CSE 105:
Data Structures and Algorithms-I
(Part 2)

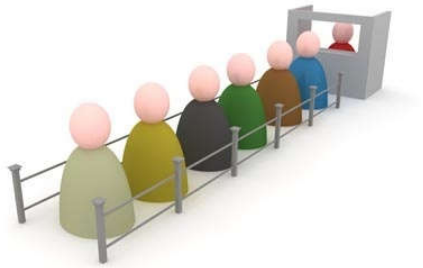
Instructor

Dr Md Monirul Islam

Queues

FIFO: First in, First Out

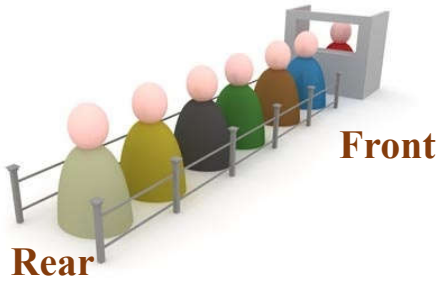
Restricted form of list: Insert at one end, remove from the other.



Queues

FIFO: First in, First Out

Restricted form of list: Insert at one end, remove from the other.



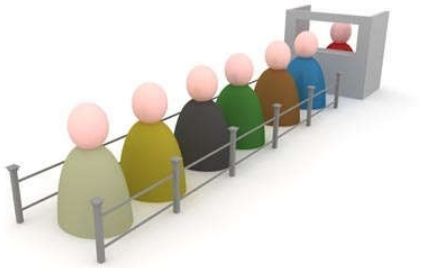
Queues

FIFO: First in, First Out

Restricted form of list: Insert at one end, remove from the other.

Notation:

- Insert: Enqueue
- Delete: Dequeue
- First element: front, head
- Last element: rear, tail, back



Abstract Queue

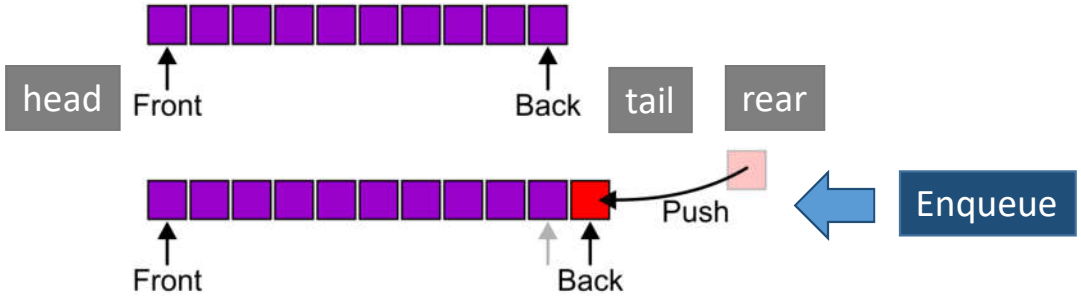
Operations:

- Insert: Enqueue
- Delete: Dequeue

Abstract Queue

Operations:

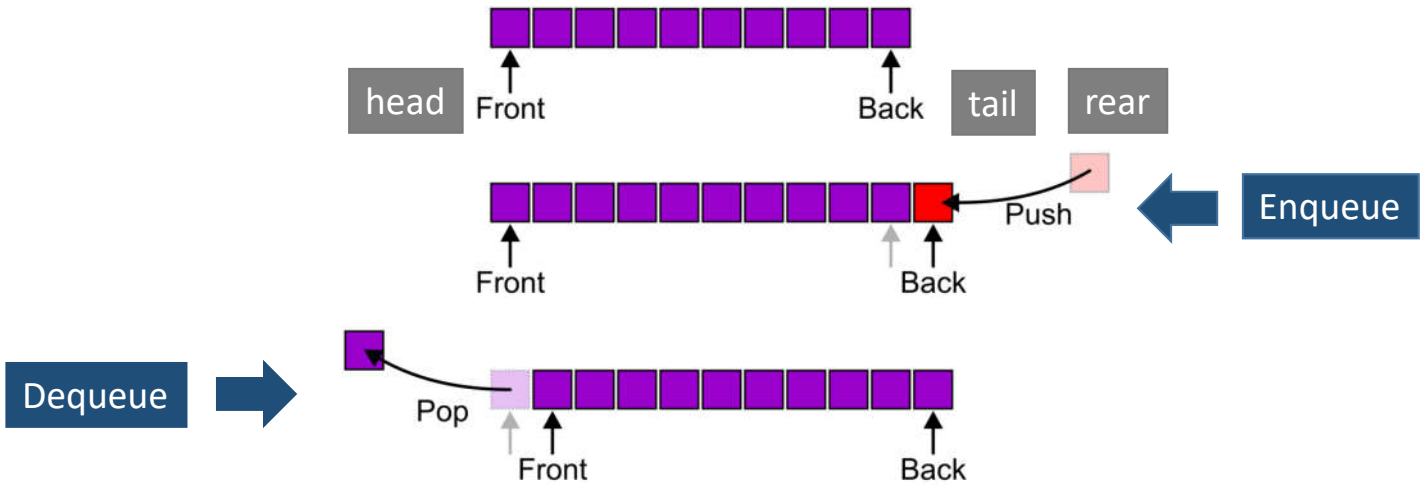
- Insert: Enqueue
- Delete: Dequeue



Abstract Queue

Operations:

- Insert: Enqueue
- Delete: Dequeue



Queue Applications

- The most common application is in client-server models
  - clients waits in a queue to be served by one or more busy servers
  - Later served in the order of arrival

- Grocery stores, banks, and airport security use queues

- Most shared computer services are servers:
  - Web, file, ftp, database, mail, printers, *etc*.
- . . . .

