Data Structures and Algorithm Analysis

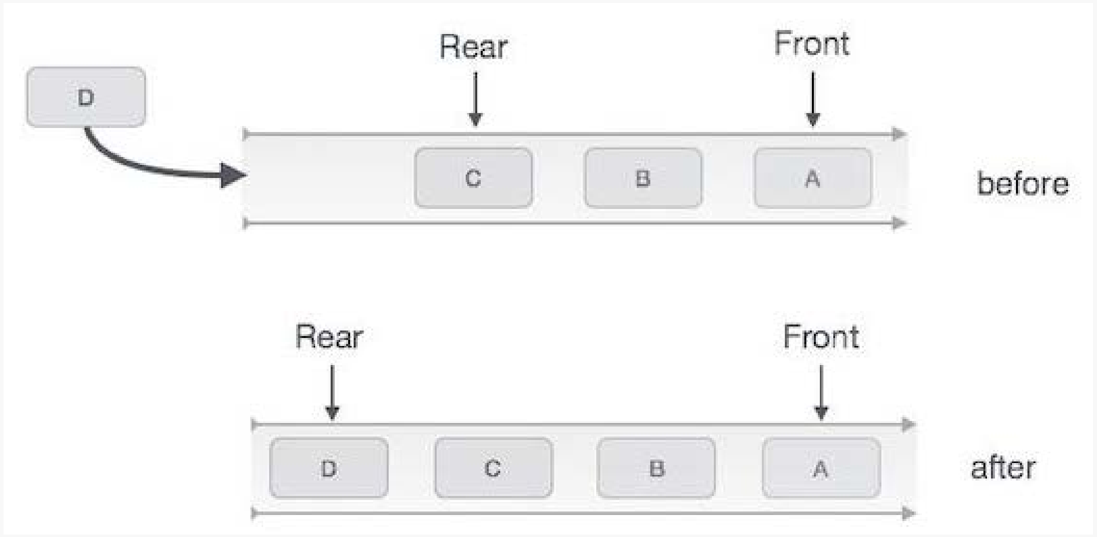
8

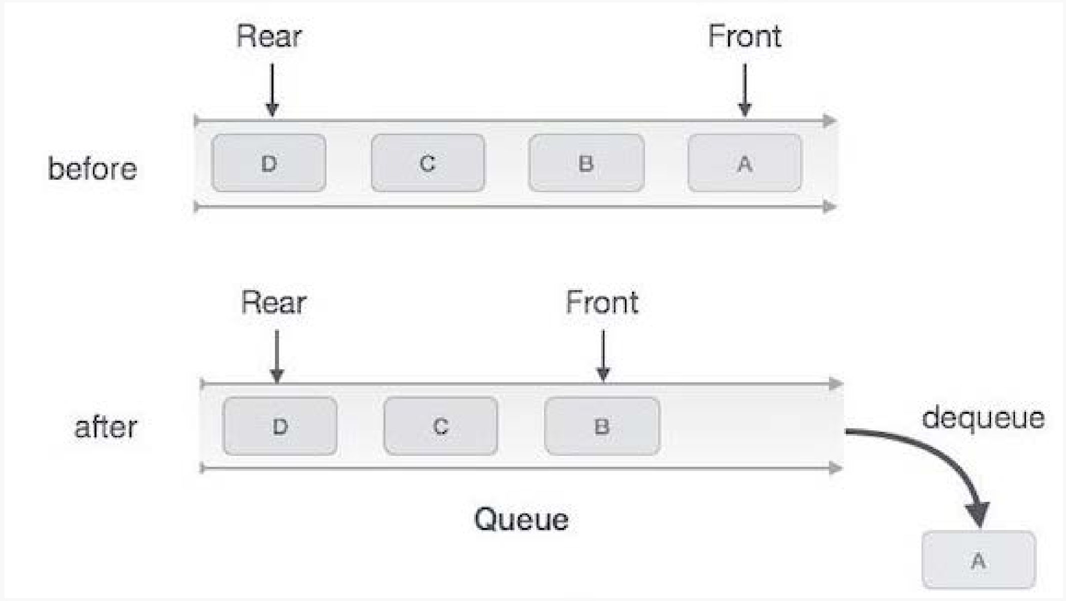
Dr. Syed Asim Jalal
Department of Computer Science
University of Peshawar

Queues

Queue

- Queue is a linear data structure.
- Queue is an ordered collection of items from which items may be removed 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.
- The first element inserted into the queue is the first element to be removed. Queue is therefore also called **FIFO (First-In First-Out)** structure.
- Queue data structured is used when data is processed in sequence


Rear
Front
D
C
B
A
before
Rear
Front
D
C
B
A
after


Rear
Front
before
D
C
B
A
Rear
Front
after
D
C
B
dequeue
Queue
A

- Everyday life examples of Queue includes:
  - People waiting in line at a bank
  - Cars passing through Toll gate

Applications in Computer systems

- Time sharing system:
  - In a time sharing system different programs are assigned to processor for execution.
  - Programs are added to Queue to wait for turns to executed.
- Managing Buffer
  - Queue implements buffers, for sending data between a fast computer and any slow device – for example Printers.

Queue Operations

- Insert(q, A):
  - *place A at the rear of the queue q*
  - *Also called enqueue()*
- Delete(q)
  - *remove the front element from q and return it.*
  - *All called dequeue()*
- IsEmpty(q)
  - *return TRUE if the queue is empty,*
  - *FALSE otherwise*

What is Queue underflow?

- Delete operation is only possible if a queue is not empty
- The condition resulting from trying to remove an element from an empty queue.
- A queue is checked for underflow before removing an element.

```
if(  !  q.IsEmpty()  )
    q.Dequeue( item );
```

What is Queue overflow?

- The condition resulting from trying to add an element onto a full queue.
- Queue is checked for overflow before add elements in that queue

```
        if(   ! q.IsFull()  )
                  q.Enqueue(item);
```

