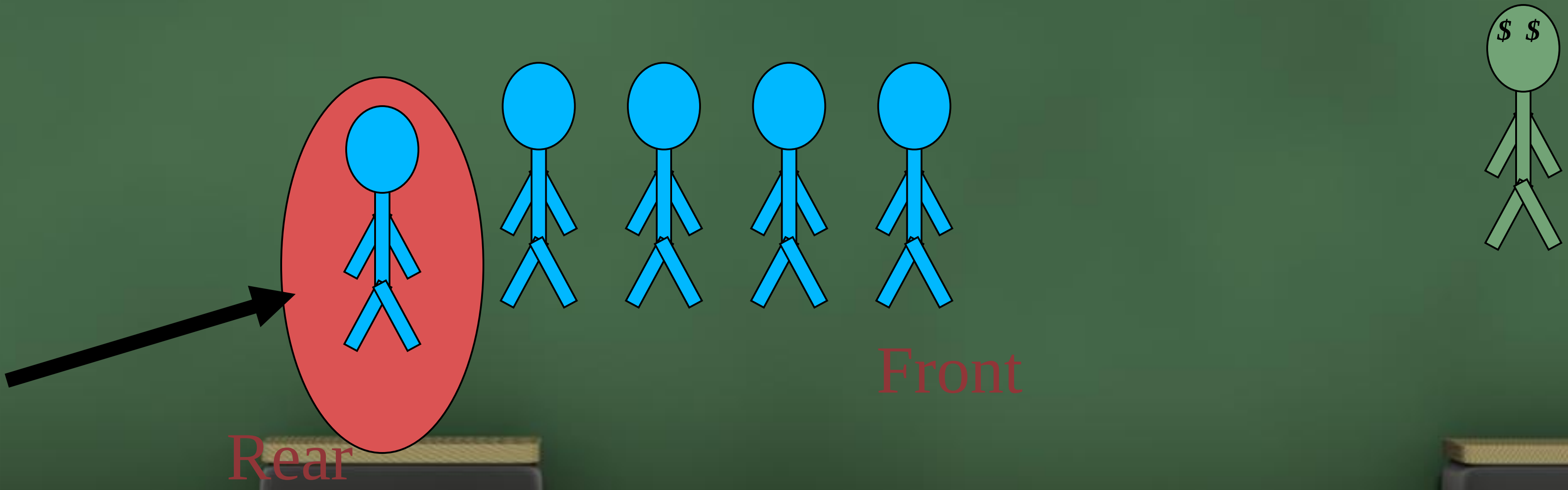
•The Queue Operations

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



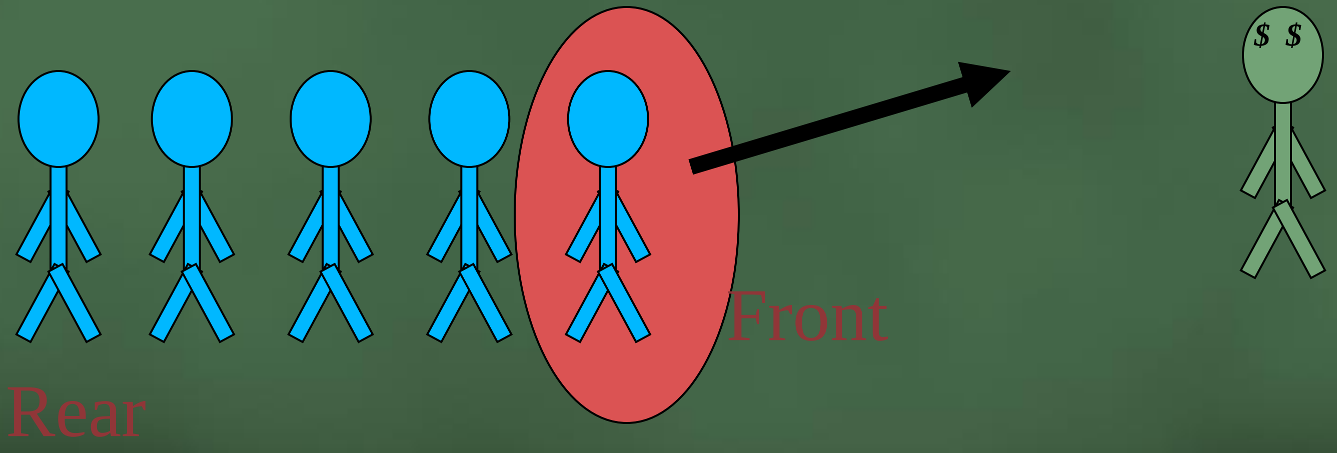
•The Queue Operations

- 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

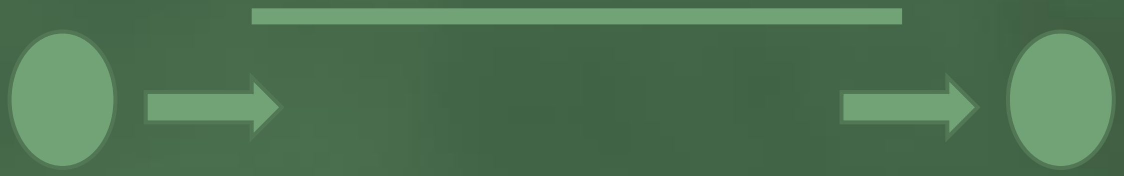
- When an item is taken from the queue, it always comes from the front. The C++ queue calls this a **pop**, although it is usually called a **dequeue** operation.