Implementations

We will look at two implementations of queues:

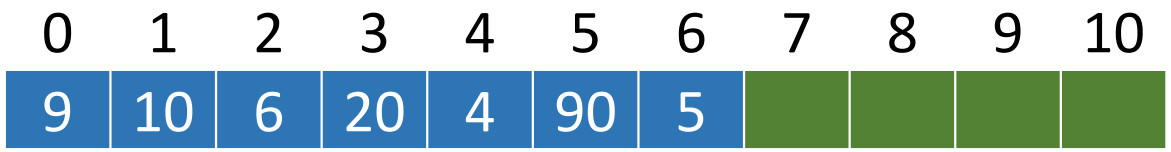
- Array based
- Linked list based

Requirements:

- All queue operations must run in $\Theta(1)$ time

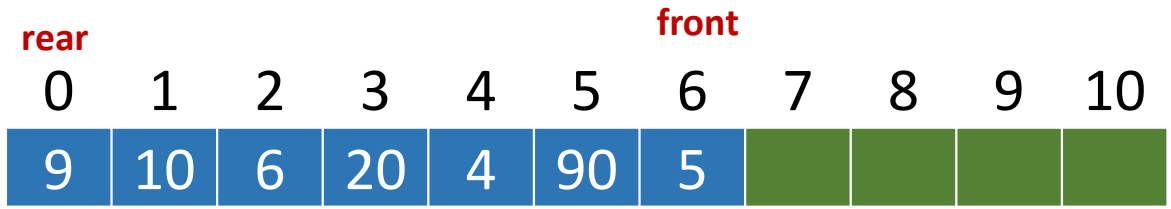
Array Based Implementation

- The array-based implementation needs some thought
- A weak implementation may result inefficiency



Array Based Implementation

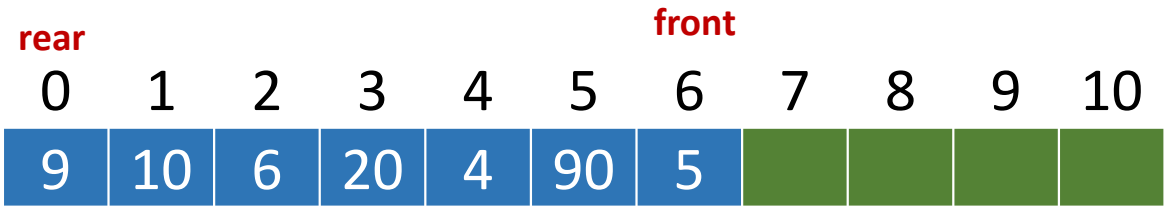
- Assume, *n* elements in the queue and they are stored in the first *n* positions of the array.



Array Based Implementation

Implementation1:

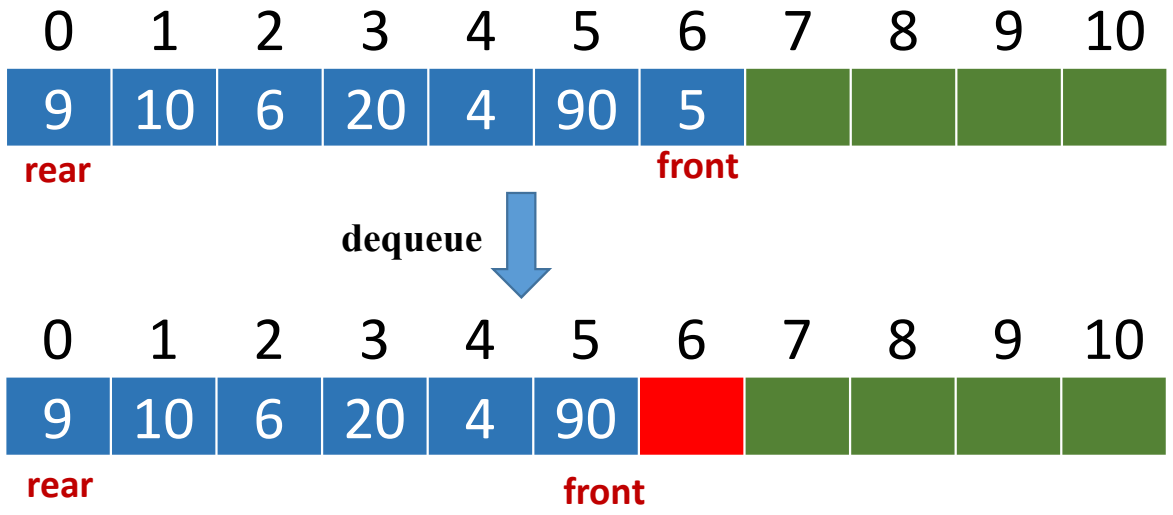
The rear element of the queue is at position 0



Array Based Implementation

Implementation1:

The rear element of the queue is at position 0, **dequeue operations require only O(1) time because the front element** of the queue is the last element in the array.



