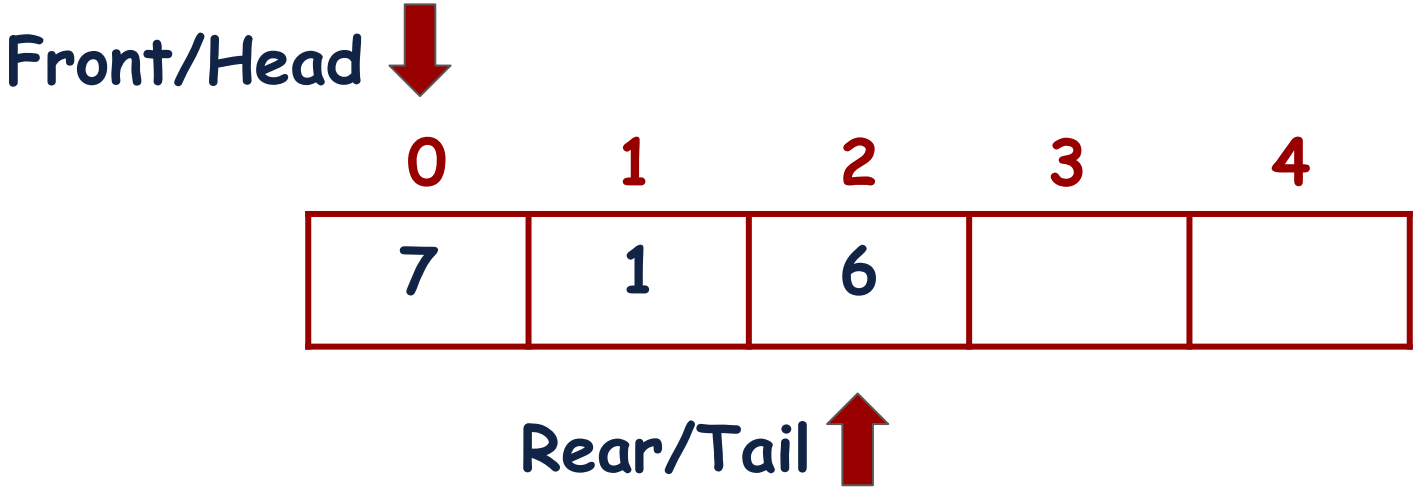
Circular Queue Data Structure

Queue: Review

Previously, we studied Queue Data Structure in which we add elements at the rear end and remove elements from the front end.

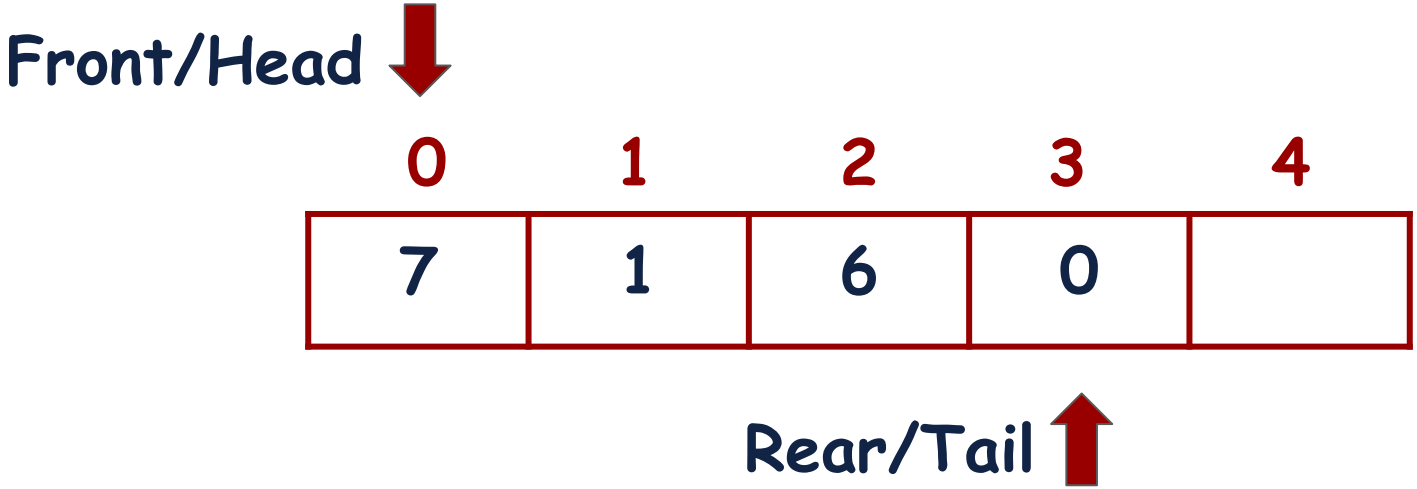
Queue: Review

Suppose we have Enqueued 3 elements in the Queue (implemented with Array).



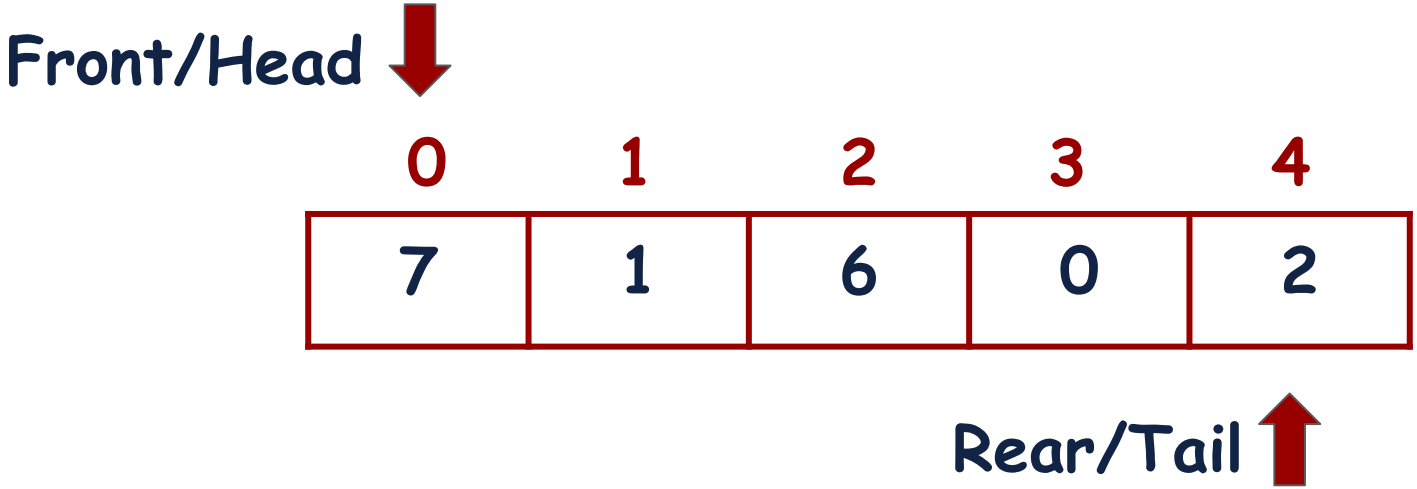
Queue: Review

Suppose we have also Enqueued 4th element in the Queue.