•Queue ADT

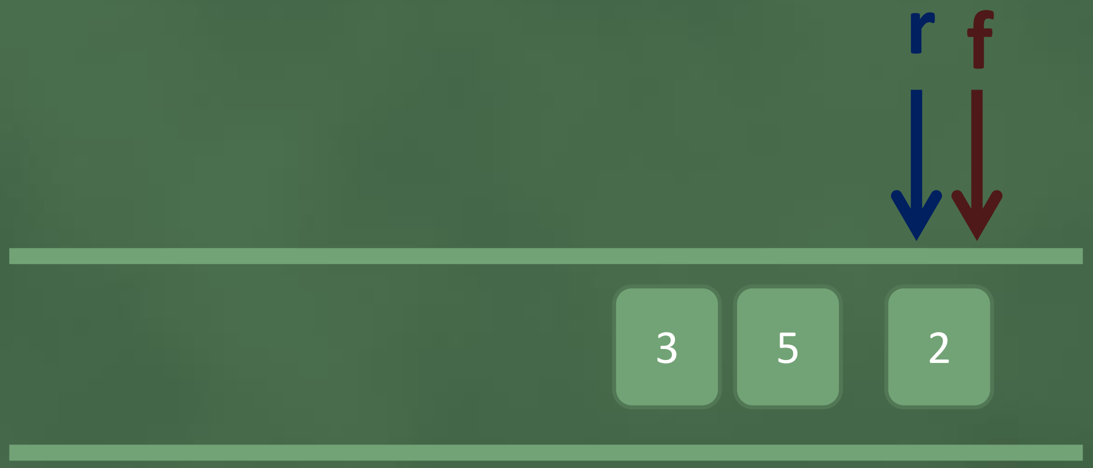
```
AbstractDataType queue {
  instances
          ordered list of elements; one end is the front; the other is the rear;
  operations
          IsEmpty():        Return true if queue is empty, return false otherwise
          size():           Return the number of elements in the queue
          front():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

- Queue



Enqueue

Dequeue