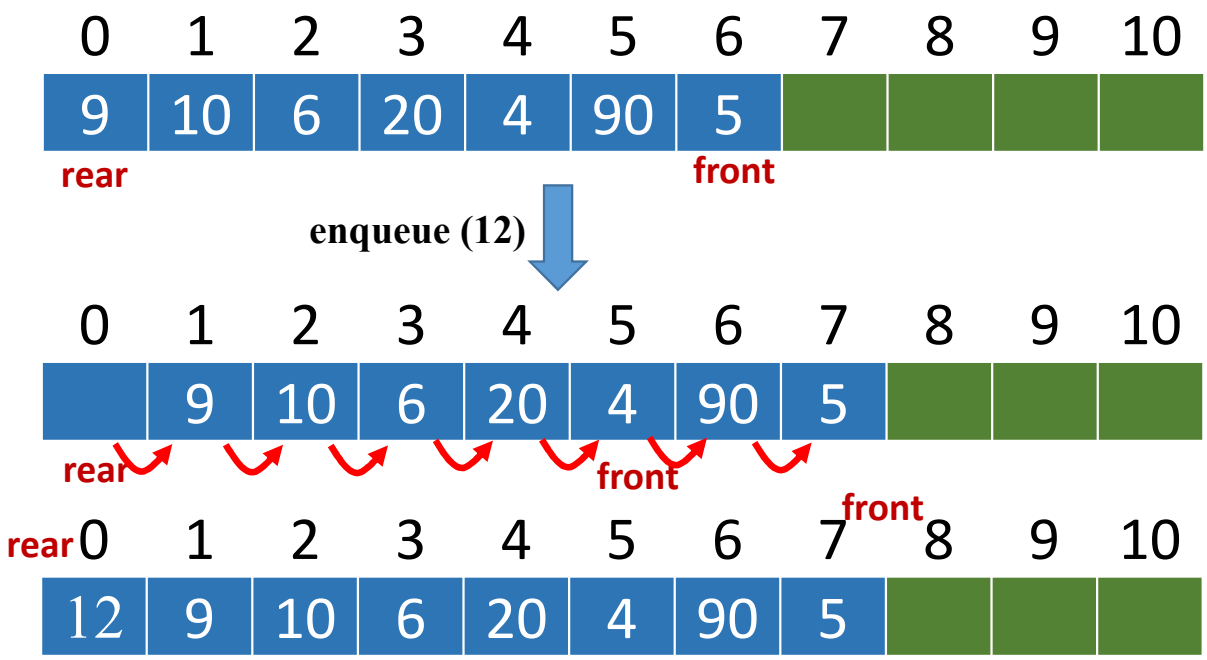
Array Based Implementation

Implementation1:

The rear element of the queue is at position 0,

enqueue operations require only O(*n*) time because the rear element of the queue is the first element in the array.

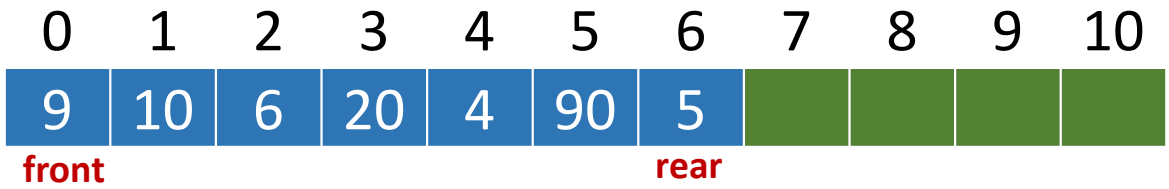
We need *n* shifts.



Array Based Implementation

Implementation2:

The rear element of the queue is at position n-1



Array Based Implementation

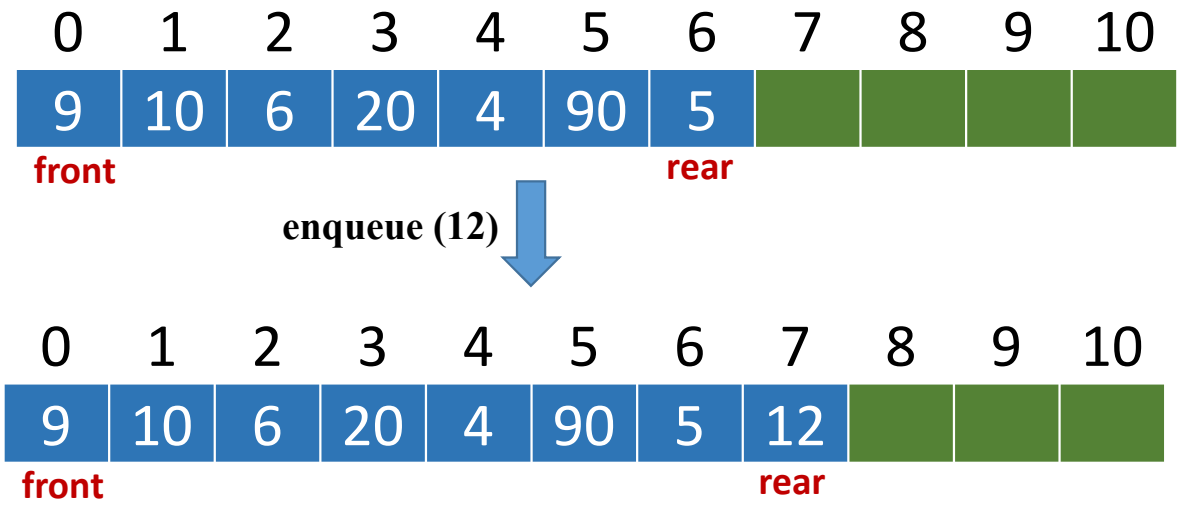
Implementation2:

The rear element of the queue is at position n-1

enqueue operations require only O(1) time because the rear element of the queue is the last element in the array.

We need NO shifts.

Similar to appending at the end.



Array Based Implementation

Implementation2:

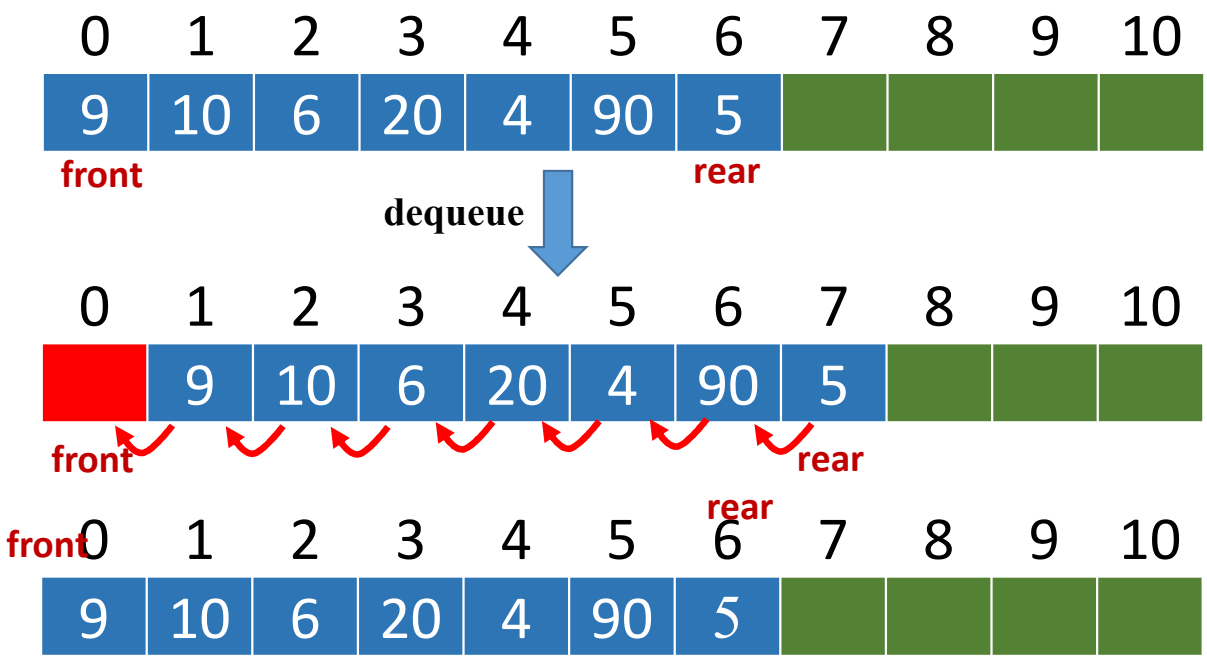
The rear element of the queue is at position 0,

dequeue operations require only O(*n*) time

because the front element of the queue is the

lastst element in the array.