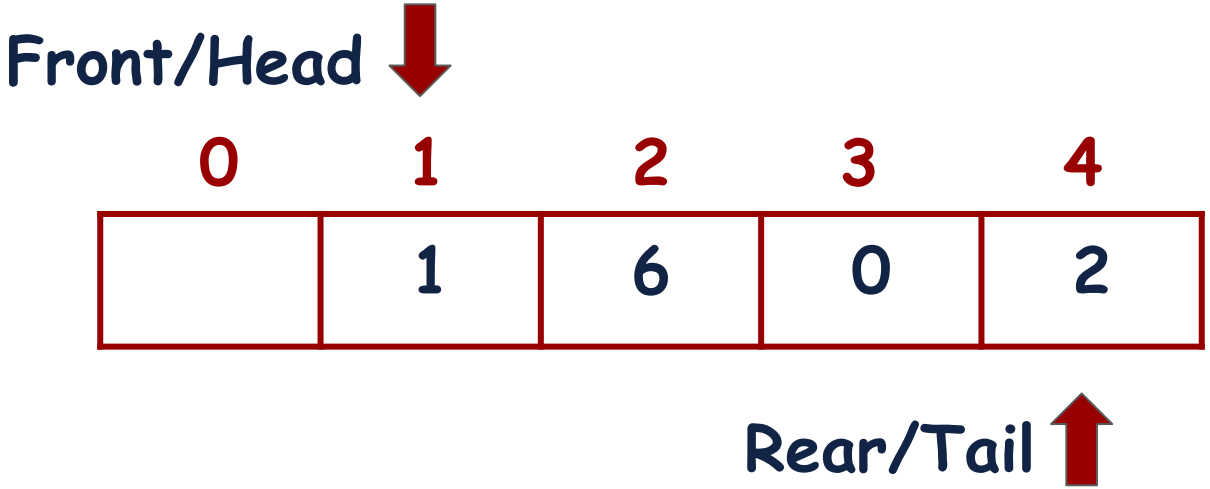
Queue: Review

Suppose we have also Enqueued 5th element in the Queue.



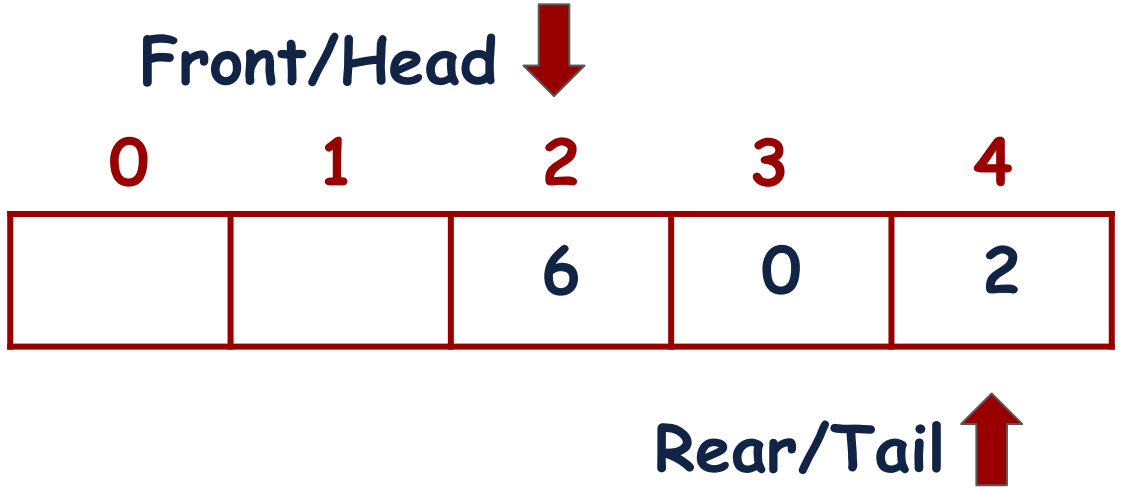
Queue: Review

Now, we Dequeued first element from the Queue.



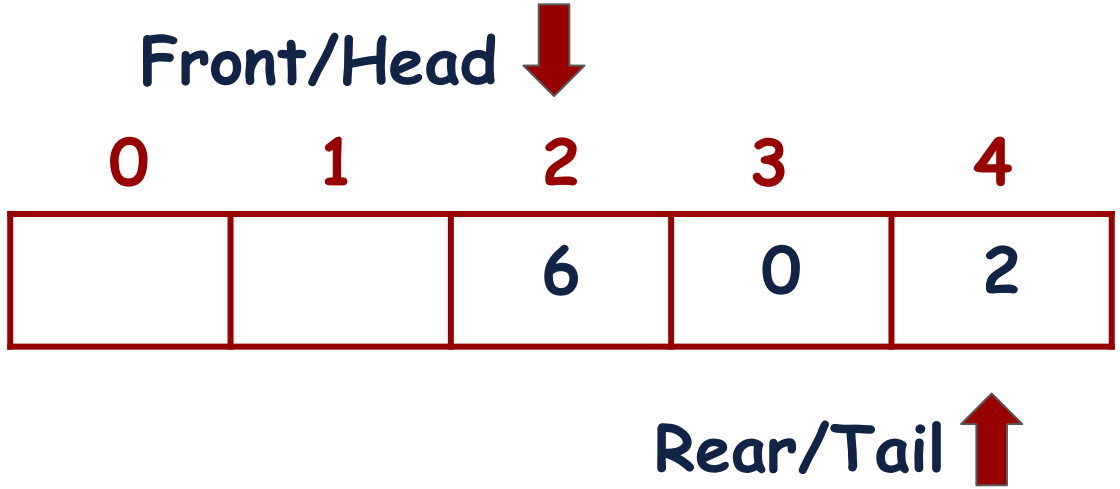
Queue: Review

Now, we Dequeued another element from the Queue.



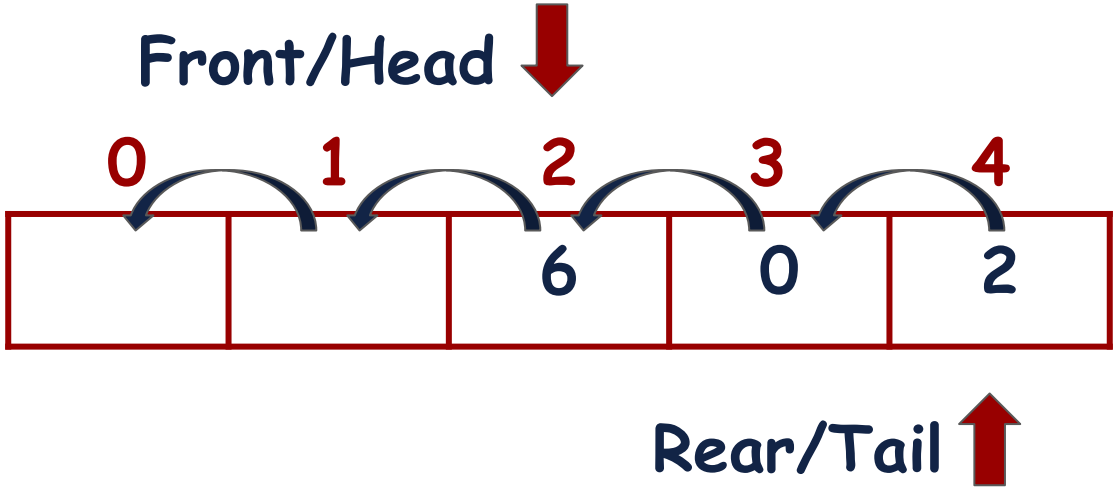
Queue: Review

The issue with this approach is that there is empty space left at the start of the array and we can not use it.