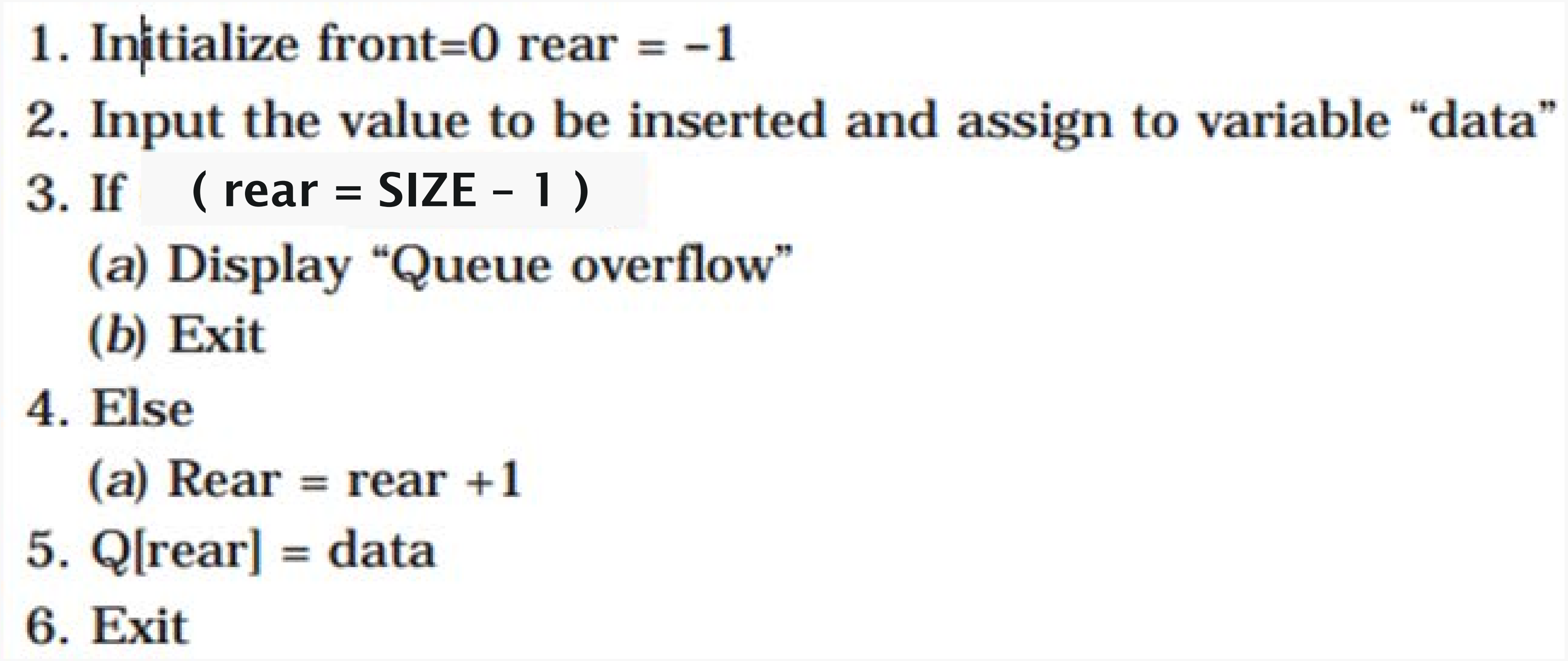
Representation of Queue

- Queue can also be implemented using both
  - **Array**
  - **Linked List**
- The decision to use Array or a Linked List depends on the requirements of the queue application.

Array Implementation

- Queue could be implemented using Array with two variables or indexes, **Front** and **Rear**.
  - Rear is initialized to -1,
  - Front is initialized to 0.
- Insert operation would look like
    - q[++rear] = x
    - // increment Rear index and assign new value
- Remove operation would like
    - X = q[front++]
    - // Assign value and increment Front index
- Queue is empty condition
    - Rear < front

- Algorithm for adding an element in a Queue would look like this:

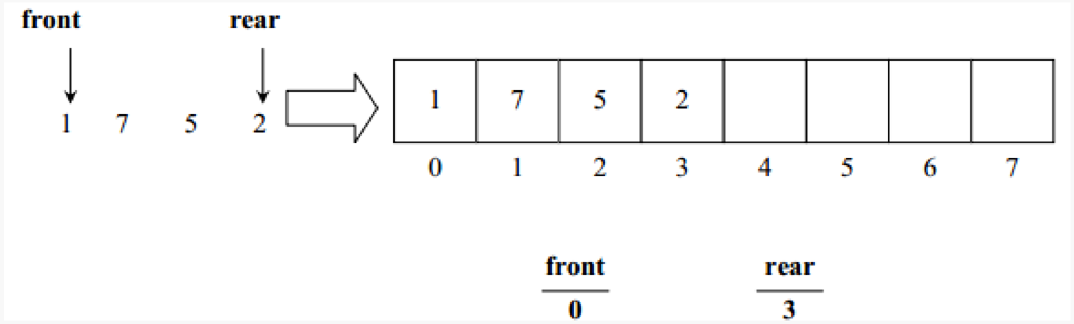


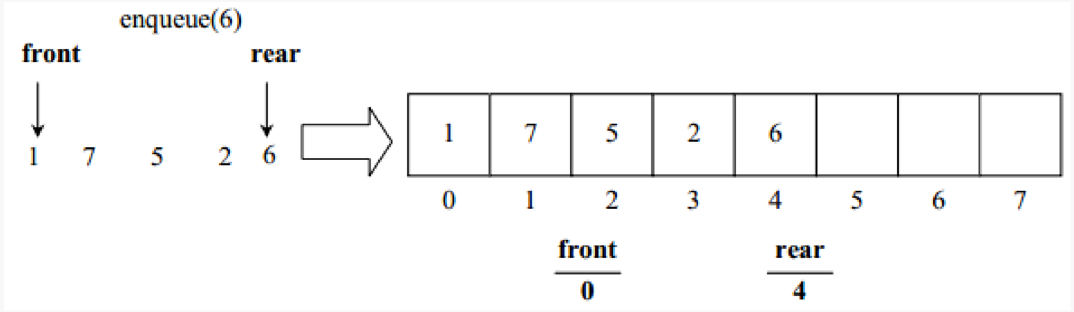
```
1. Initialize front=0 rear = -1
2. Input the value to be inserted and assign to variable "data"
3. If ( rear = SIZE – 1 )
    (a) Display "Queue overflow"
    (b) Exit
4. Else
    (a) Rear = rear +1
5. Q[rear] = data
6. Exit
```

- Algorithm for removing an element from a Queue would look like this:

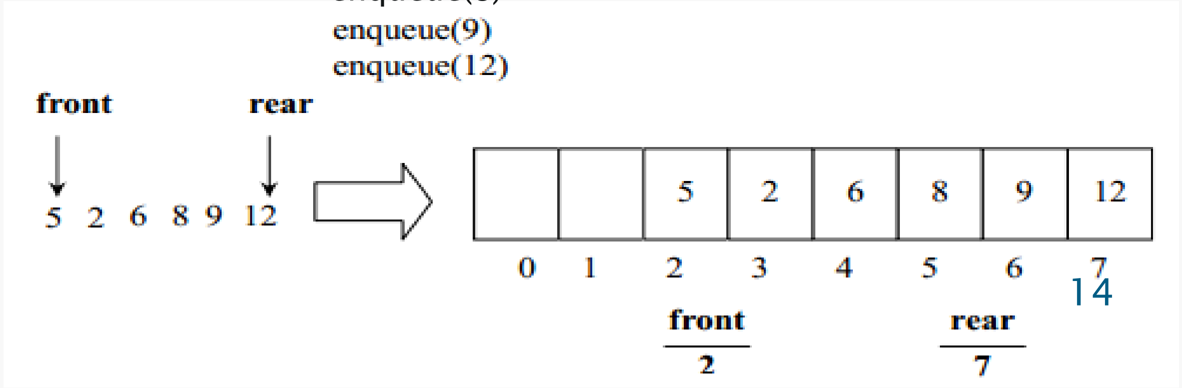
```
1. If (  rear <  front  )
      (a) Display "The queue is empty"
      (b) Exit
2. else
      (a) Data = Q[front]
3. Front = front +1
4. Exit
```