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

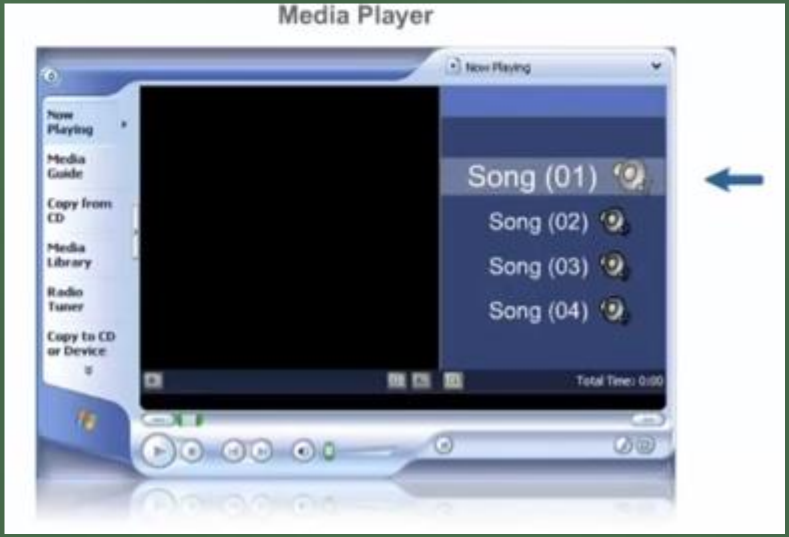
Applications of Queue

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

Applications of Queue

Can’t You see?
I am busy

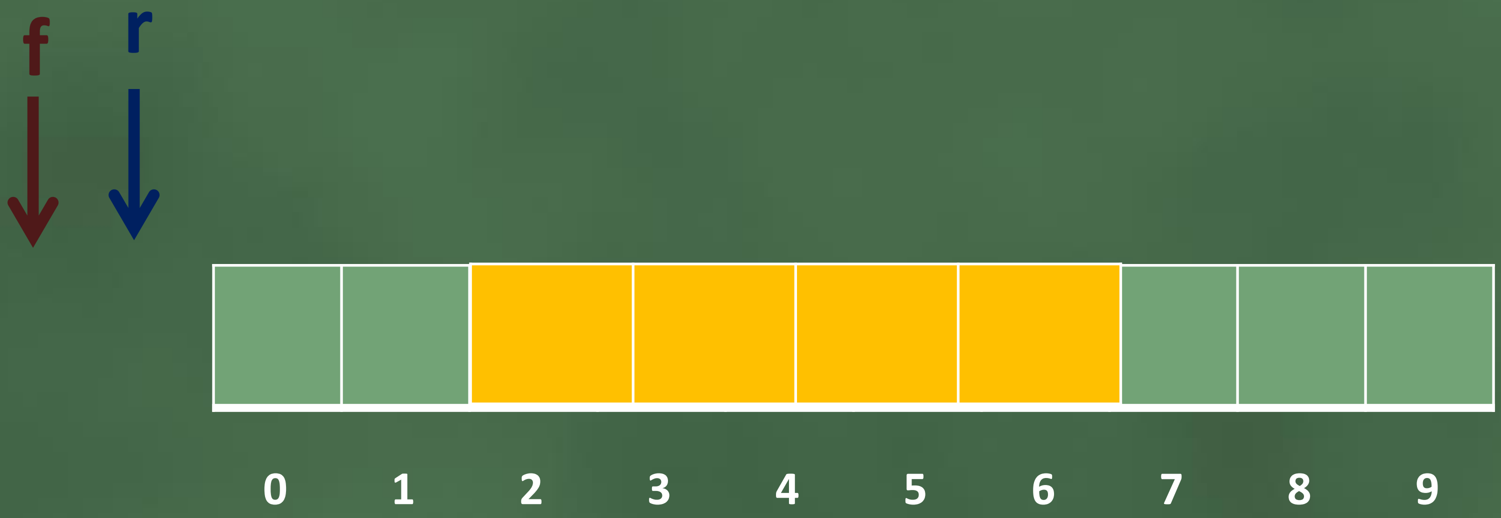
Applications of Queue



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

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

Implementation of Queue

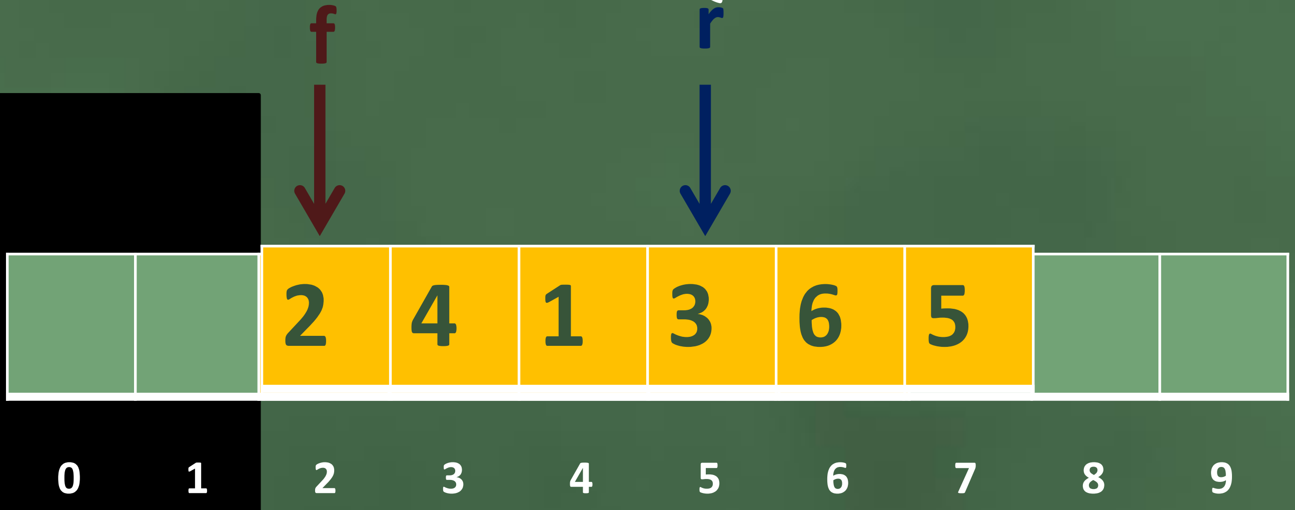