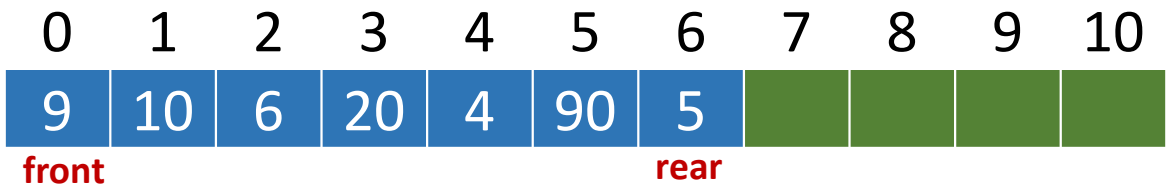
We need *n* shifts.



Array Based Implementation

Solution 1:
front at 0, rear at position n-1

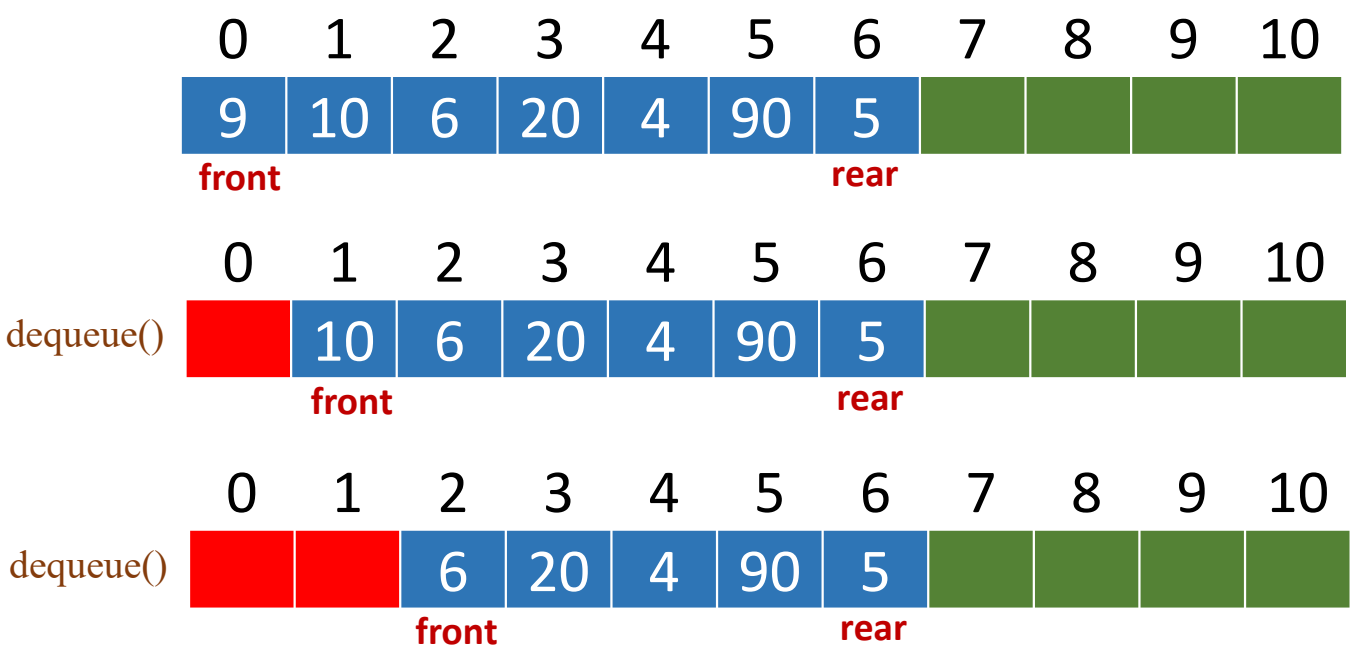
Move both front and rear as elements deleted and inserted



Array Based Implementation

Solution 1:
front at 0, rear at position *n*-1

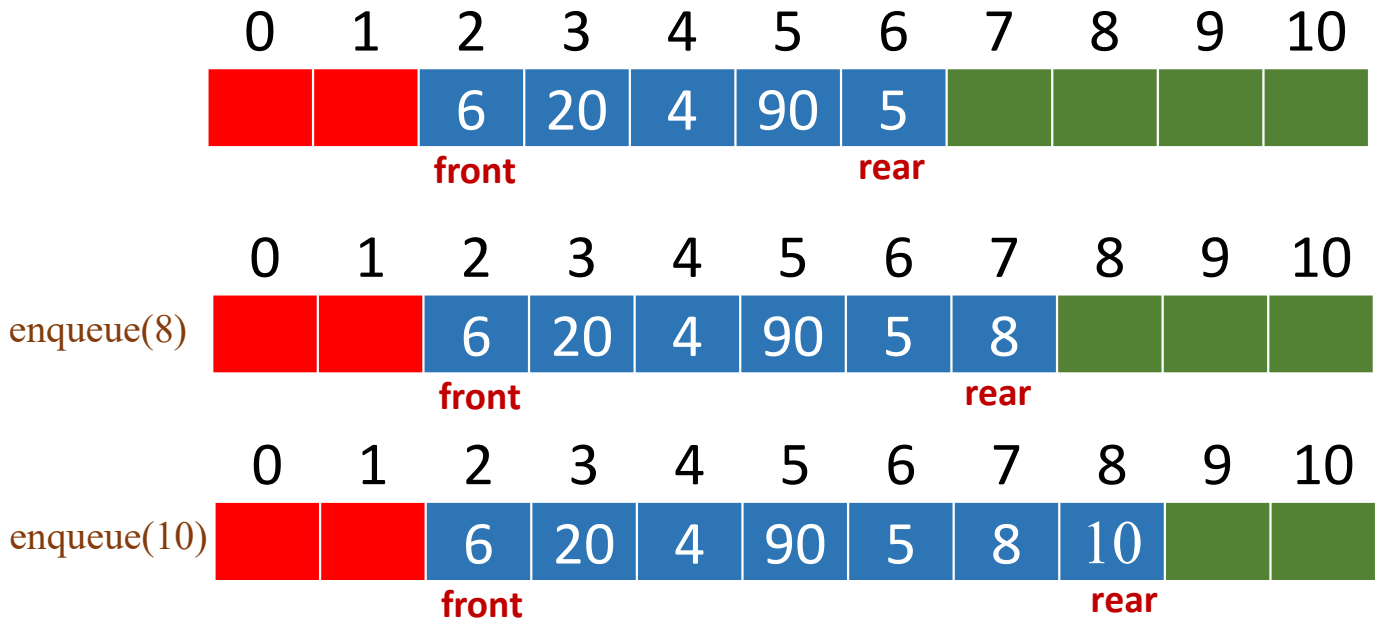
Move both front and rear as elements deleted and inserted