There are problems in the algorithm we identified so far





dequeue()
dequeue()
enqueue(8)

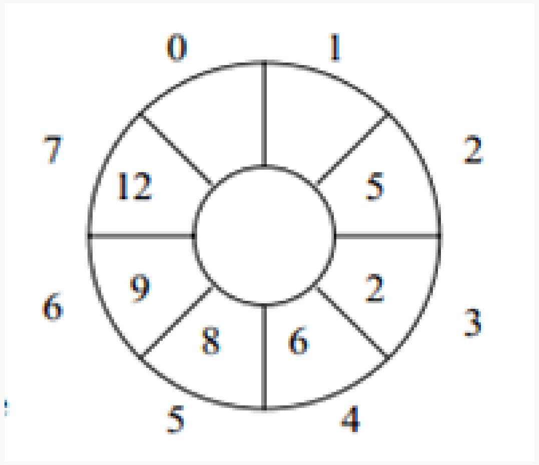


The problems are

- the array that was used to implement it, has reached its limit as the **last location of the array is in use now**. We know that there is some problem with the array after it attained the size limit. We observed the similar problem while implementing a stack with the help of an array. We can also see that two locations at the start of the array are vacant. Therefore, we should consider how to use those locations appropriately in adding more elements in the array.
- Although, we have Insert and Remove operations running in constantly, yet we created a new problem that we cannot Insert new elements even though there are two places available at the start of the array. **The solution to this problem lies in allowing the queue to wrap around in a circular way**.

What are Implementation issues to consider when implementing Queue through Circular Arrays?

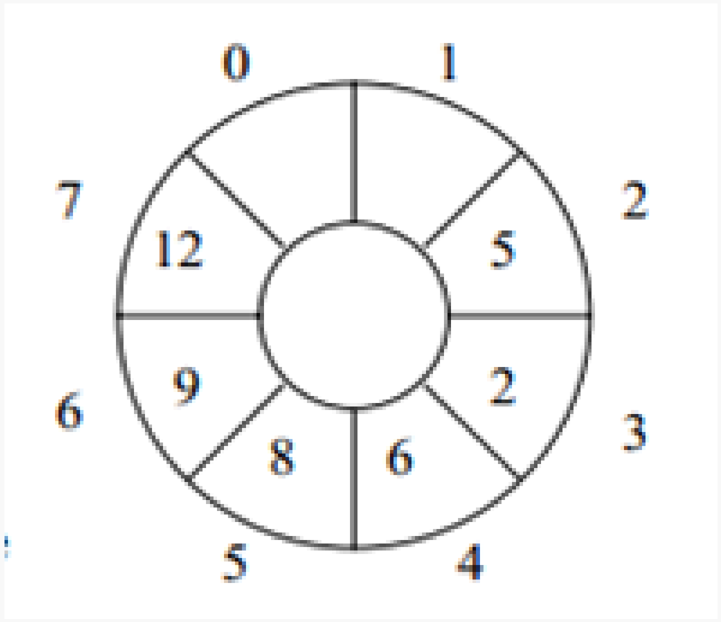
- How to reuse empty spaces in the arrays which become vacant after delete (de-queue operations)?
- How do we know if a queue is full or empty based on the values of **Front** and **Rear** indexes?
- What values should be used to initialize *front* and *rear* indexes?

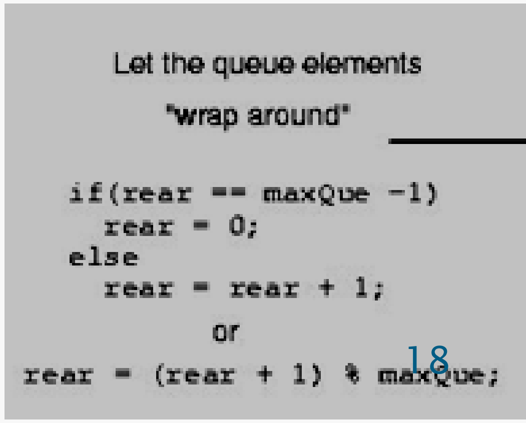


Circular Array

- Circular Array is an array where indexes starts back from 0 after reaching maximum value. If we have 8 elements array. Then range of array indexes are from 0 to 7.
- In circular array when an index is reached till 7 then the next index would become 0 starting from start.. Making it a circular array.

- In circular array we will increment indexes like:
  - **rear = (rear + 1) % size**
  - **front = (front + 1) % size**




Let the queue elements
"wrap around"

```
if(rear == maxQue -1)
  rear = 0;
else
  rear = rear + 1;
```

or

```
rear = (rear + 1) % maxQue;
```

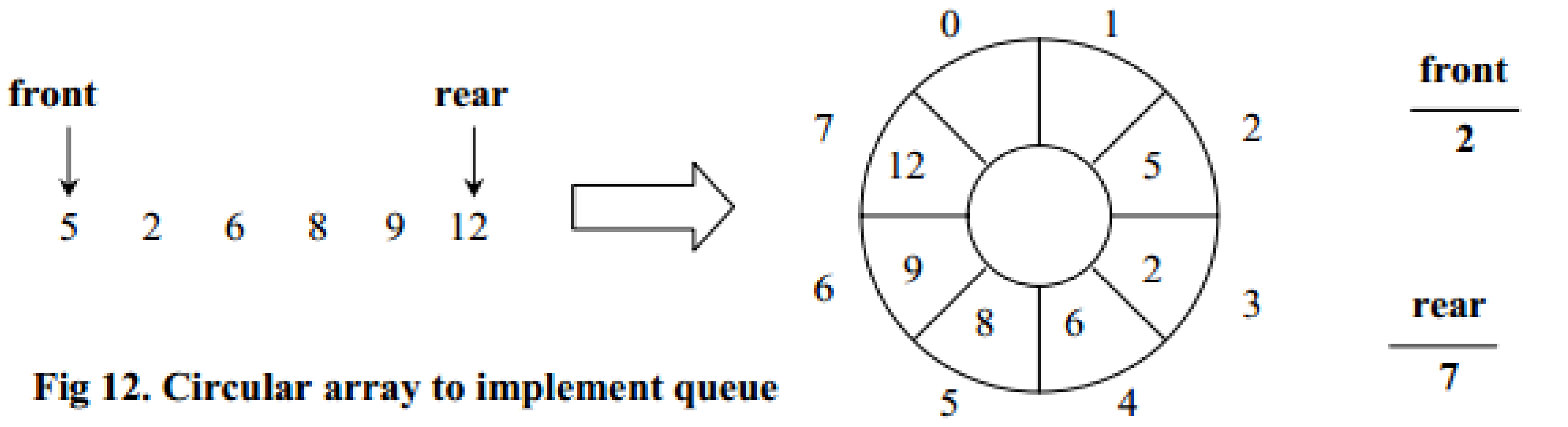


Fig 12. Circular array to implement queue

The number of locations in the above circular array are also eight, starting from index *0* to index *7*. The index numbers are written outside the circle incremented in the clock-wise direction. To insert an element *21* in the array , we insert this element in the location, which is next to index *7*.