```
int A[10]
front=-1,
rear=-1
```

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

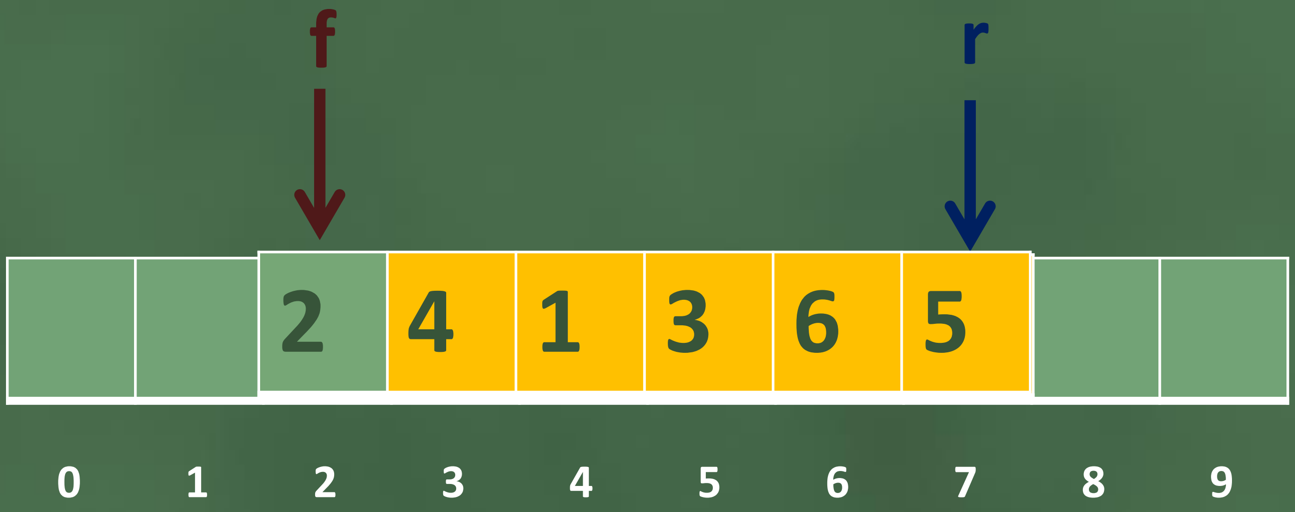
Implementation of Queue

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

    else if IsEmpty()
            front=rear=0

    else
            rear=rear+1

  a[rear]=x

}
```



Enqueue(5)

