Queue
Data Structure

Problem: Tickets Counter

You have to implement a menu based program in which you have the option to add a customer, give ticket to a customer and view the customers in the line.

```
----Tickets Counter----
1. Add Customer
2. Give Ticket to the customer
3. View all the Customers waiting in the Line
4. Exit
Enter Option:
```

Problem: Tickets Counter

If you have added 3 customers in the line and you have given ticket to only the first customer then after pressing option 3 output will be:

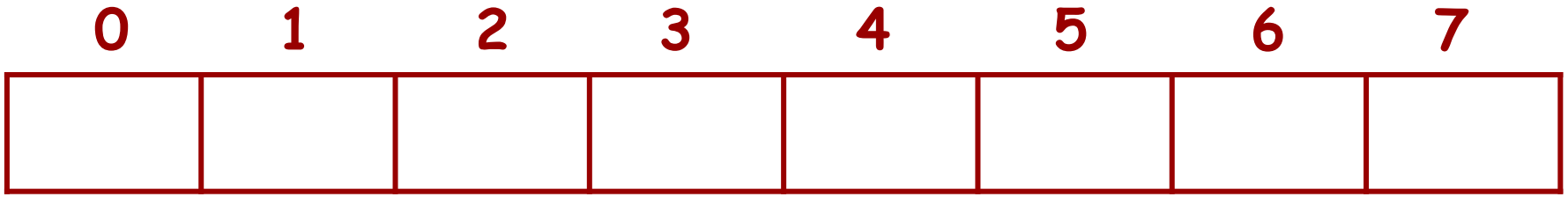
```
----Tickets Counter----
1. Add Customer
2. Give Ticket to the customer
3. View all the Customers waiting in the Line
4. Exit
Enter Option: 3
Customers Line : 2      3
Press any Key to continue
```

Problem: Tickets Counter

How can we implement that?

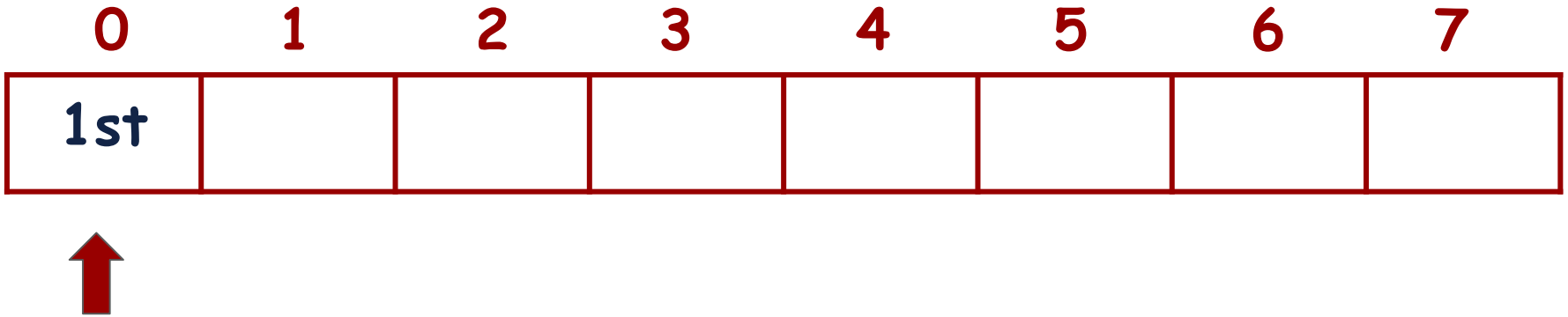
Problem: Tickets Counter

There can be more than one customers therefore we have to store the information of the customers in the Array.