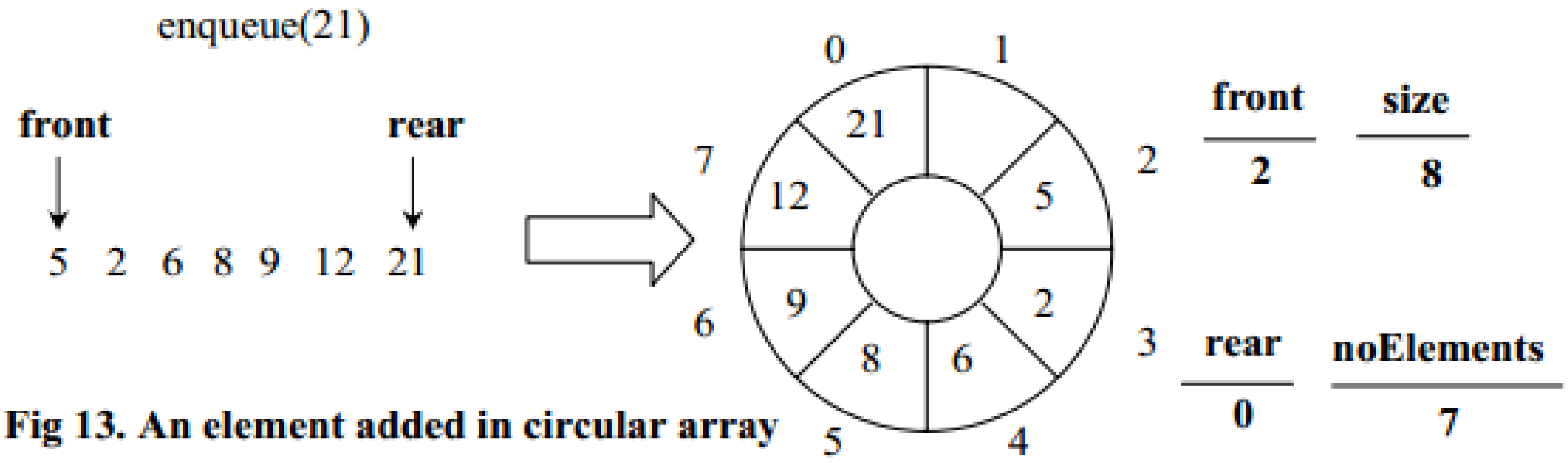


Fig 13. An element added in circular array

Enqueue using circular array

Using the variable "total-number-of-elements"

Let's add another element in the queue.

enqueue(7)

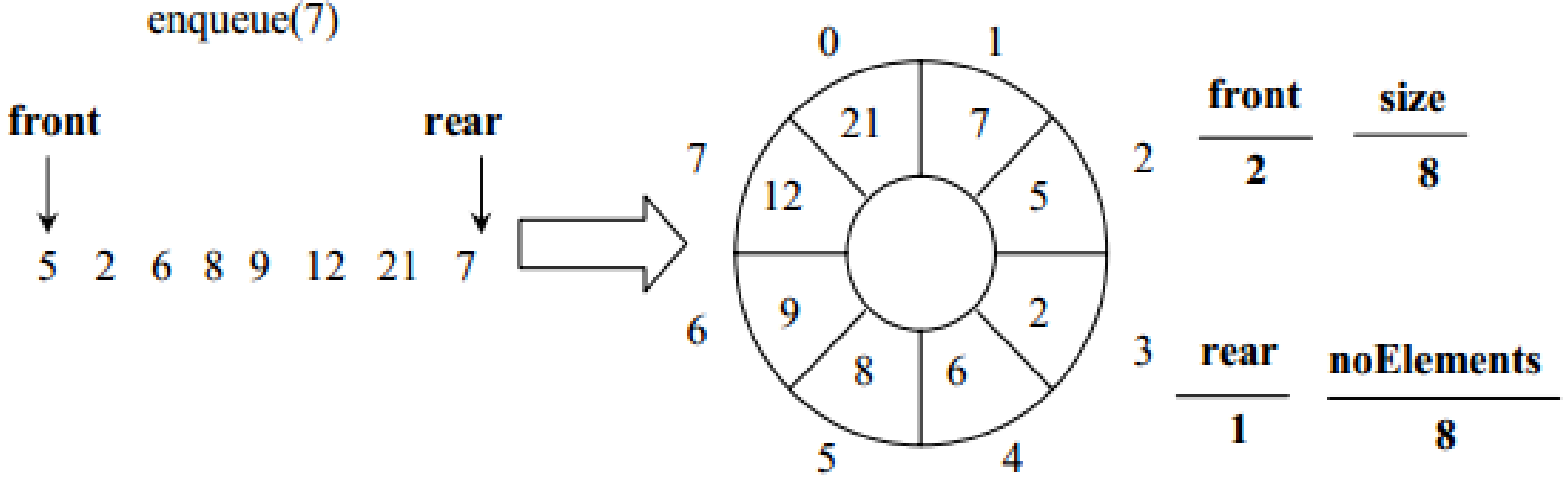
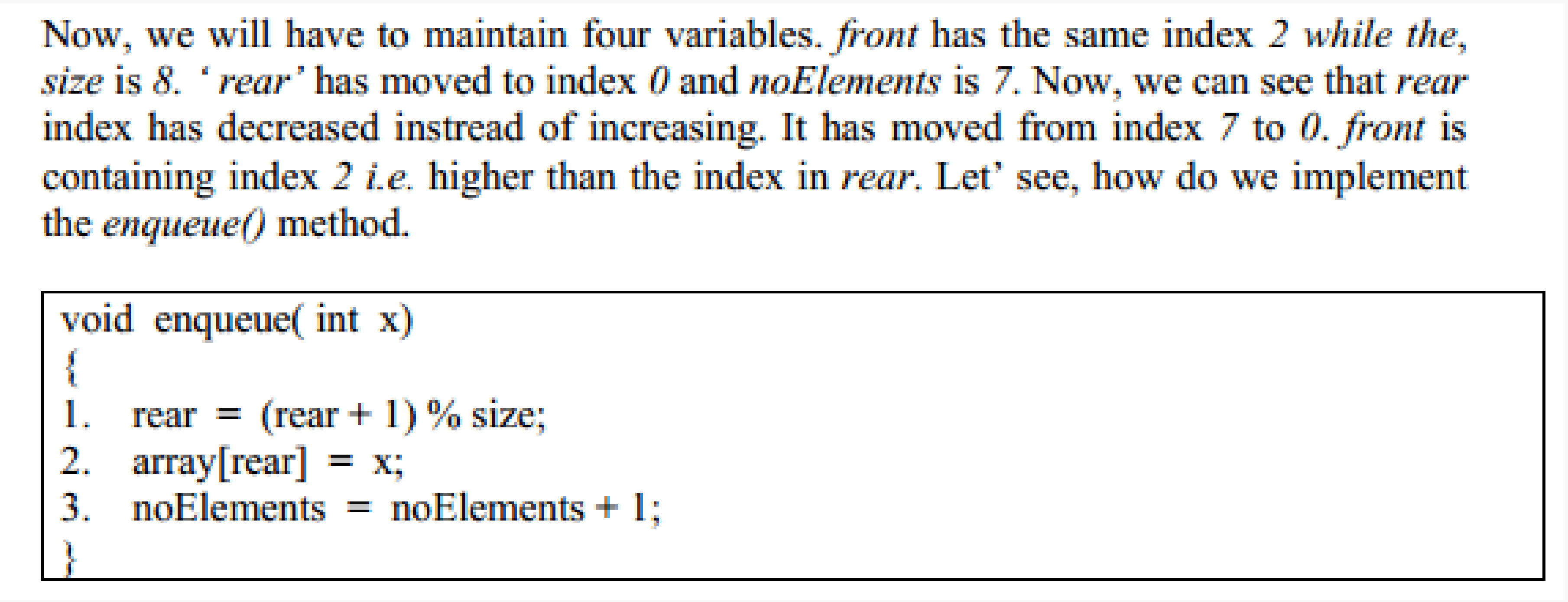