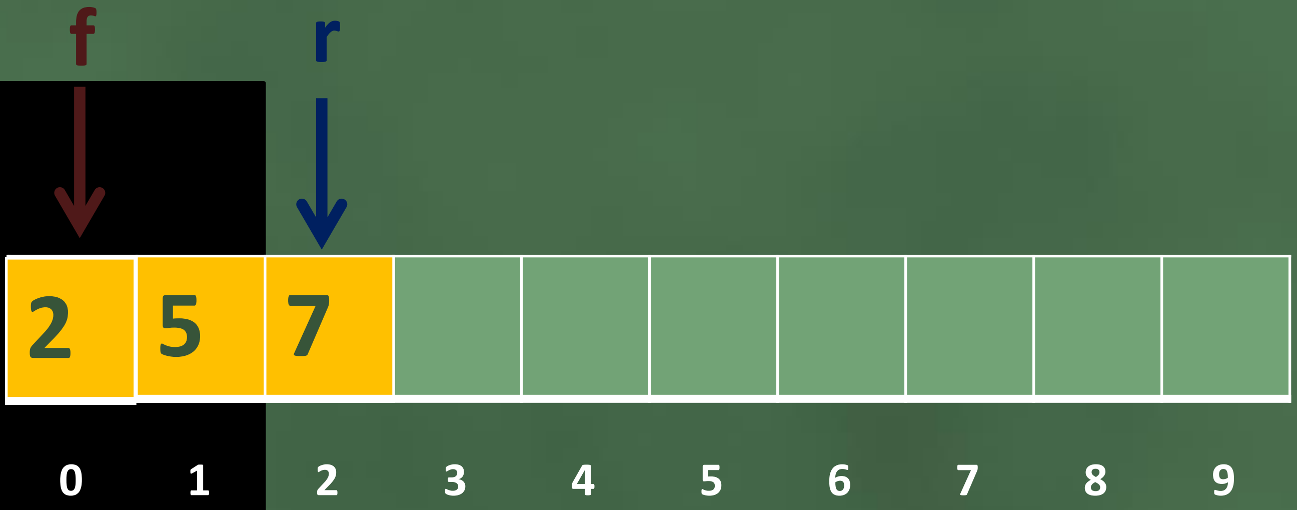
- Implementation of Queue



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

Implementation of Queue

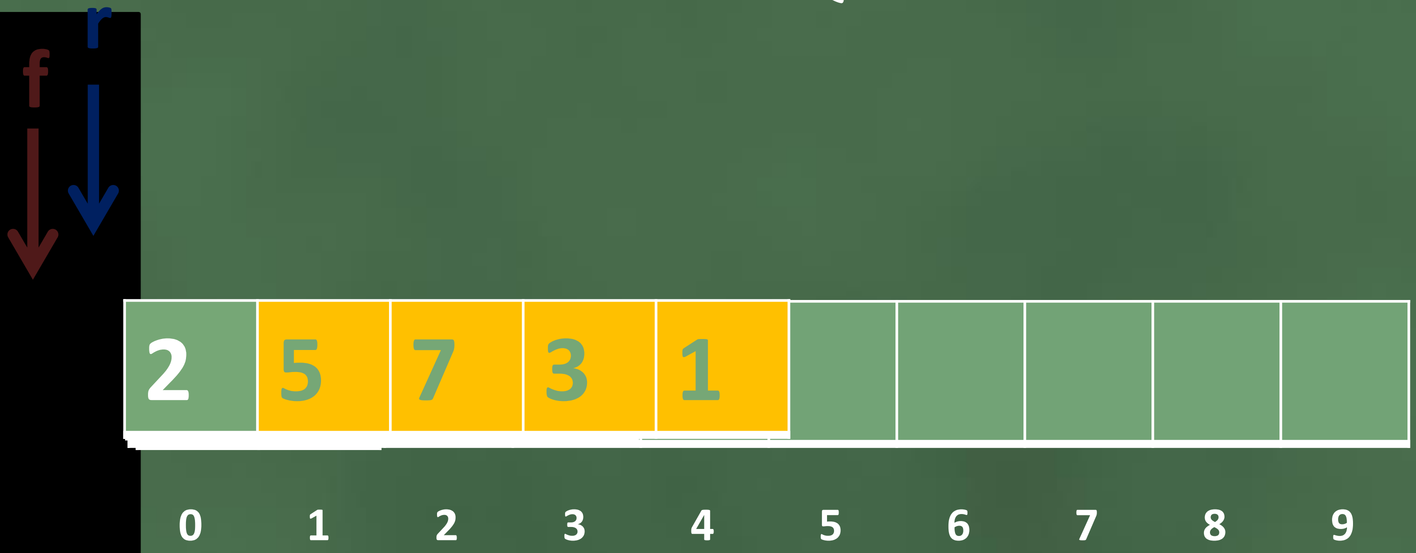
```
Dequeue()
{
   if IsEmpty()
         return

    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

}
```

```
Deqeueue()
{
  if IsEmpty()
        return

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

     else
             front = front +1
}
```



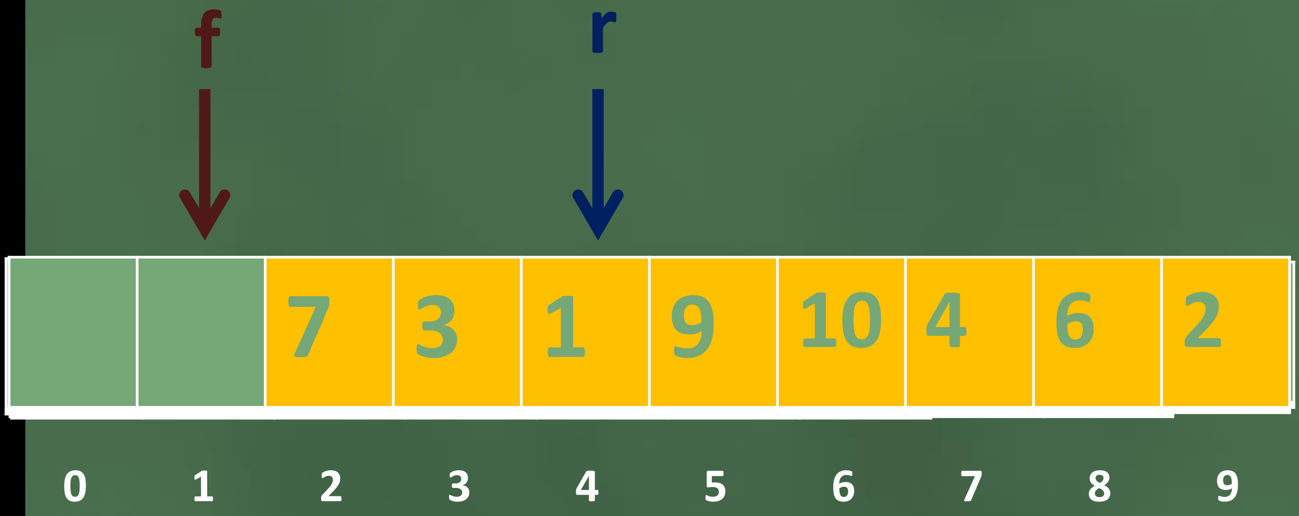
0	1	2	3	4	5	6	7	8	9
2	5	7	3	1					

Enqueue(2)
Enqueue(5)
Enqueue(7)
Deqeue()
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()
{
   if IsEmpty()
         return