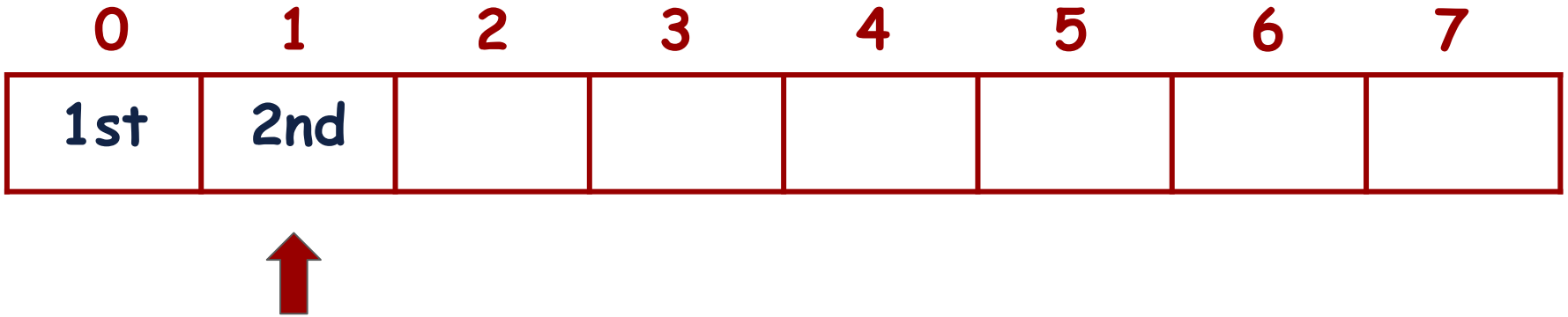
Problem: Tickets Counter

If first customer comes then we add it into the array.



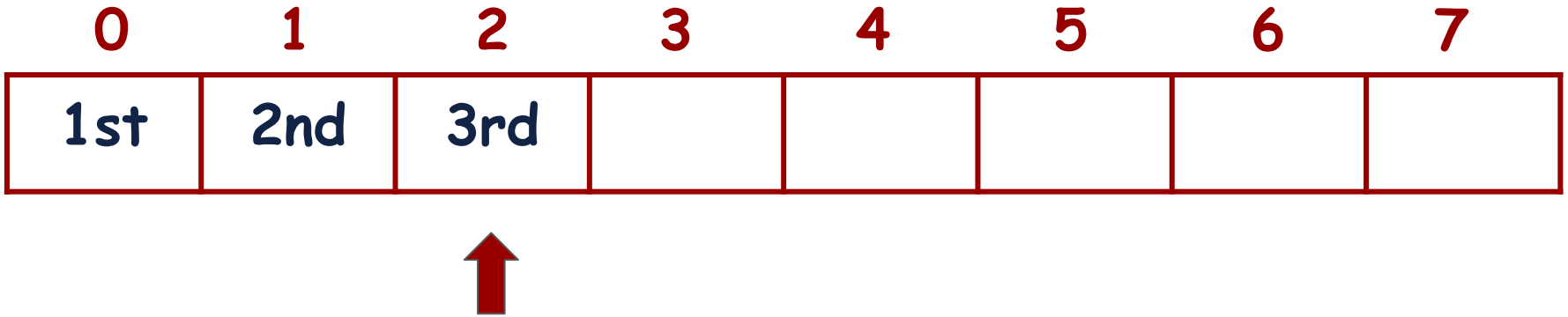
Problem: Tickets Counter

If second customer comes then we add it at the end of the array.