Array Based Implementation

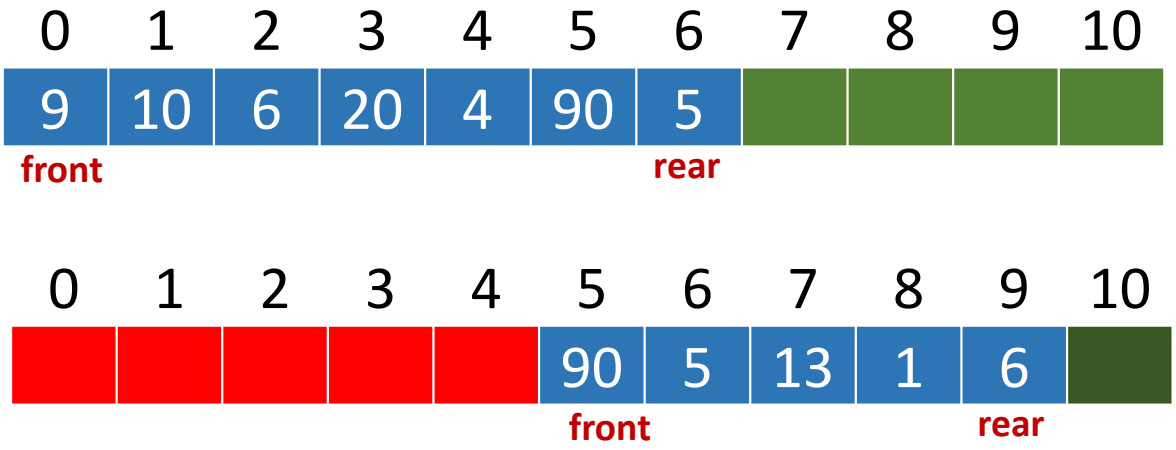
Solution 1:
front at 0, rear at position n-1

Move both front and rear as elements deleted and inserted



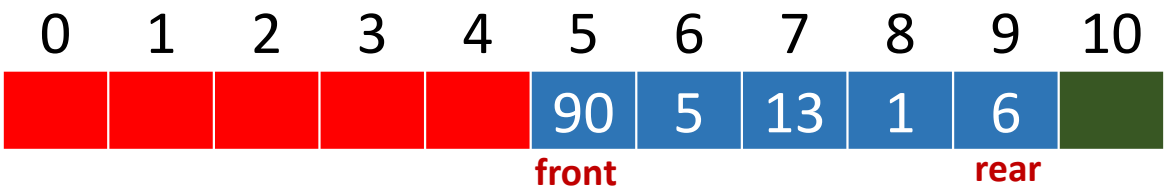
Performance

- Enqueue: O(1)
- Dequeue: O(1)
- Drifting Queue problem:
  - Dequeue => the front index increases.
  - Over time, space created in lower indexed positions
  - the queue will run out of space despite having empty slots at lower end



Performance

- After another single push queue will run out

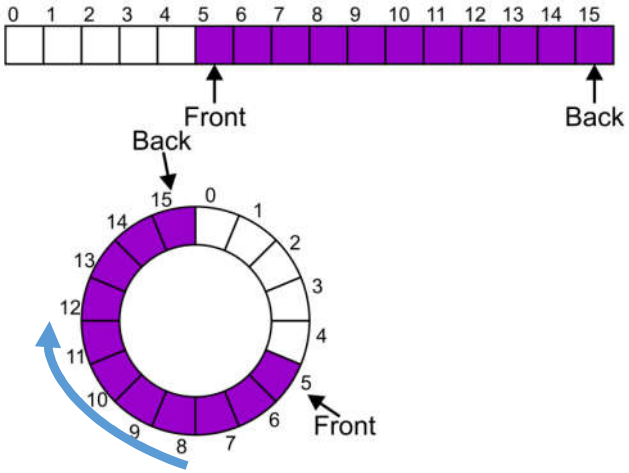


Solution to Drifting Queue Problem: Implementation (2)

Instead of viewing the array on the range 0, …, 15, consider the indices being cyclic:

…, 15, 0, 1, …, 15, 0, 1, …, 15, 0, 1, …

This is referred to as a *circular array*



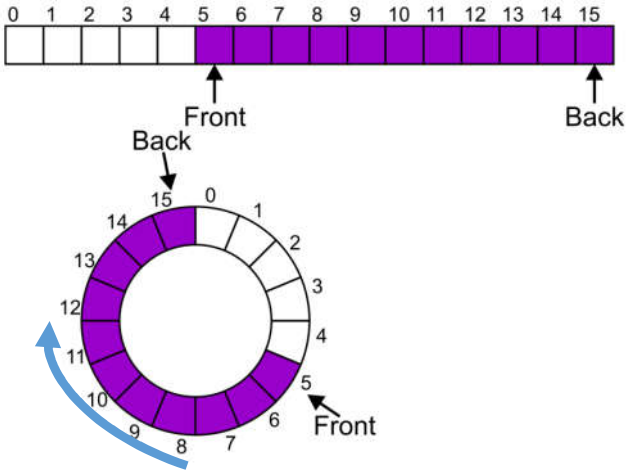
Solution to Drifting Queue Problem: Implementation (2)