Fig 14. Another element added in circular array

Now, the queue becomes full

Insert / Enqueue Algorithm



Now, we will have to maintain four variables. *front* has the same index *2 while the*, *size* is *8*. '*rear*' has moved to index *0* and *noElements* is *7*. Now, we can see that *rear* index has decreased instread of increasing. It has moved from index *7* to *0*. *front* is containing index *2 i.e.* higher than the index in *rear*. Let' see, how do we implement the *enqueue()* method.

```
void  enqueue( int  x)
{
1.    rear  =  (rear + 1) % size;
2.    array[rear]  =  x;
3.    noElements  =  noElements + 1;
}
```

Delete / dequeue Algorithm

Let's see the *dequeue()* method.

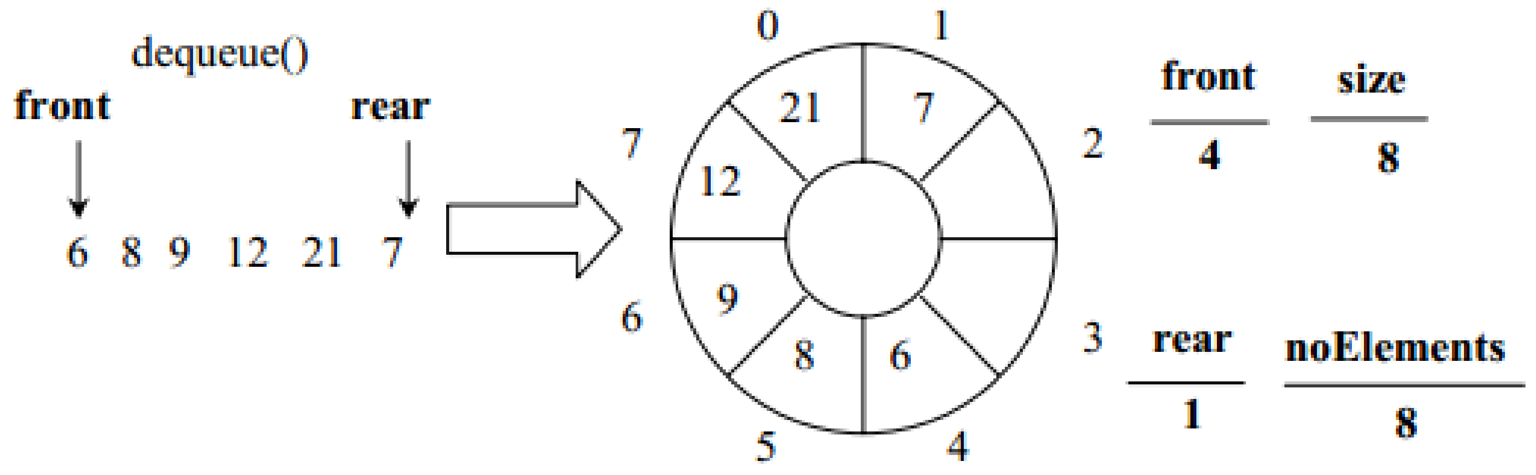
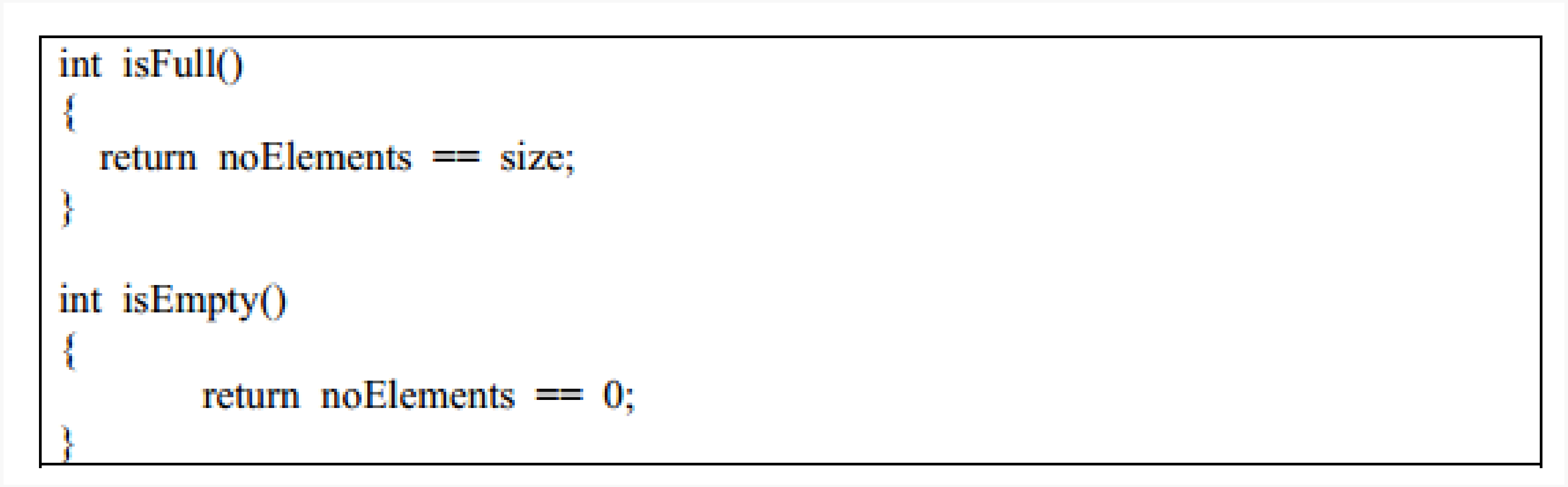