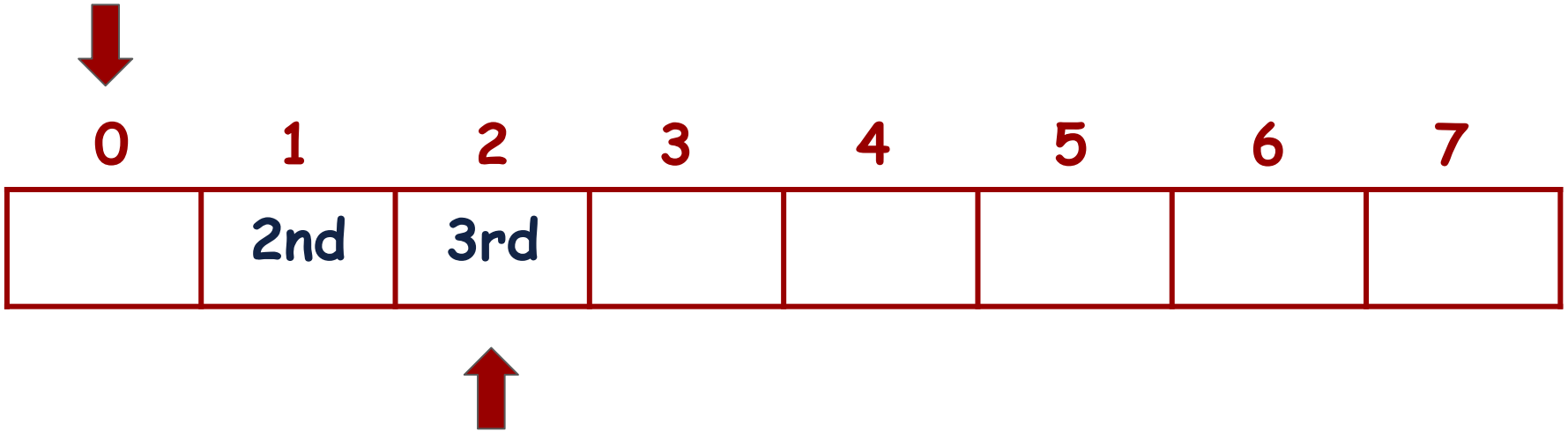
Problem: Tickets Counter

If third customer comes then we add it at the end of the array.



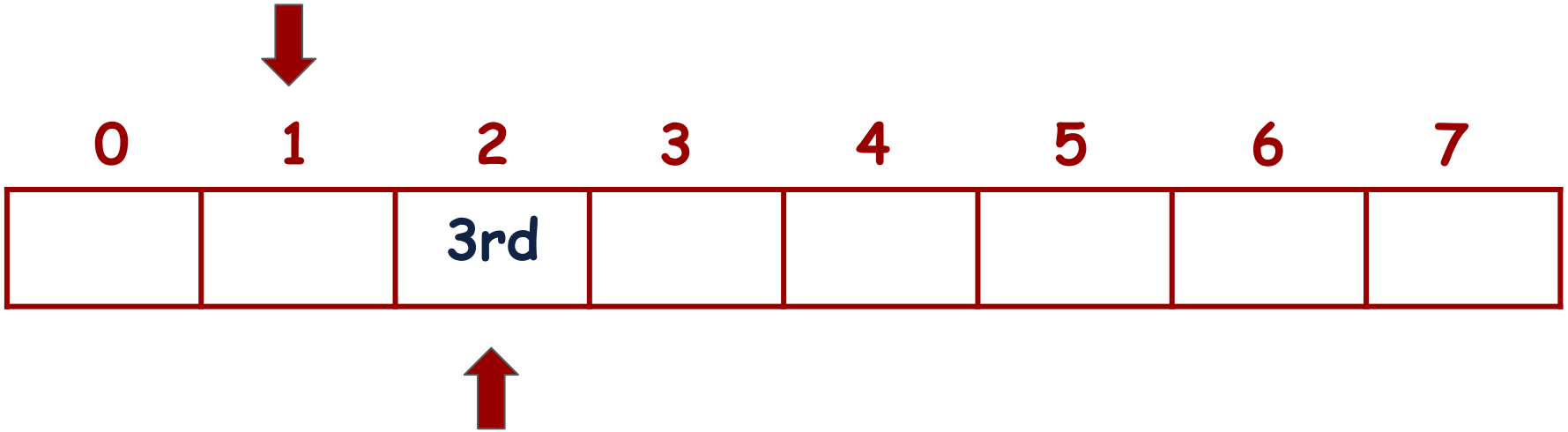
Problem: Tickets Counter

If we give ticket to one customer then we remove the first customer from the start of the array.