Instead of viewing the array on the range 0, …, 15, consider the indices being cyclic:

…, 15, 0, 1, …, 15, 0, 1, …, 15, 0, 1, …

This is referred to as a *circular array*

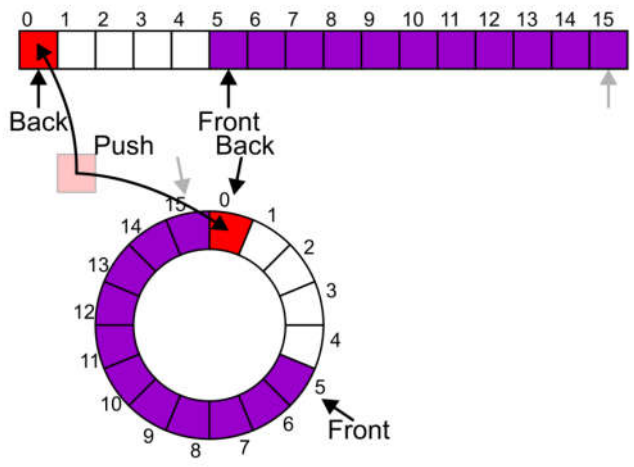
Queue has at indices from front to back



Solution to Drifting Queue Problem: Implementation (2)

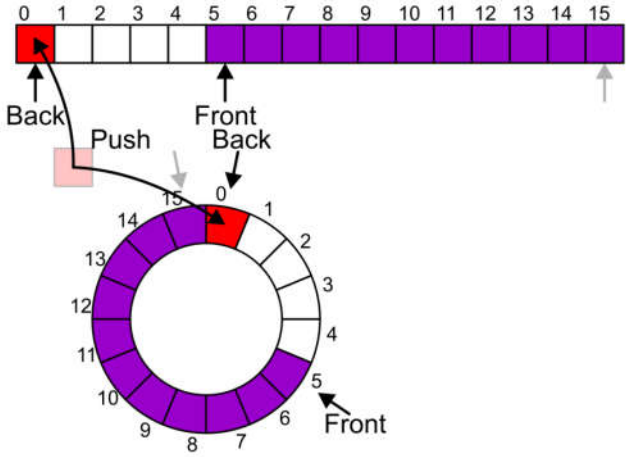
Now, the next push may be performed in the next available location of the circular array:

push => enqueu



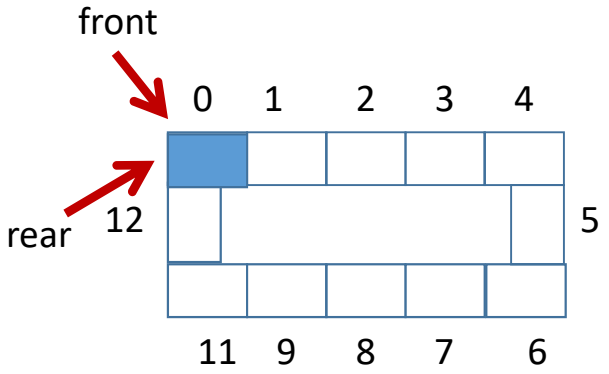
Solution to Drifting Queue Problem: Implementation (2)

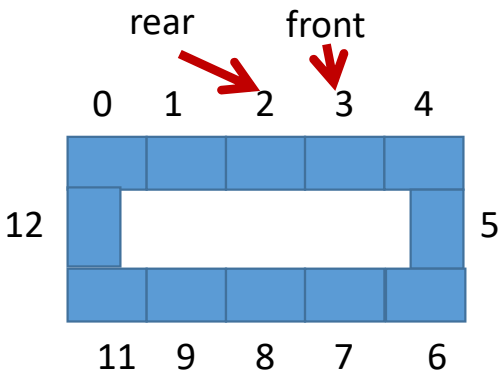
- We continue directly from the highest-numbered index to the lowest-numbered index.
- Use modulus operator (%)
  - **positions** are 0 through **size-1,**
  - **and position size-1 is defined** to immediately precede position 0

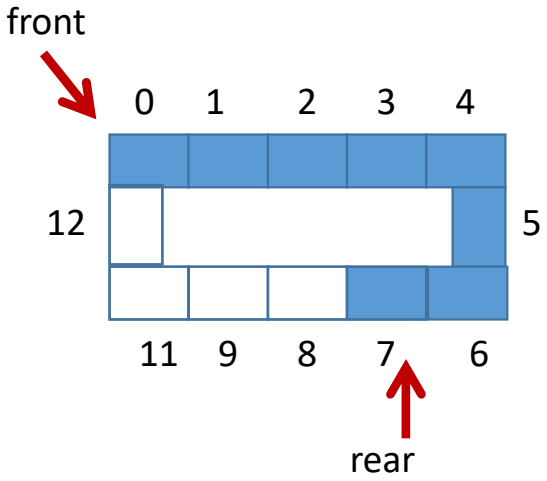


How to differentiate empty and full queue

- front = rear?
  - This means one element is there.
- rear = front -1?
  - queue is full?