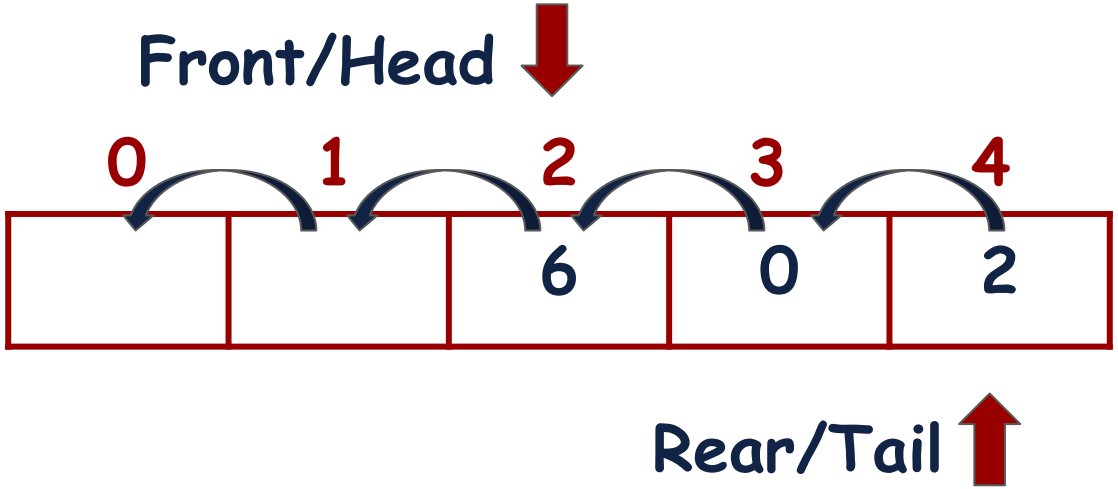
Queue: Review

In order to use the empty space, we have to shift all the elements towards the left after each Dequeue operation.



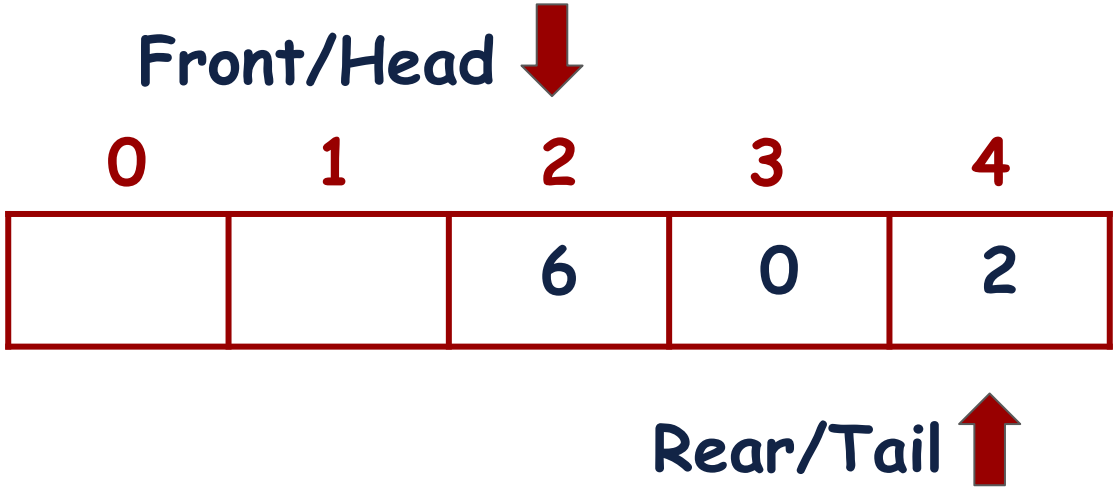
Queue: Review

But this shifting comes with a computational cost with Time Complexity O(n).



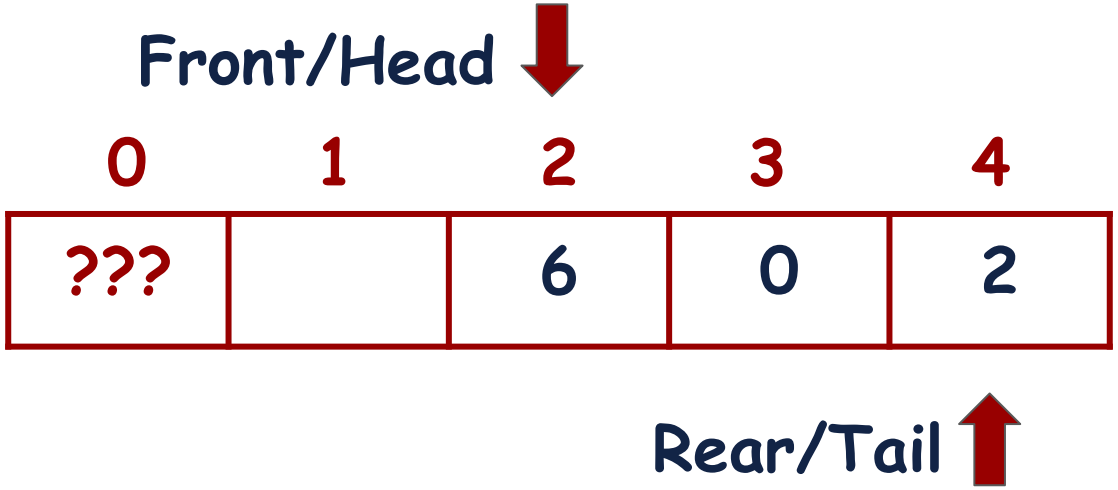
Queue: Better Solution using Array

Now, is there any better solution with **Array Implementation** in which we can use the empty space with Time Complexity **O(1)** for Dequeue operation?



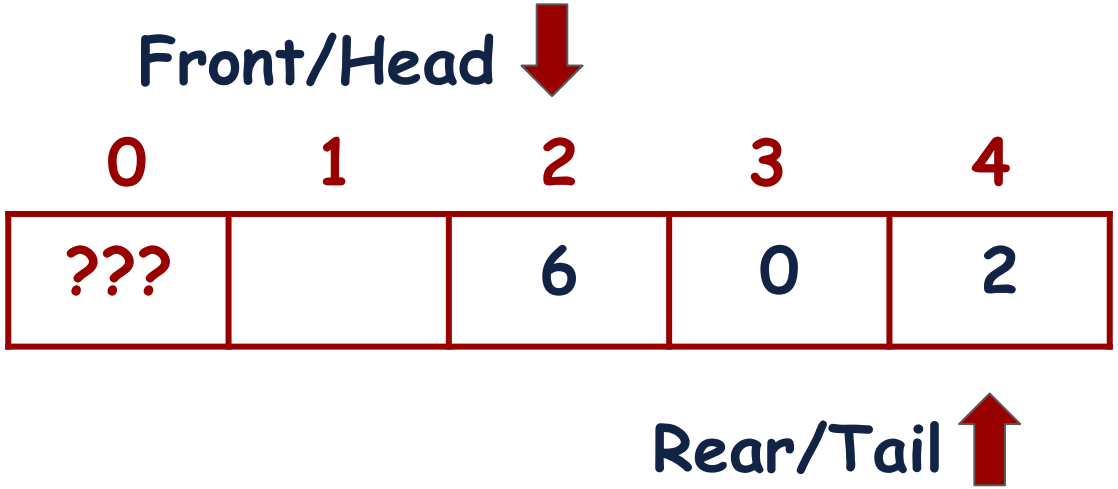
Queue: Better Solution using Array

There is empty space at the start of the array. Can we use that to **Enqueue** another element?