Fig 15. Element removed from the circular array

```
int  dequeue()
{
   int   x  =  array[front];
            front  =  (front + 1) % size;
            noElements  =  noElements - 1;
            return  x;
}
```



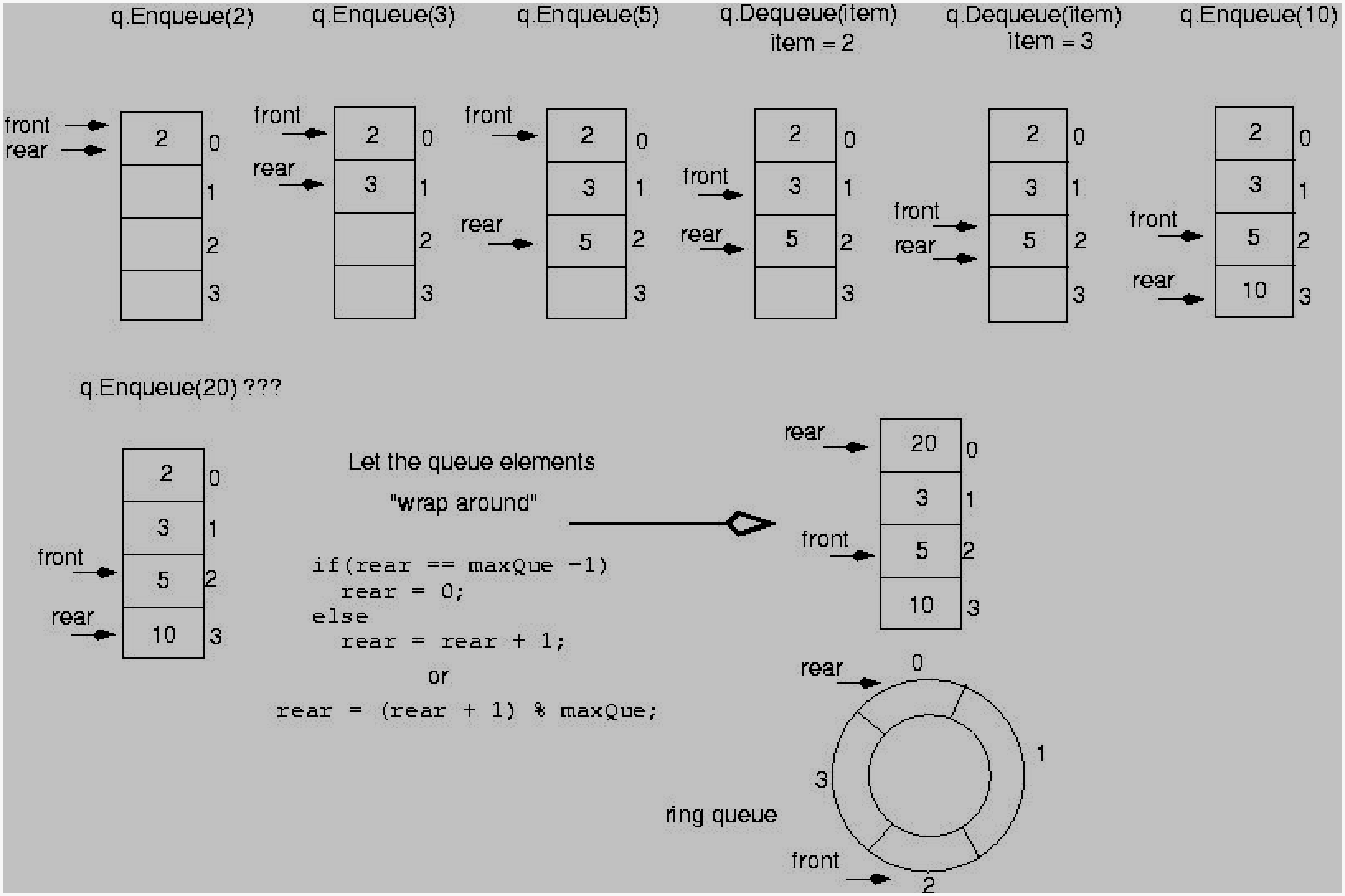
```
int  isFull()
{
   return  noElements  ==  size;
}

int  isEmpty()
{
         return  noElements  ==  0;
}
```

Over Flow and Under Flow Conditions using “Number of Elements” vairable

- isFull()returns true if the number of elements (noElements) in the array is equal to the size of the array. Otherwise, it returns false.

- Similarly isEmpty()looks at the number of elements (noElements) in the queue. If there is no element, it returns true or vice versa..

Queue *enqueue* and *dequeue* operations as a *Circular Structure*.



```
if(rear == maxQue -1)
   rear = 0;
else
   rear = rear + 1;
```