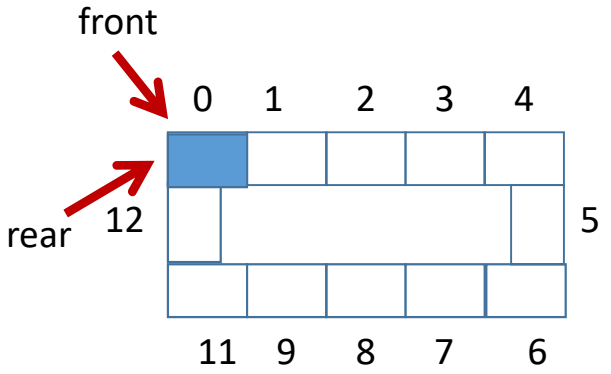


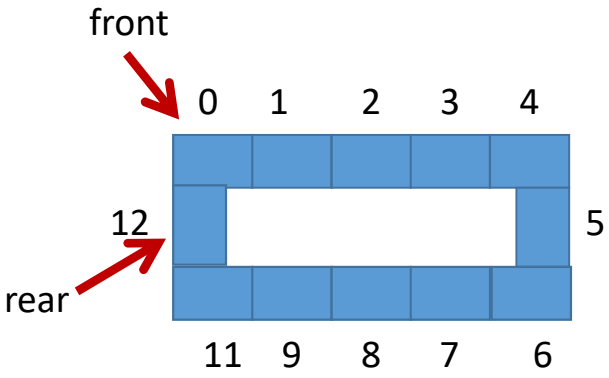


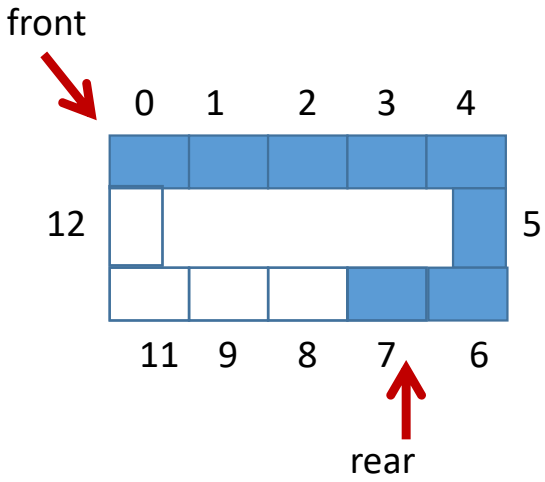


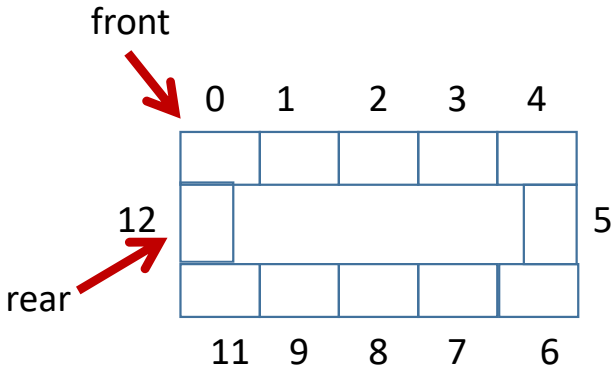
How to differentiate empty and full queue

- front = rear?
  - This means one element is there.
- rear = front -1?
  - queue is full?
  - queue is empty?



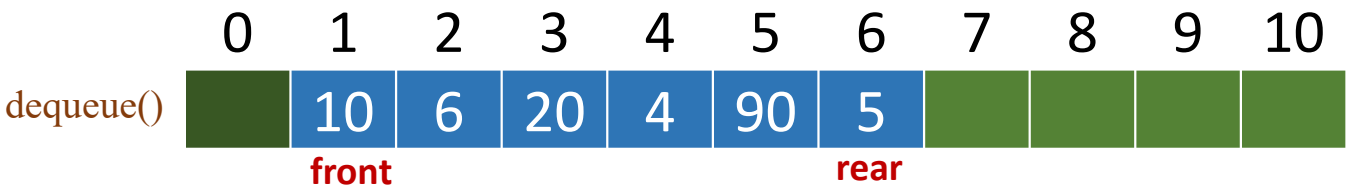


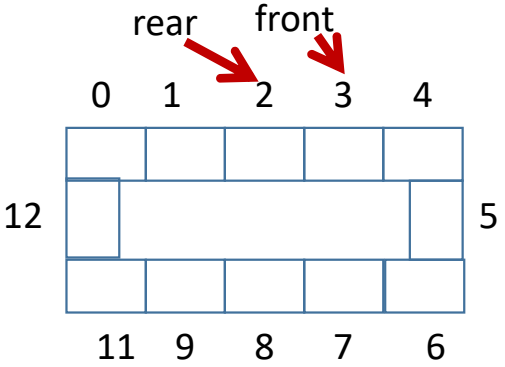


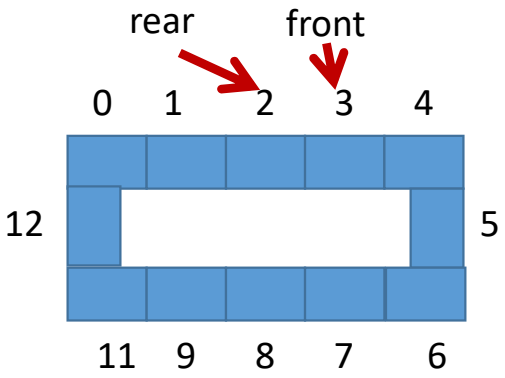


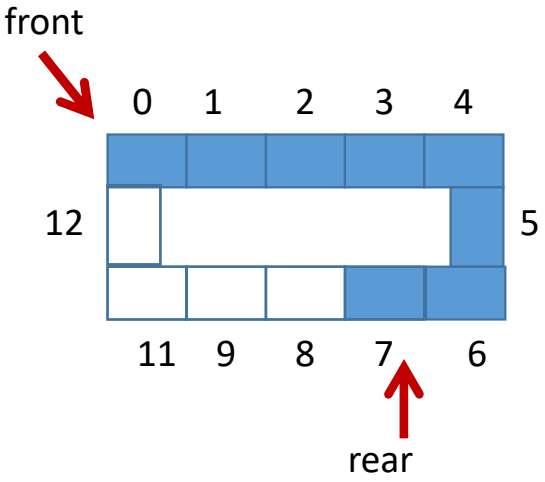
How to differentiate empty and full queue