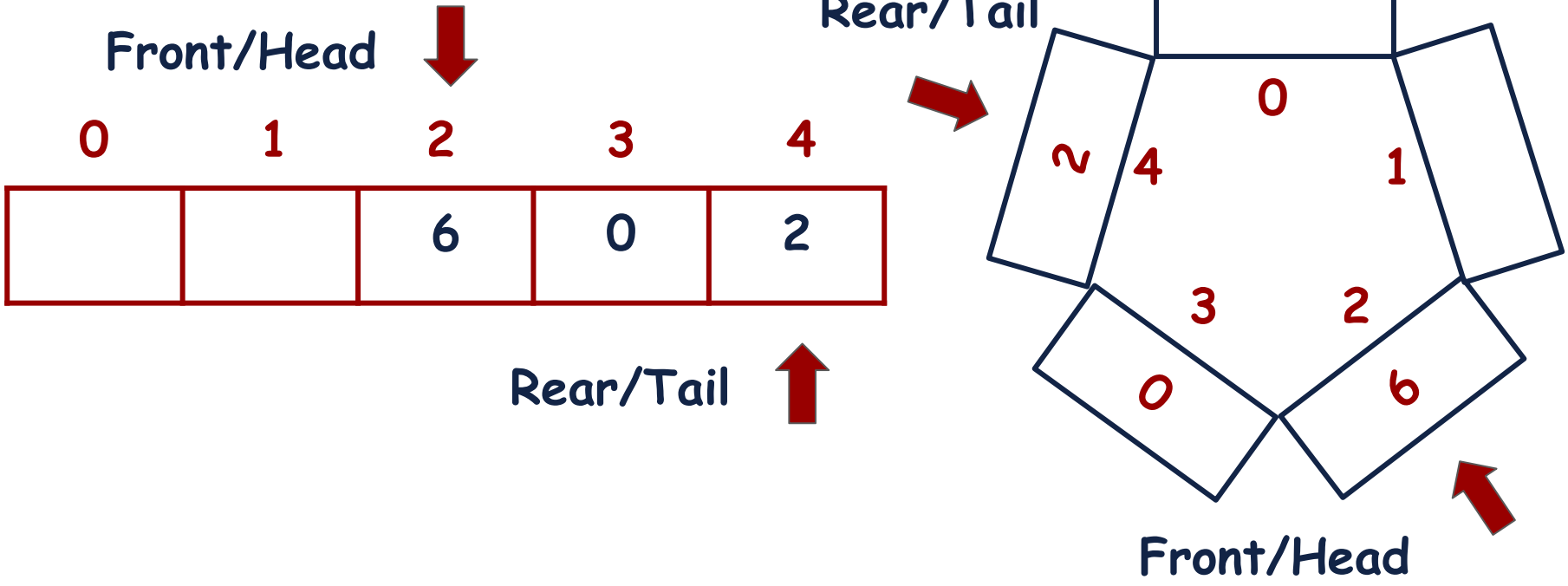


Circular Queue

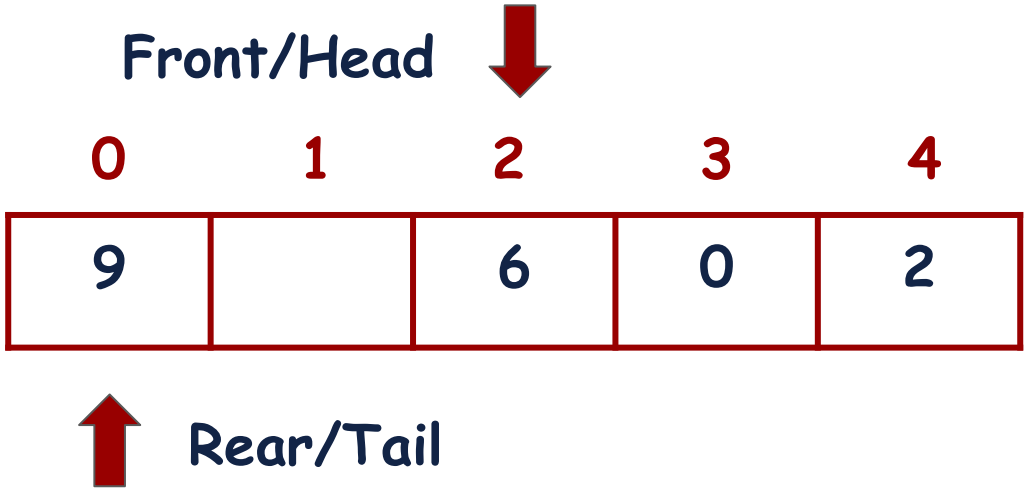
Yes we can..!!!
This is the extended version of Queue known as Circular Queue.