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

     else
            front = front +1
}
```

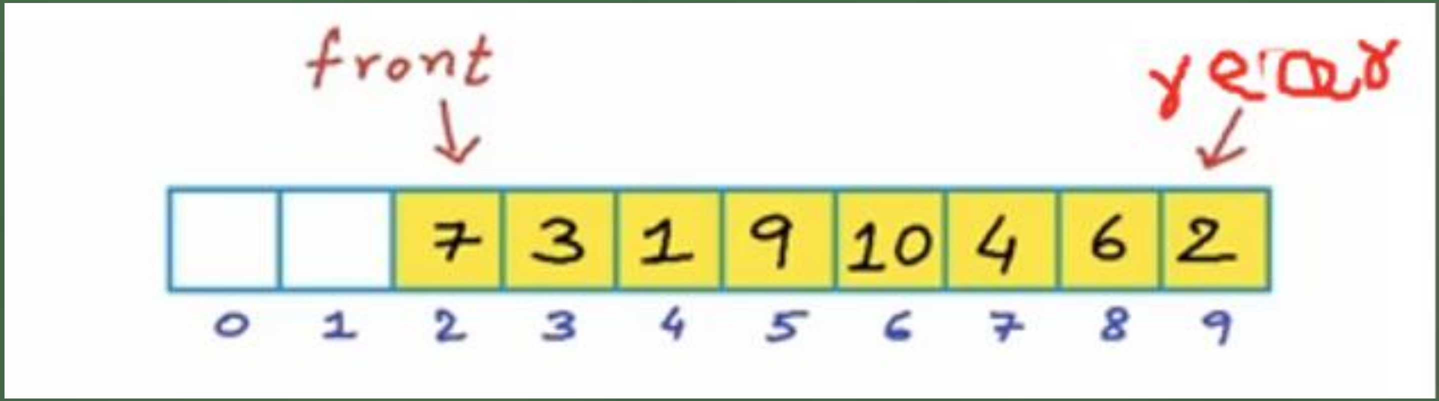


0	1	2	3	4	5	6	7	8	9
		7	3	1	9	10	4	6	2

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

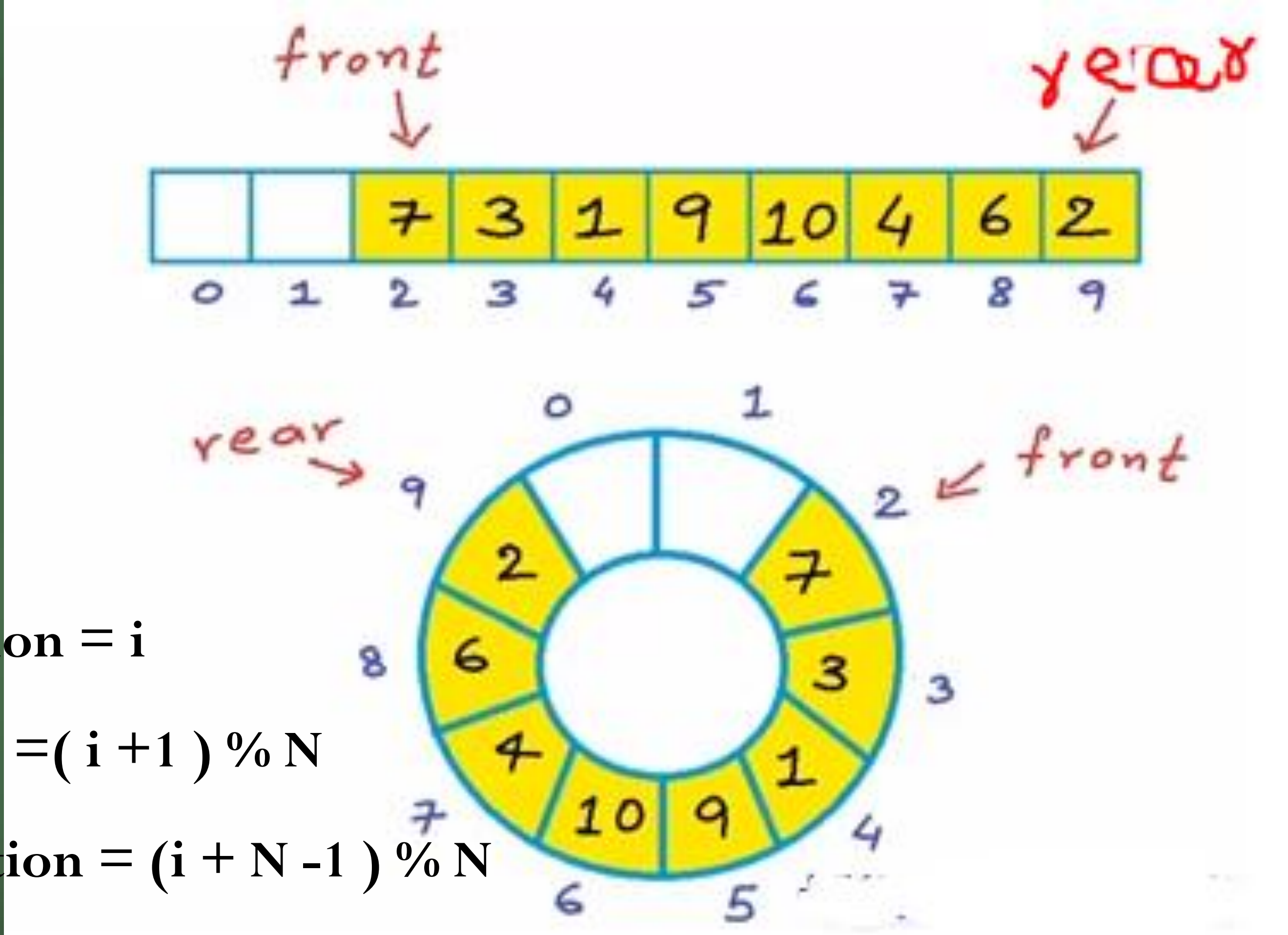
Enqueue(9)
Enqueue(10)
Enqueue(4)
Enqueue(6)
Dequeue()
Enqueue(2)

- CIRCULAR QUEUE



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

•CIRCULAR QUEUE

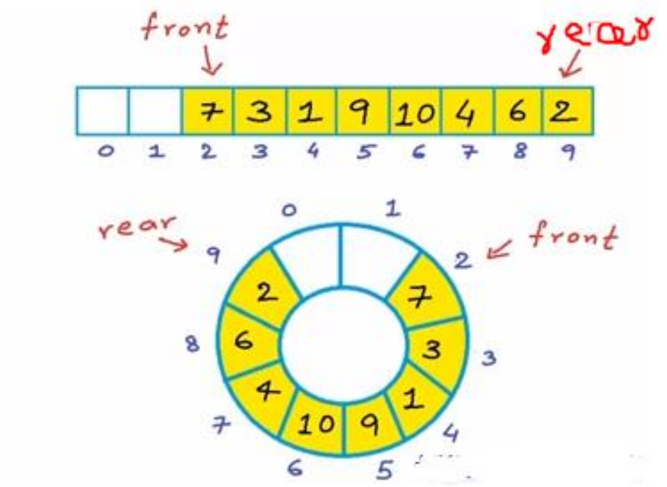