or

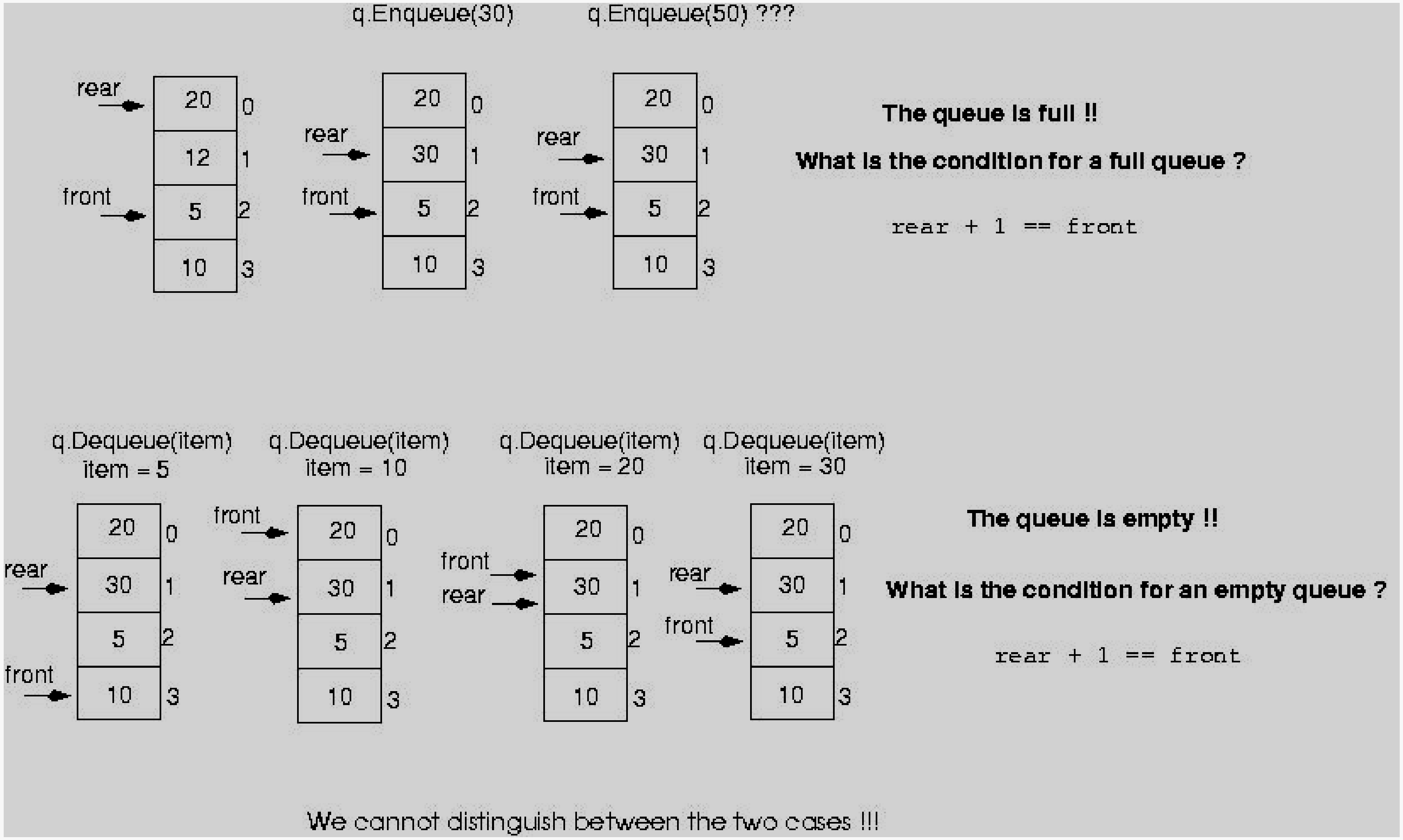
```
rear = (rear + 1) % maxQue;
```

Is there way to find Under Flow and Over Flow conditions using Front and Rear indexes /variables?

Now we will identify another method of implementing Overflow and Underflow of Queue using Front and Rear variables without using the Number of Elements variable in the queue.

We will find what values of Front and Rear will identify Full and Empty conditions of a Queue.

How do we know if a queue is full or empty using Rear and Front pointers?



The queue is full !!

What is the condition for a full queue ?

`rear + 1 == front`

The queue is empty !!

What is the condition for an empty queue ?

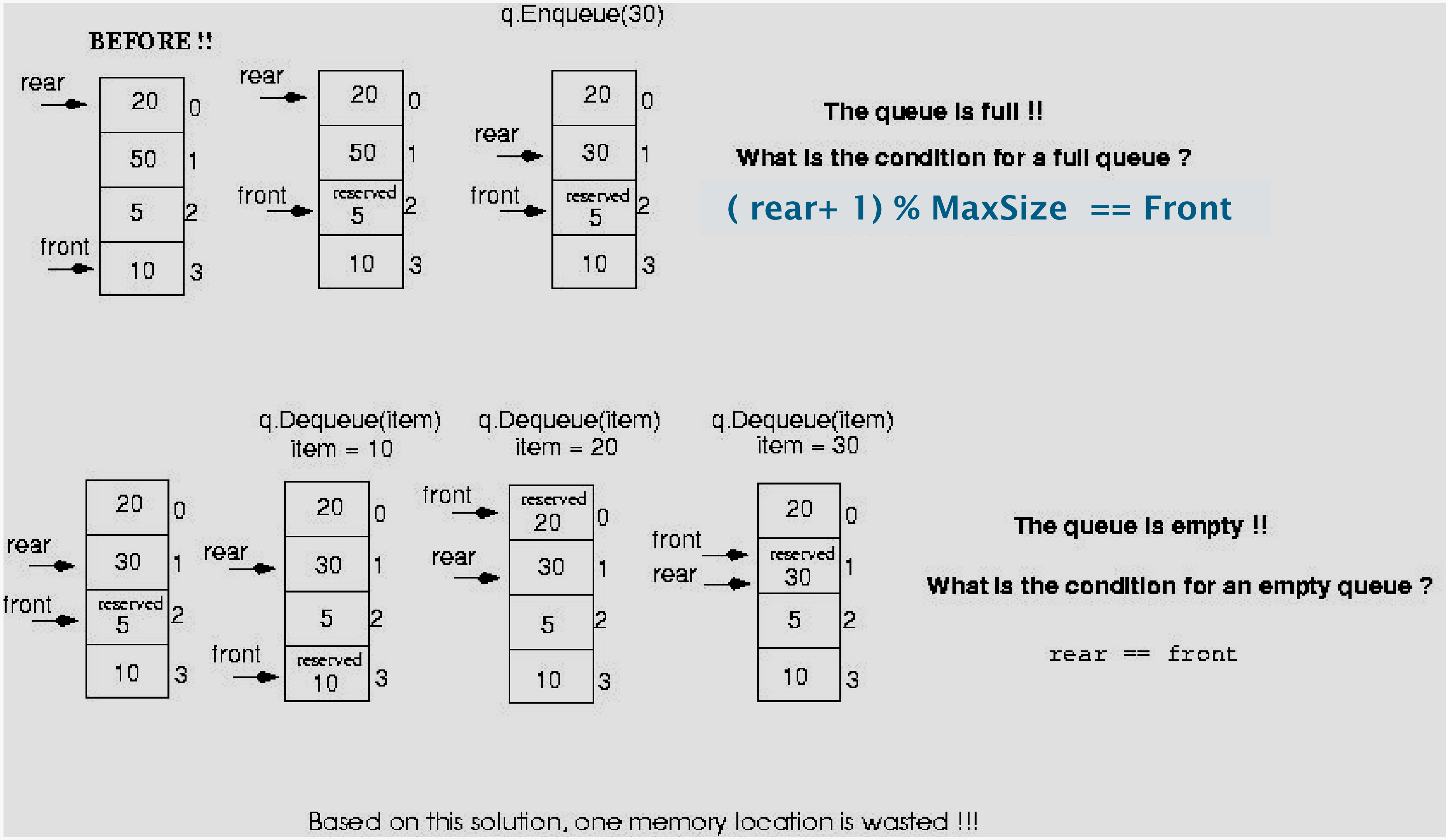
`rear + 1 == front`

We cannot distinguish between the two cases !!!

So in CIRCULAR IMPLEMENTATION of Queue both Empty and Full cases Rear and Front has the same relation of equation

So we need to find another way to implement the Circular Queue so that Empty and Full cases have different relationships of Rear and Front to identify Overflow and Underflow scenarios.