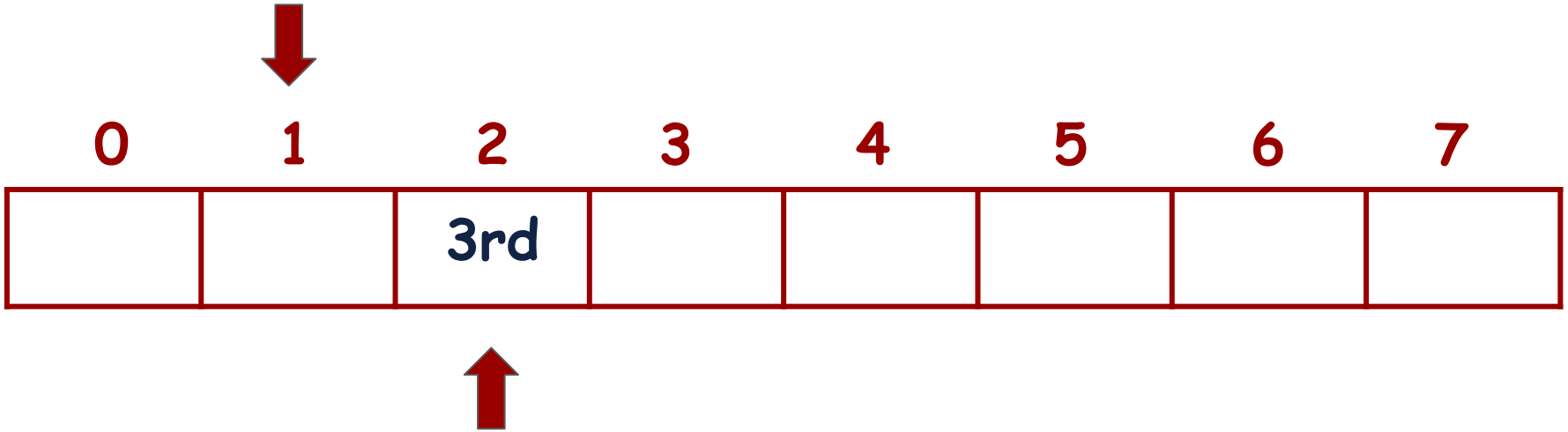
Problem: Tickets Counter

If we give ticket to another customer then we remove the second customer from the start of the array.



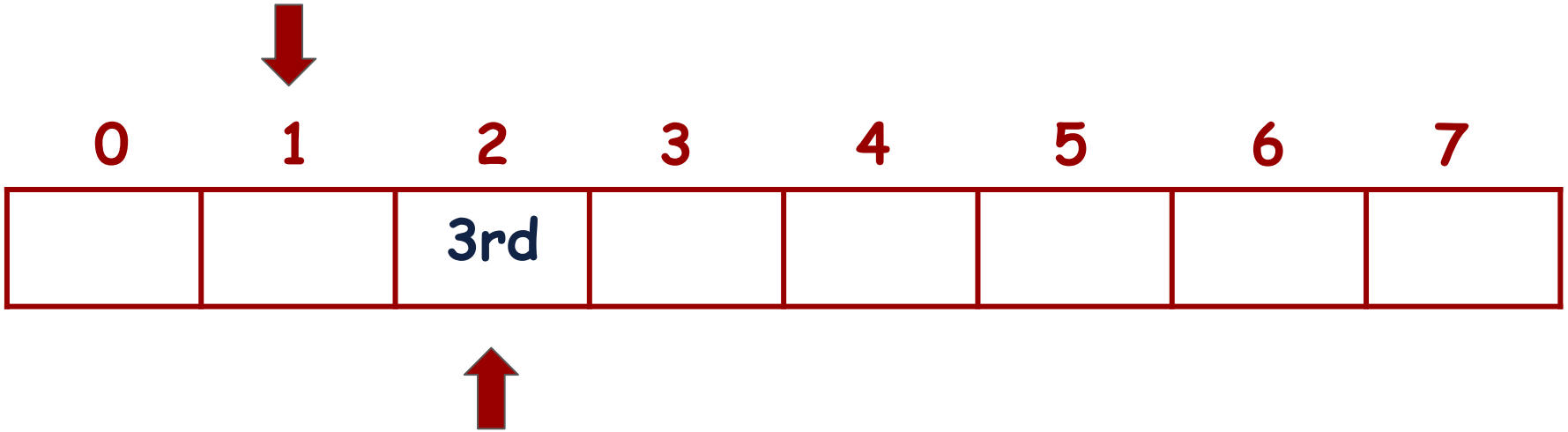
Problem: Tickets Counter

The constraints on the array are that we are adding the new customer at the end of the array and we are removing the customers from the start of the array.