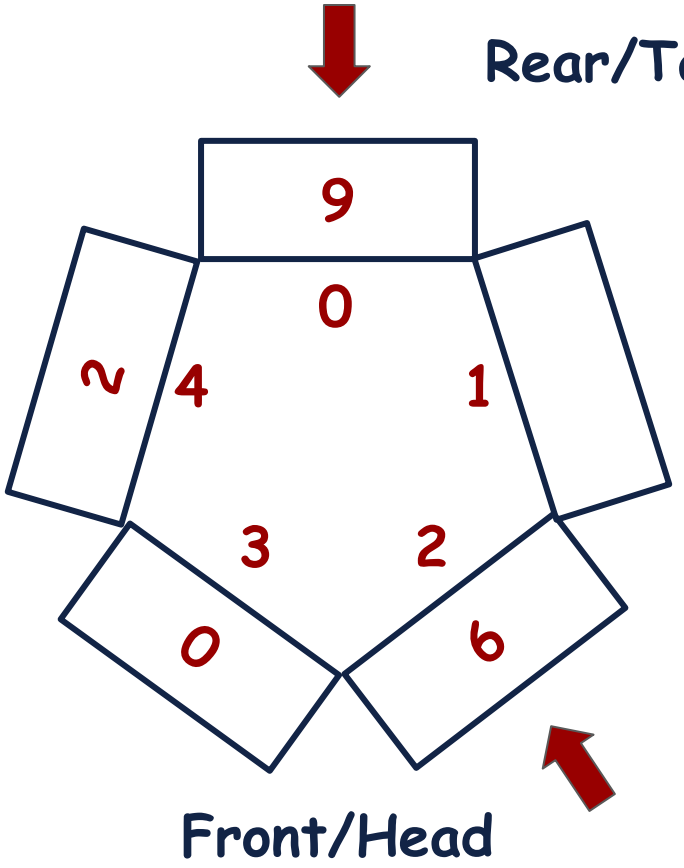


Circular Queue

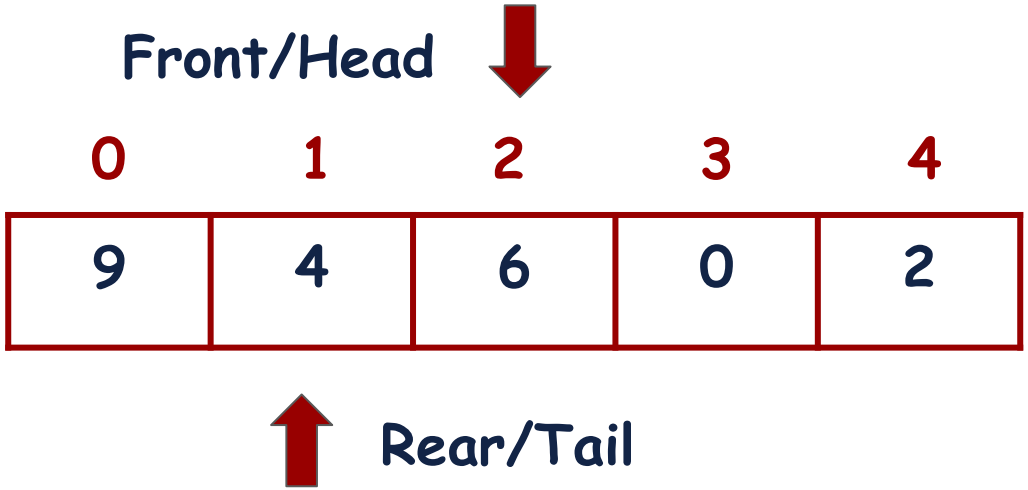


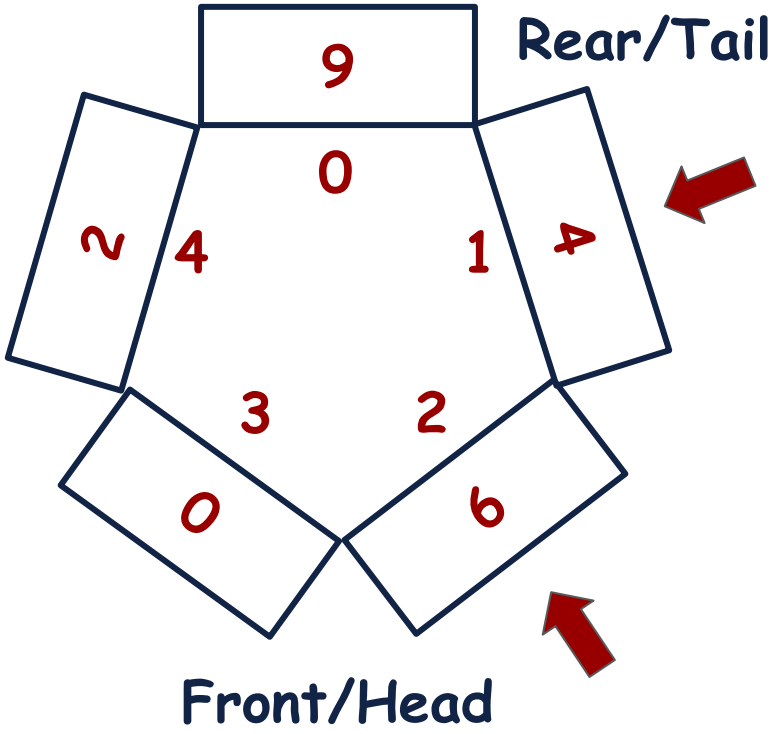
Circular Queue: Enqueue another element





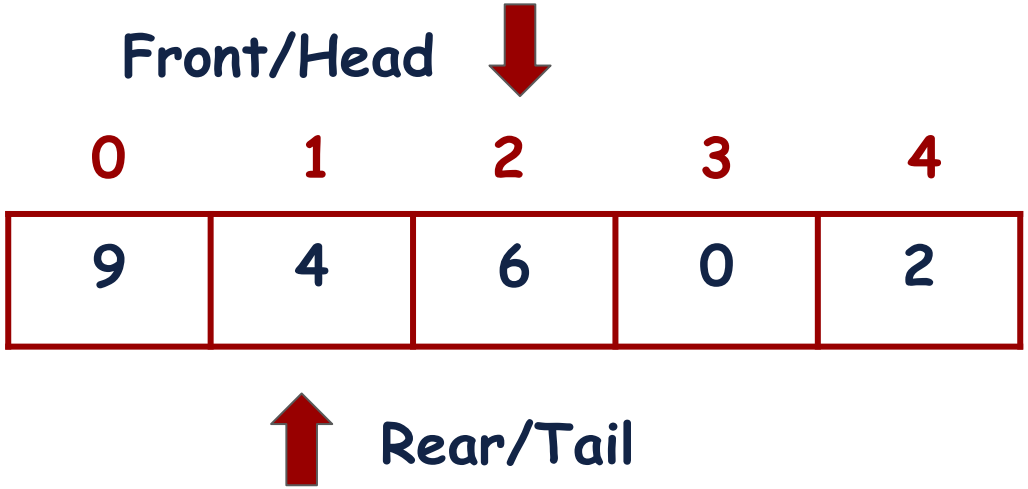
Circular Queue: Enqueue another element

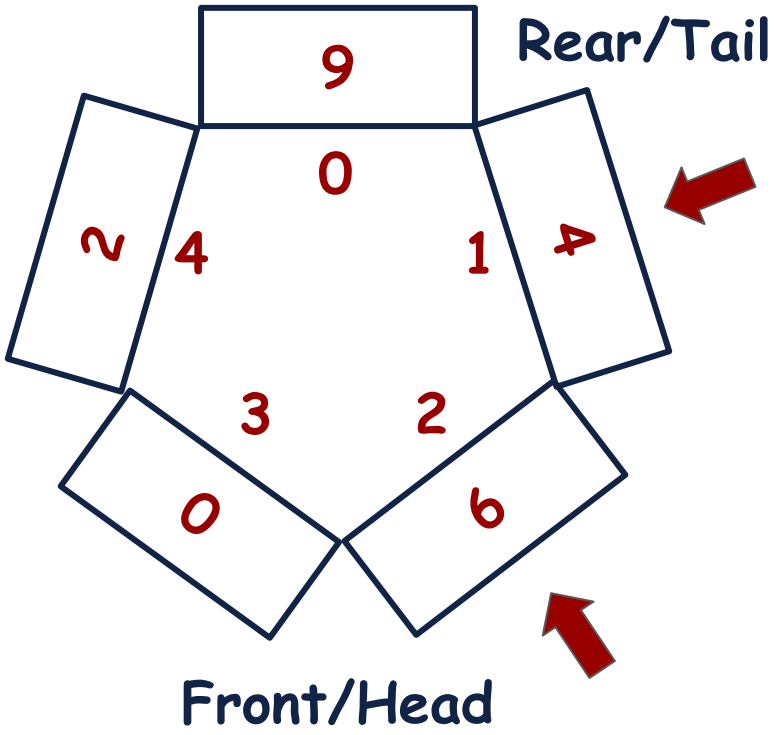




Circular Queue

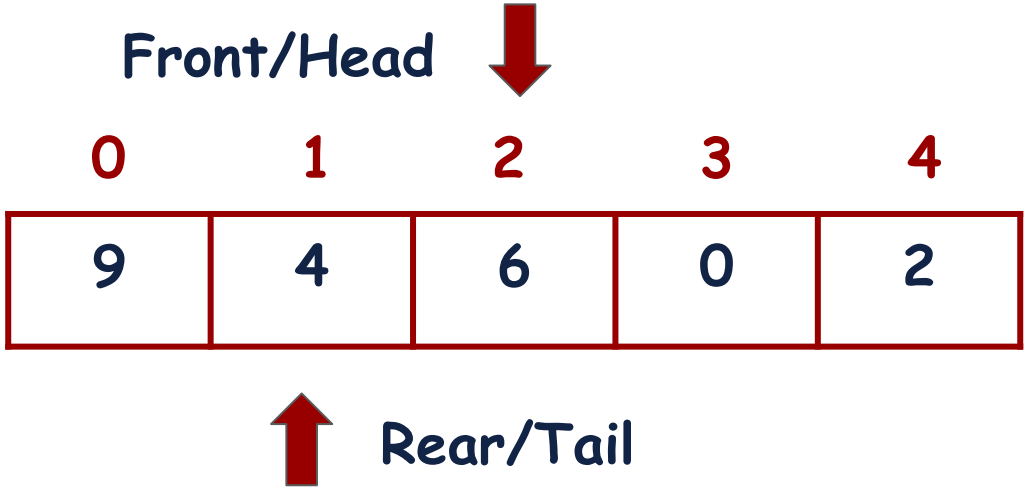
Now what will be the update condition for rear pointer?