- When is empty



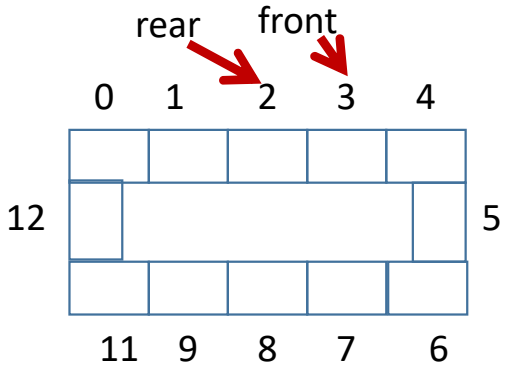


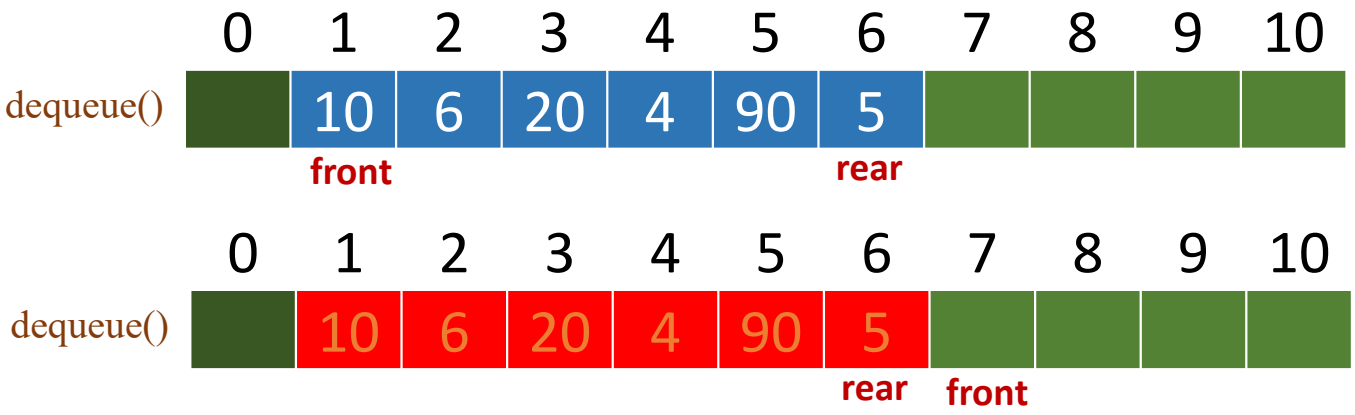




How to differentiate empty and full queue

- When is empty

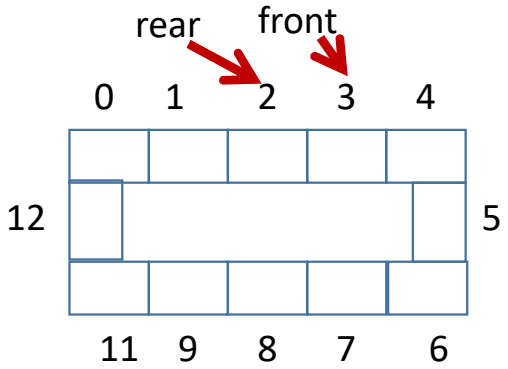


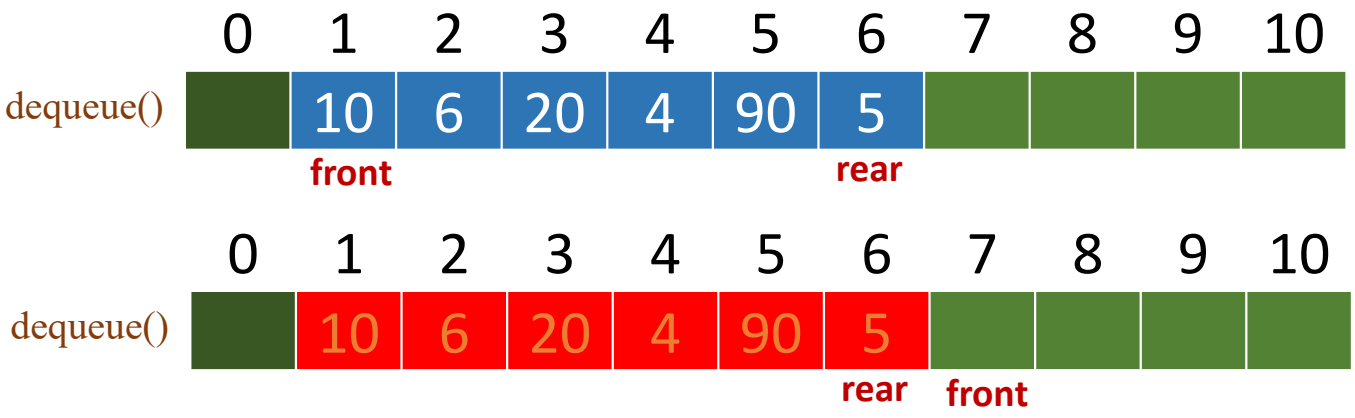


How to differentiate empty and full queue

- When is empty

use special case: rear = front=-1;





Array based Implementation of Queue

```
int maxSize;  // Max size of queue
int front, rear;
int  *array;
```

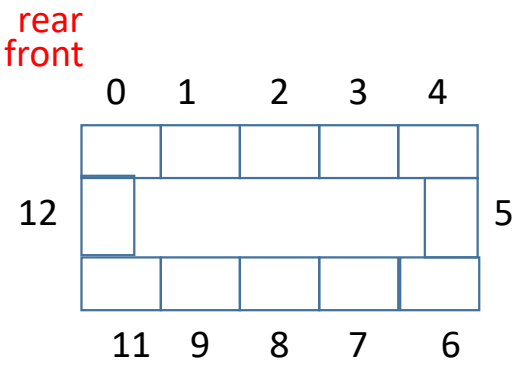
```
initialize ()
front = rear = -1;
array  = //make necessary allocation of
size maxSize;
```

isEmpty()

```
return rear== -1;
```

isFull()

```
return (rear+1 )% maxSize ==front;
```



Array based Implementation of Queue

```
enqueue(int data)
if (isFull() ) //error handling
else
     rear=(rear+1)%maxSize
     array[rear] =data;
     if front==-1   //first element
         front=0;
```

```
pop()
if (isempty() ) //error handling

else
     data=array[front];
     if (front==rear)     //last element
          front=rear=-1;
     else
           rear=(rear+1)%maxSize
     return data
```