MODIFIED IMPLEMENATION:
Make *front* point to the element **preceding** the front element in the queue (one memory location will be wasted).



The queue is full !!

What is the condition for a full queue ?

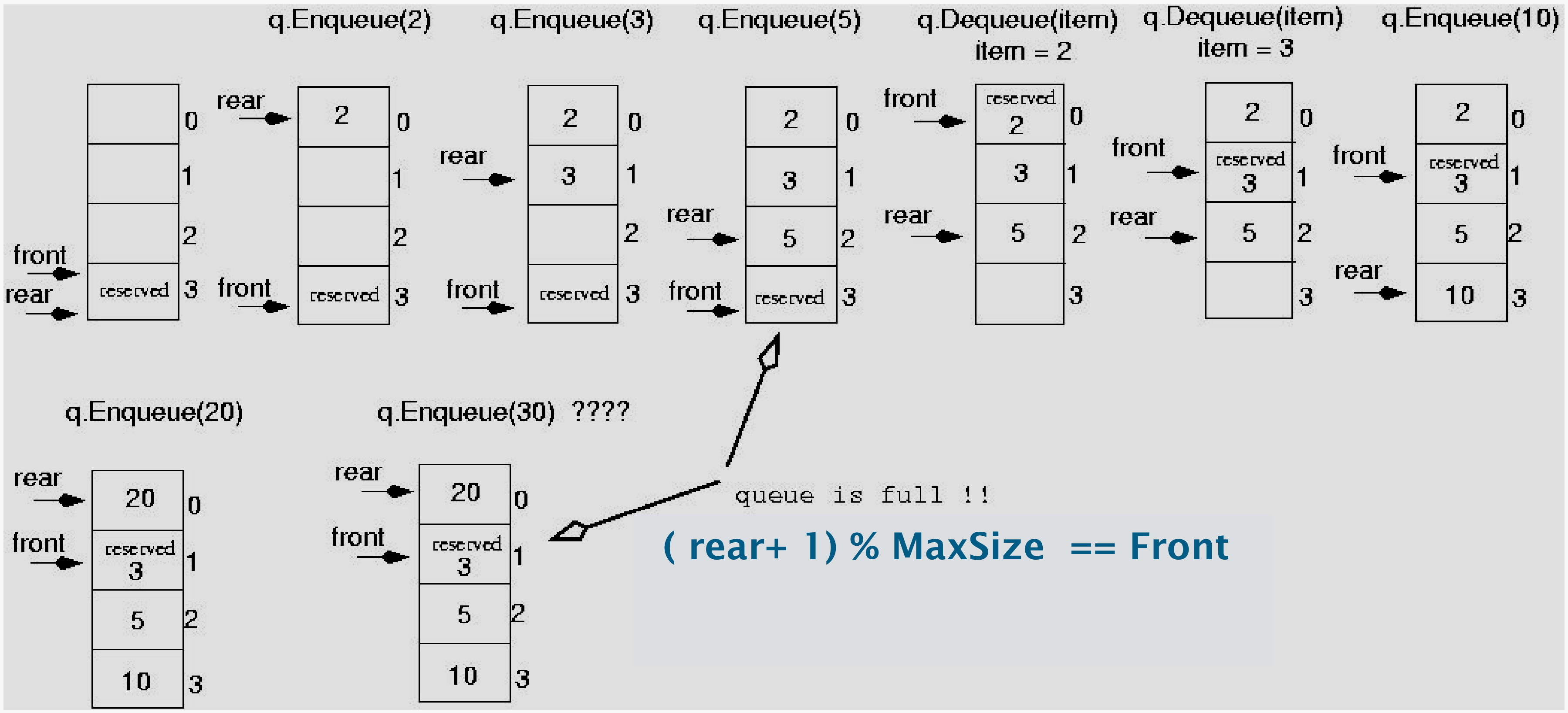
(rear+ 1) % MaxSize == Front

The queue is empty !!

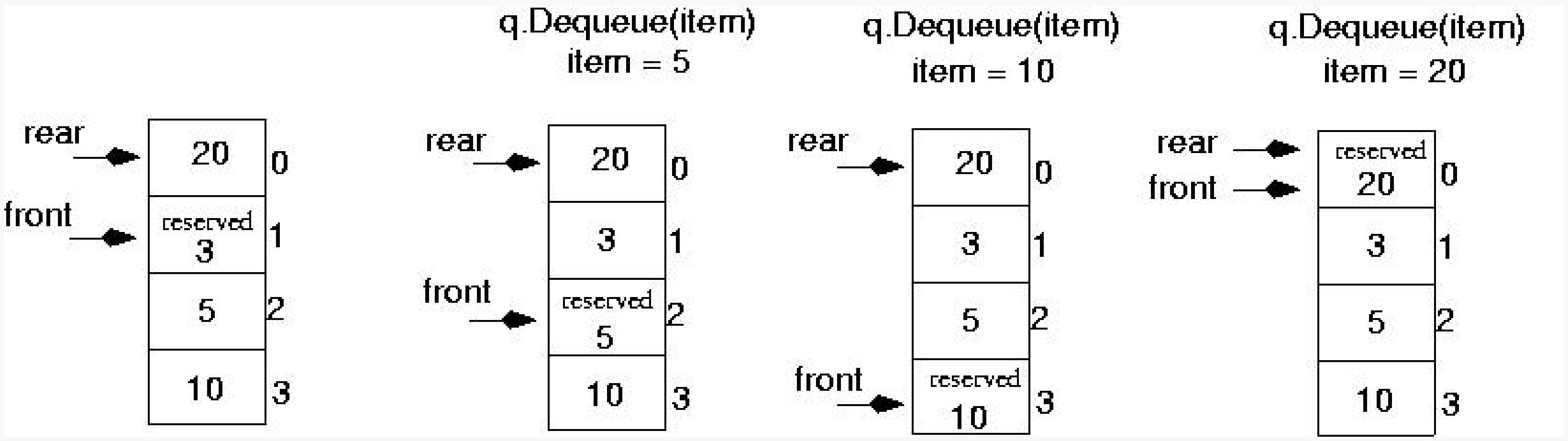
What is the condition for an empty queue ?

```
rear == front
```

Based on this solution, one memory location is wasted !!!



Front = 0 and Rear = 0



Queue is empty for rear == front

Initialization of Front and Rear:

So we can initialize *front* and *rear to 0* as this would also represent empty queue in the start.

Algorithms Circular Queue

- While implementing Circular Queue using Array, initialise Front = 0 and Rear = 0

- Algorithm for Enqueue

```
Enqueue(value)
{
 if (Front ==  ( rear+ 1) % size  )
        Over flow, Insert not possible
        Exit
  rear = (rear+ 1) % size
  array[rear] = value
 }
```

- Algorithm for Delete / Dequeue

```
dequeue(value)
{
 if (Front == Rear)
        Queue is Empty, Delete not possible
        Exit
  front = (front + 1) % size
  x = array[front]
}
```