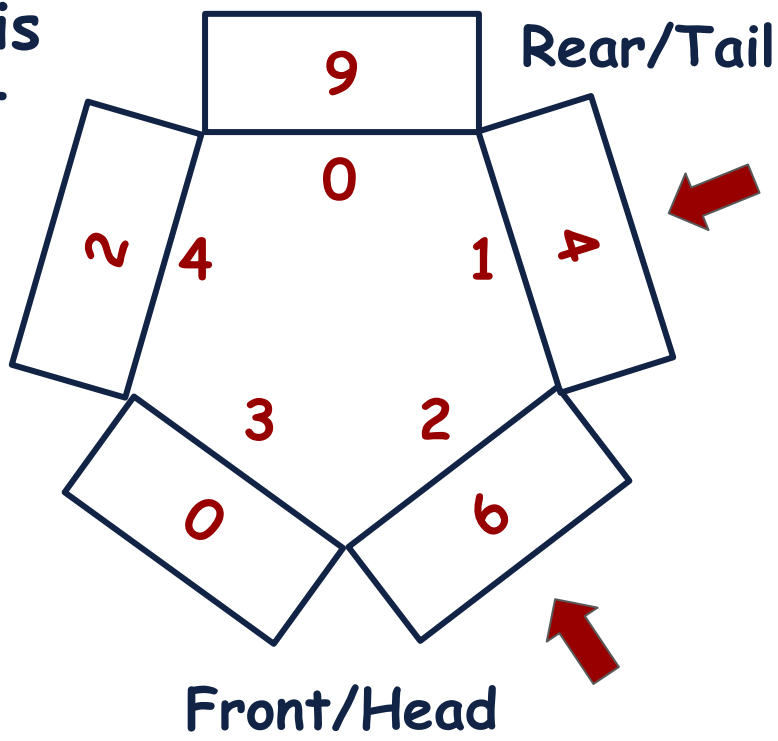




Circular Queue

When the rear pointer reaches the end of the array and there is empty space at the start then it starts from 0 index again.

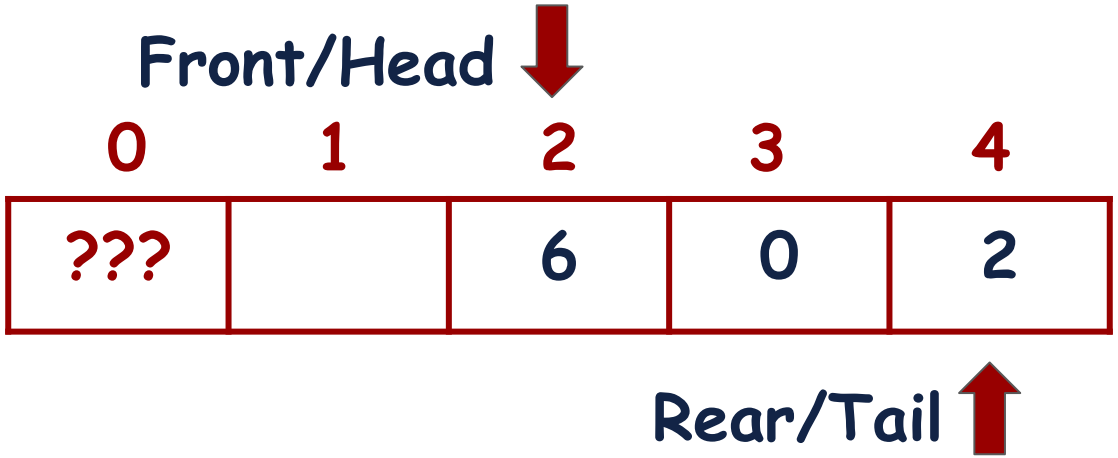




Circular Queue: Rear update Condition

In order to go to the start of the array when Rear reaches the end of the array, the general formula is:

```
rear = (rear + 1) % arrSize
```



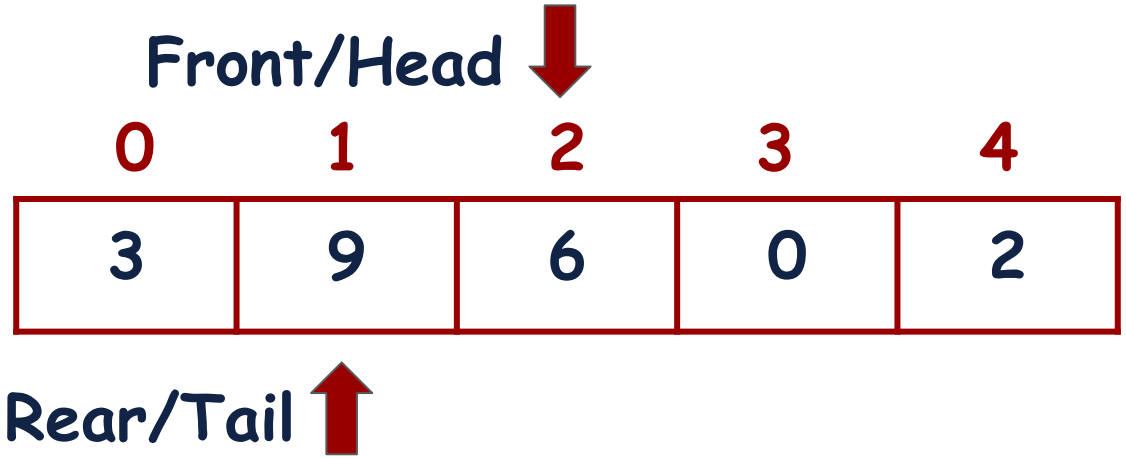
Circular Queue: isFull Condition

Now, what will be the condition to check if the Array is full or not?

Circular Queue: isFull Condition

Now, what will be the condition to check if the Array is full or not?

Case 1:

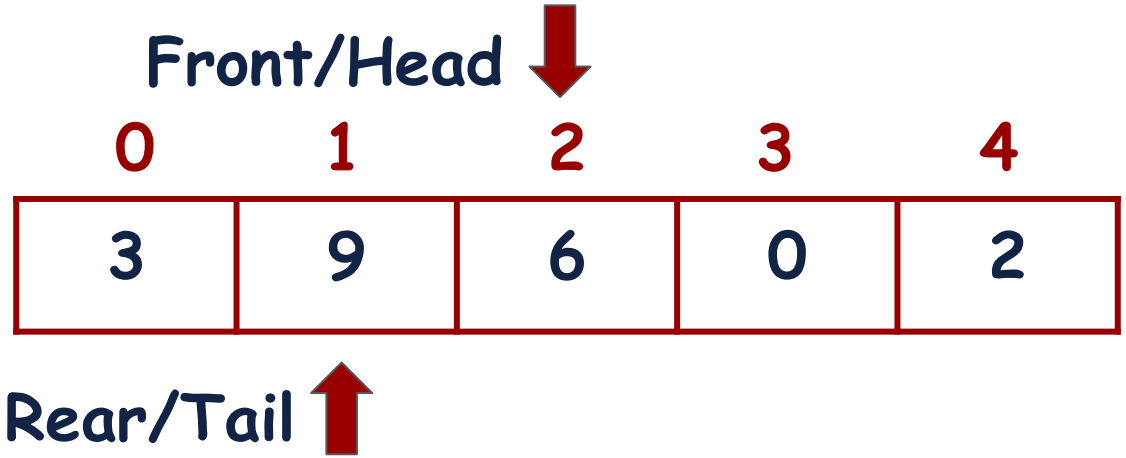


Circular Queue: isFull Condition

Now, what will be the condition to check if the Array is full or not?