Current Position = i

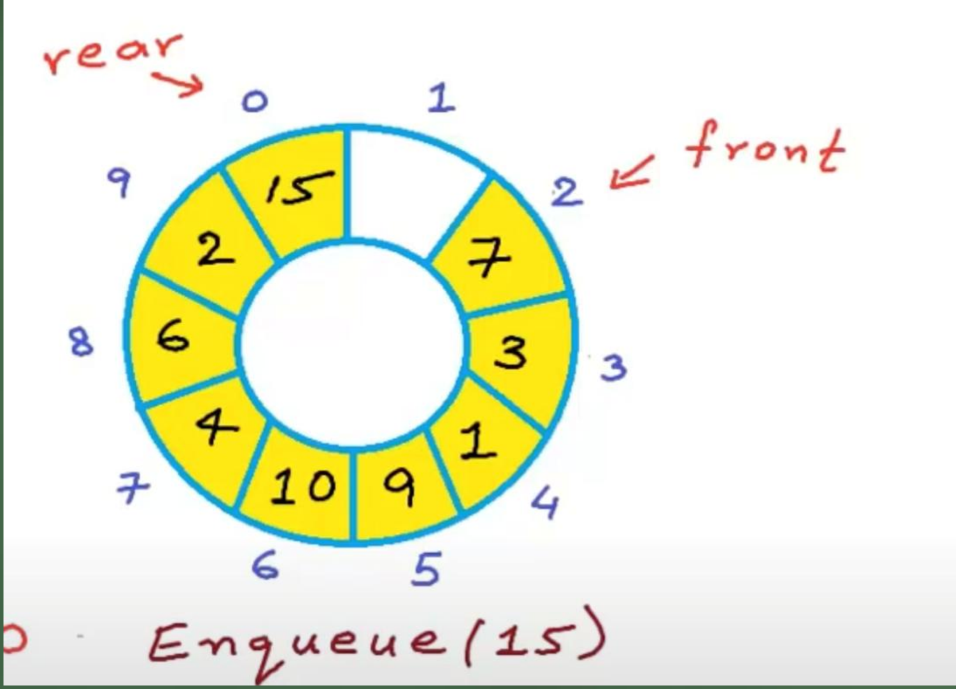
Next Position =(i +1) % N

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

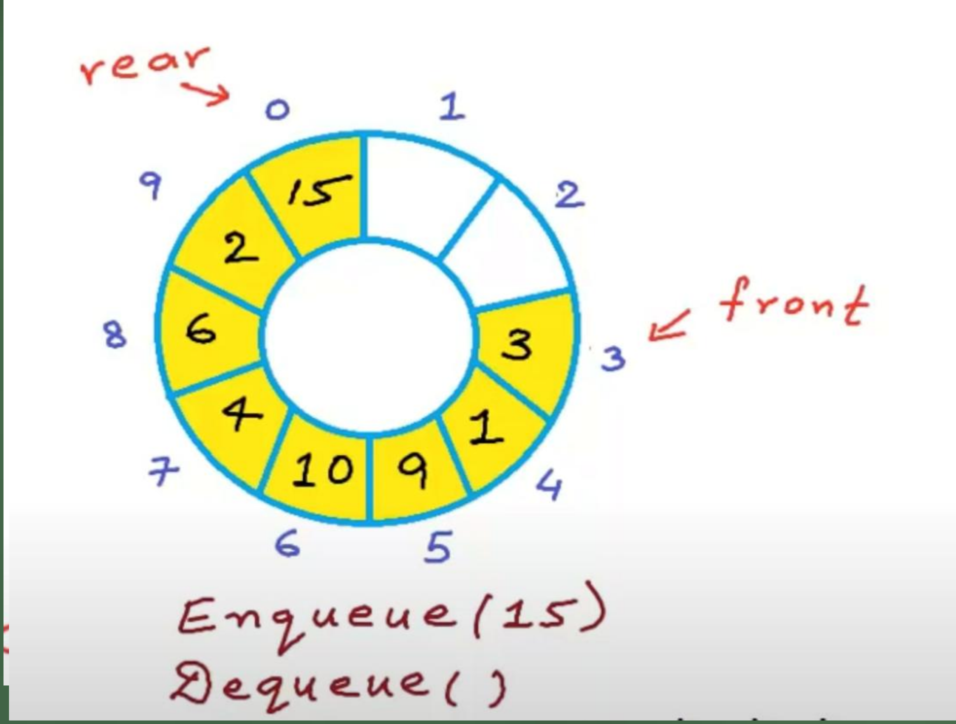
```
IsEmpty()
{
   if front ==-1 && rear ==-1
      return true
   else
      retrun false
}
```



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

   else if IsEmpty()
         front=rear=0

   else
         rear= ( rear + 1 ) % N

  a[rear]=x

}
```