front == rear + 1

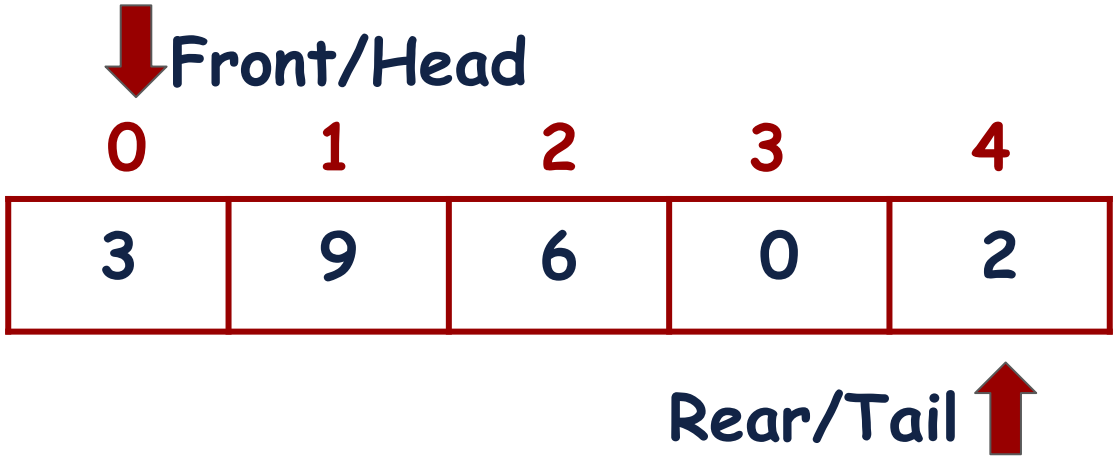
Case 1:



Circular Queue: isFull Condition

Now, what will be the condition to check if the Array is full or not?

Case 2:

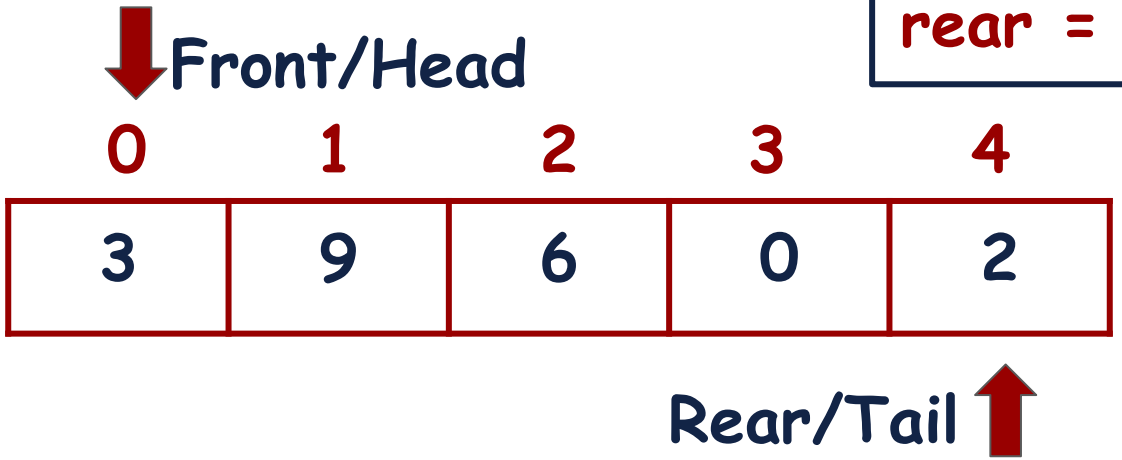


Circular Queue: isFull Condition

Now, what will be the condition to check if the Array is full or not?

```
front == 0
&&
rear = arrSize - 1
```

Case 2:



Circular Queue: Implementation

```
const int MAX = 5;
class CircularQueue
{
    int myQueue[MAX];
    int front, rear;

public:
    CircularQueue()
    {
        front = -1;
        rear = -1;
    }
```

```
bool isEmpty()
{
    if (front == -1)
    {
        return true;
    }
    return false;
}

bool isFull()
{
    if ((front == 0 && rear == MAX - 1) ||
        (front == rear + 1))
    {
        return true;
    }
    return false;
}
```

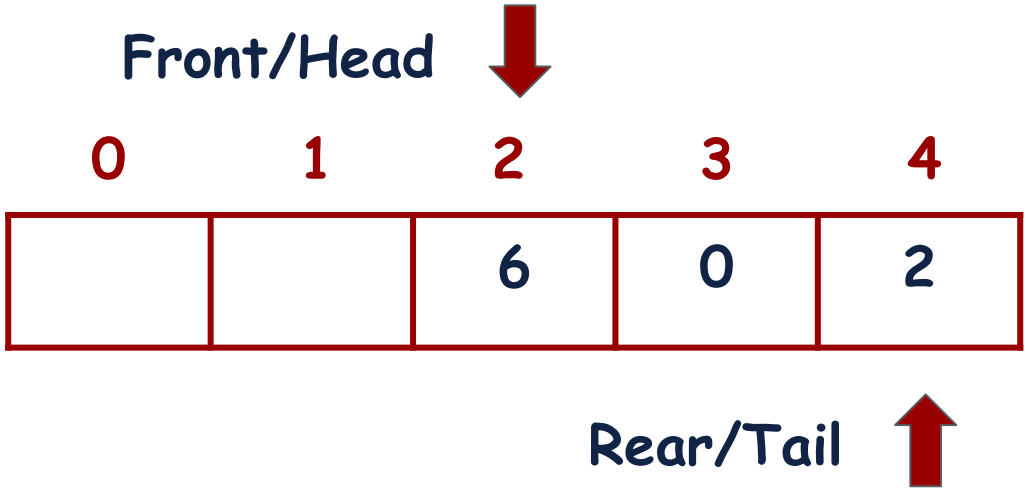
Circular Queue: Implementation

```
bool enQueue(int value)
{
    if (isFull())
    {
        cout << "Queue is Full" << endl;
        return false;
    }
    if (isEmpty())
    {
        front++;
    }
    rear = (rear + 1) % MAX;
    myQueue[rear] = value;
    return true;
}
```

```
bool deQueue()
{
    if (isEmpty())
    {
        cout << "Queue is Empty" << endl;
        return false;
    }
    int value = myQueue[front];
    if (front == rear)
    {
        front = -1;
        rear = -1;
        return true;
    }
    front = (front + 1) % MAX;
    return true;
}
```

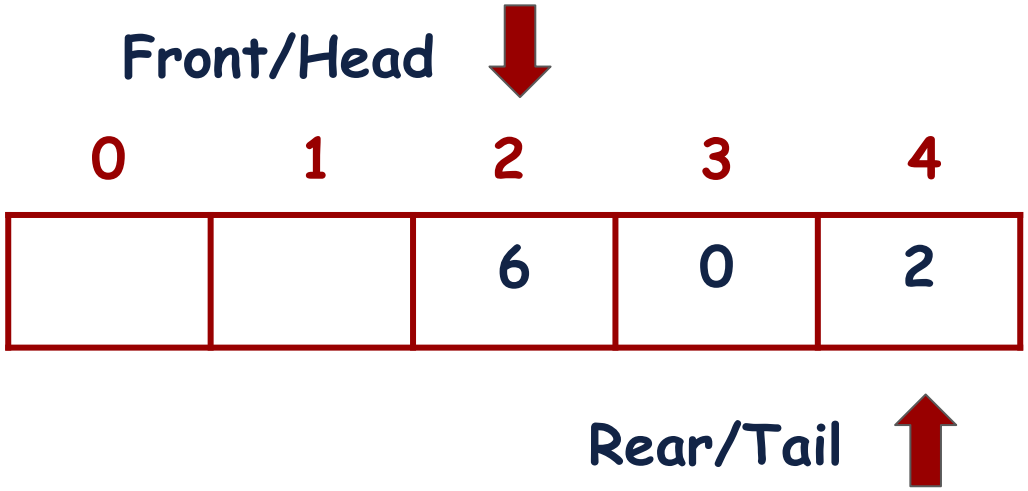
Circular Queue: Array implementation

Still the limitation of this circular queue implemented with array is that the **size** of the circular queue is **fixed**.