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

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

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

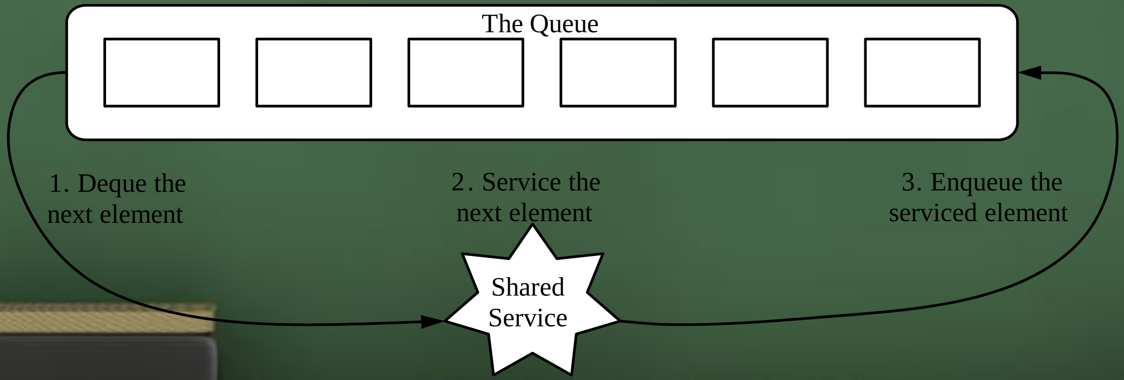


```
#include<stdio.h>
#define n 5
int main()
{
   int queue[n],ch=1,front=0,rear=0,i,j=1,x=n;
   printf("Queue using Array");
   printf("\n1.Insertion \n2.Deletion \n3.Display
\n4.Exit");
   while(ch)
   {
      printf("\nEnter the Choice:");
      scanf("%d",&ch);
      switch(ch)
      {
      case 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