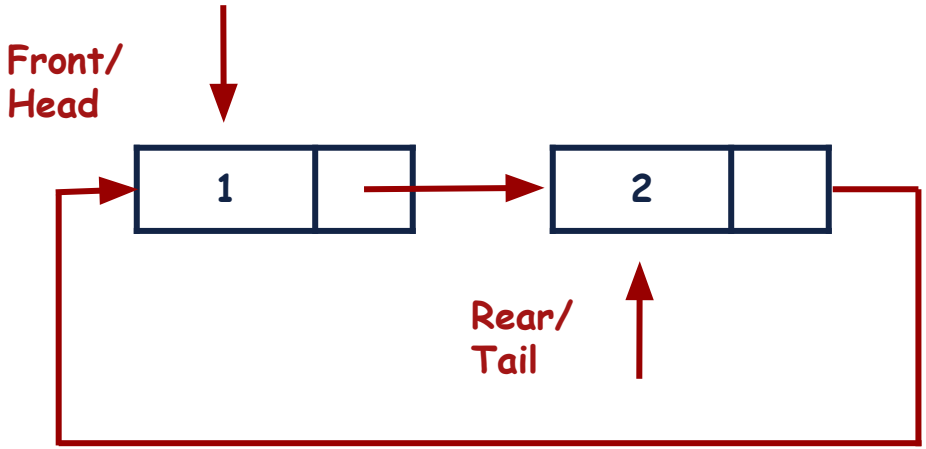
Circular Queue: LinkedList implementation

This issue can be solved with the implementation using **Linked List**.



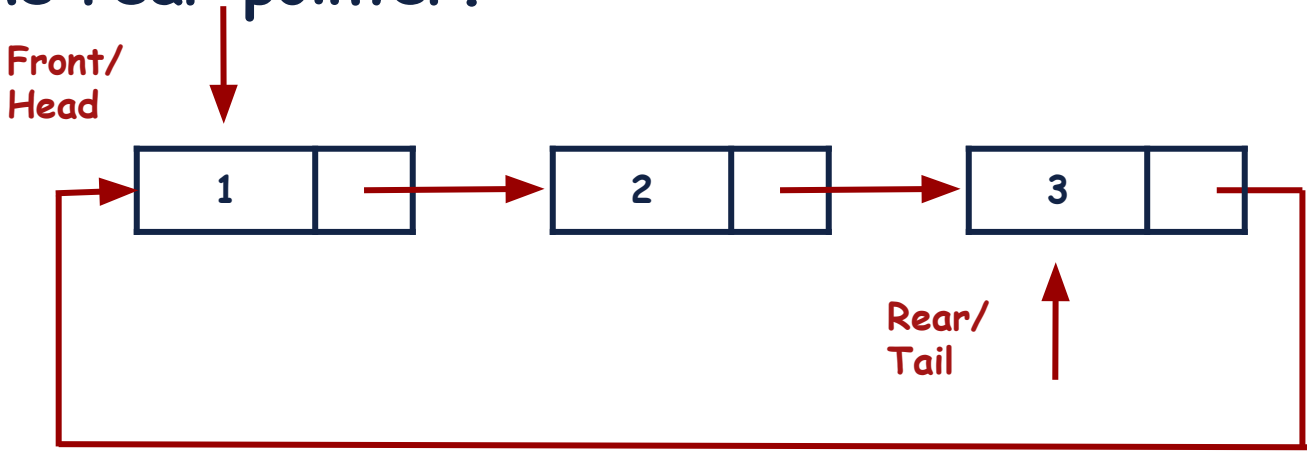
Circular Queue: LinkedList implementation

Now, instead of pointing to the NULL, rear will point towards the front of the linked list.



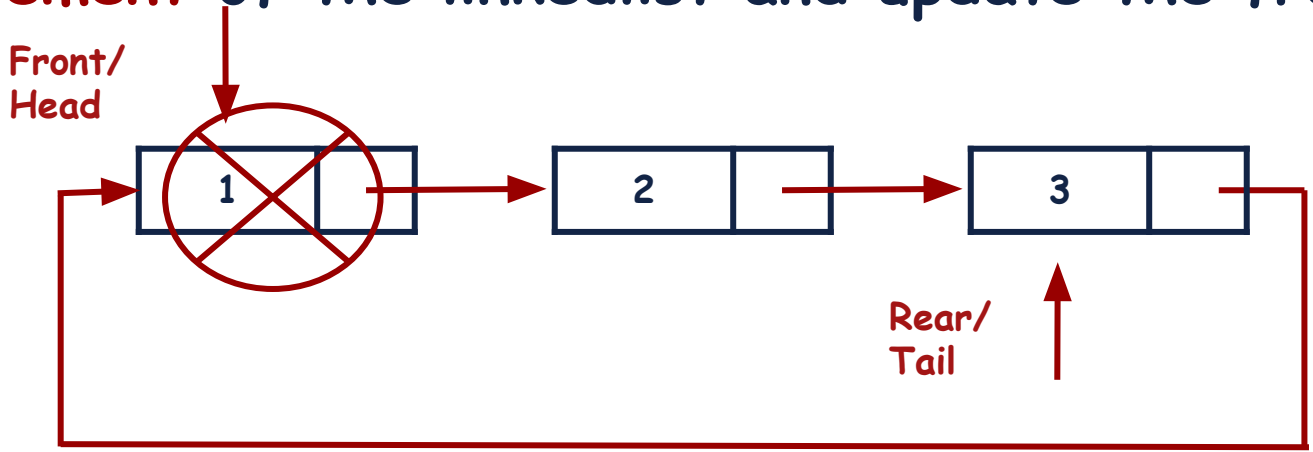
Circular Queue: LinkedList implementation

When we add an element we just add at the end of the linked list and next pointer will point to the front and update the rear pointer.



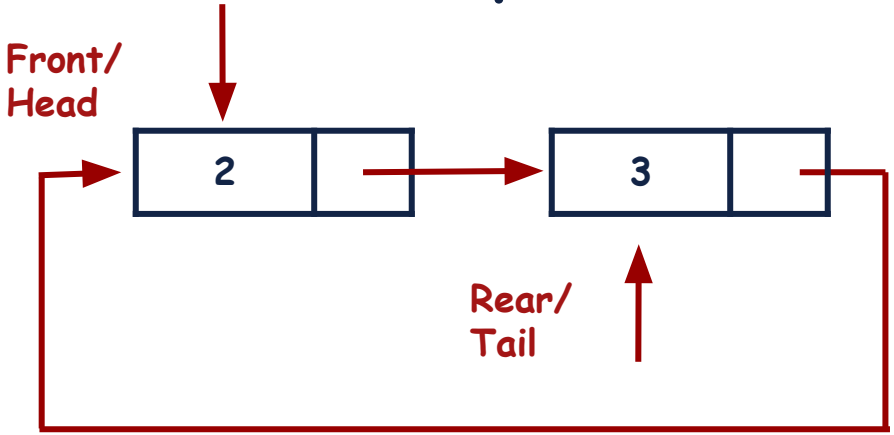
Circular Queue: LinkedList implementation

When we delete an element, we just remove the node at the start of the linked list and rear will point to the second element of the linkedlist and update the front pointer.



Circular Queue: LinkedList implementation

When we delete an element, we just remove the node at the start of the linked list and rear will point to the second element of the linkedlist and update the front pointer.



Learning Objective

Students should be able to **recognize** real life problems where **Circular Queue** data structure is appropriate to solve the problem efficiently.

Self Assessment

1. https://leetcode.com/problems/design-circular-queue/

   Solve this problem both with Array and Linked List.

2. https://leetcode.com/problems/find-the-winner-of-the-circular-game/