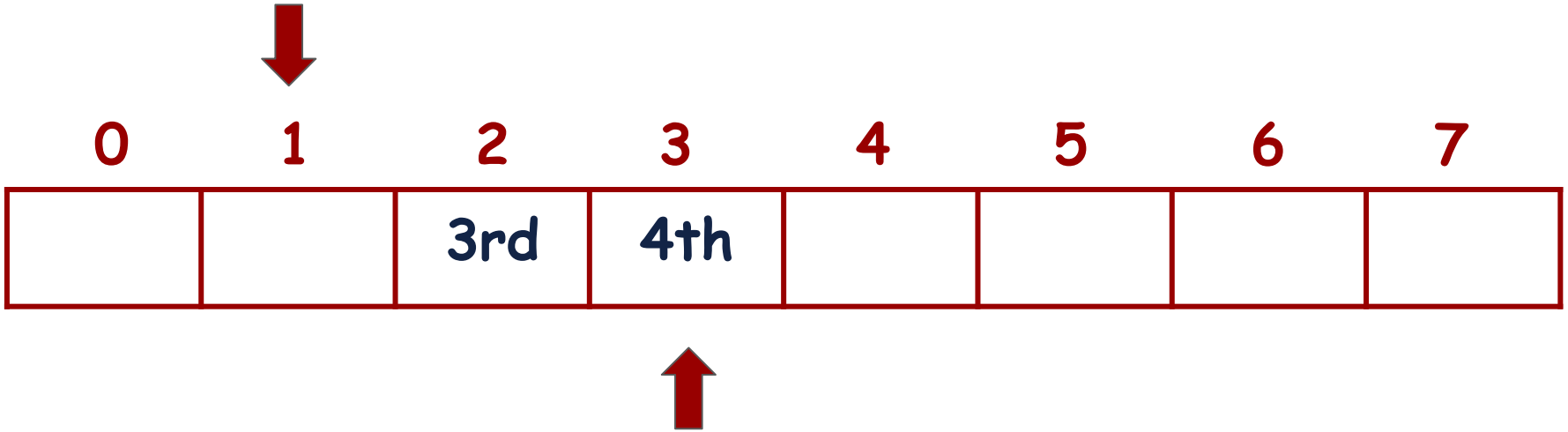
Queue

We consider this a new Data Structure and call it Queue.



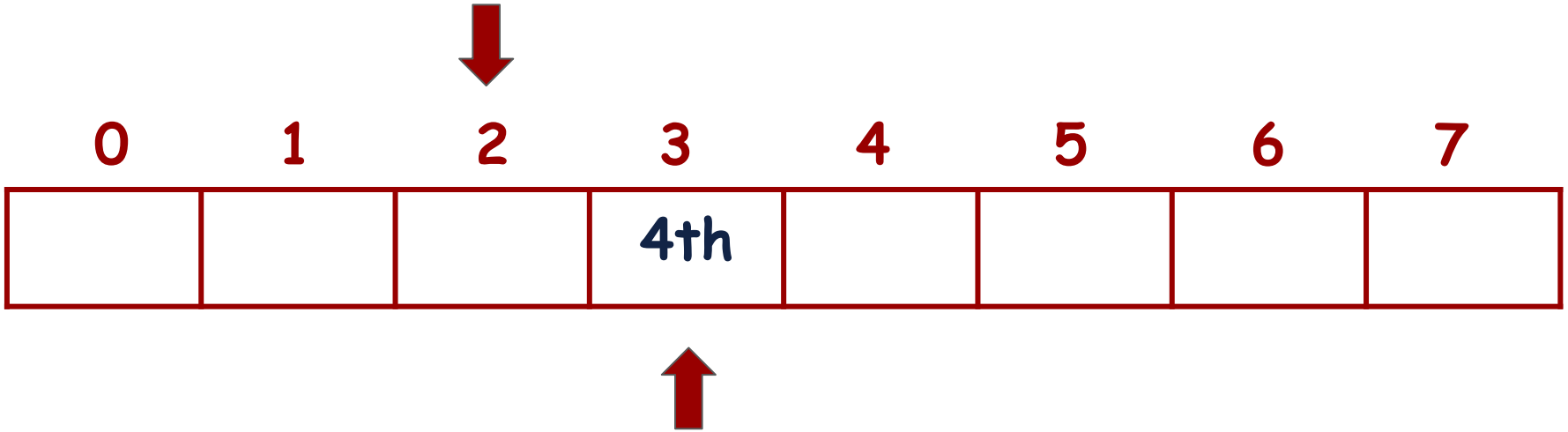
Queue: Rear or Tail

We insert new elements at the end of the Queue called as Rear or Tail.

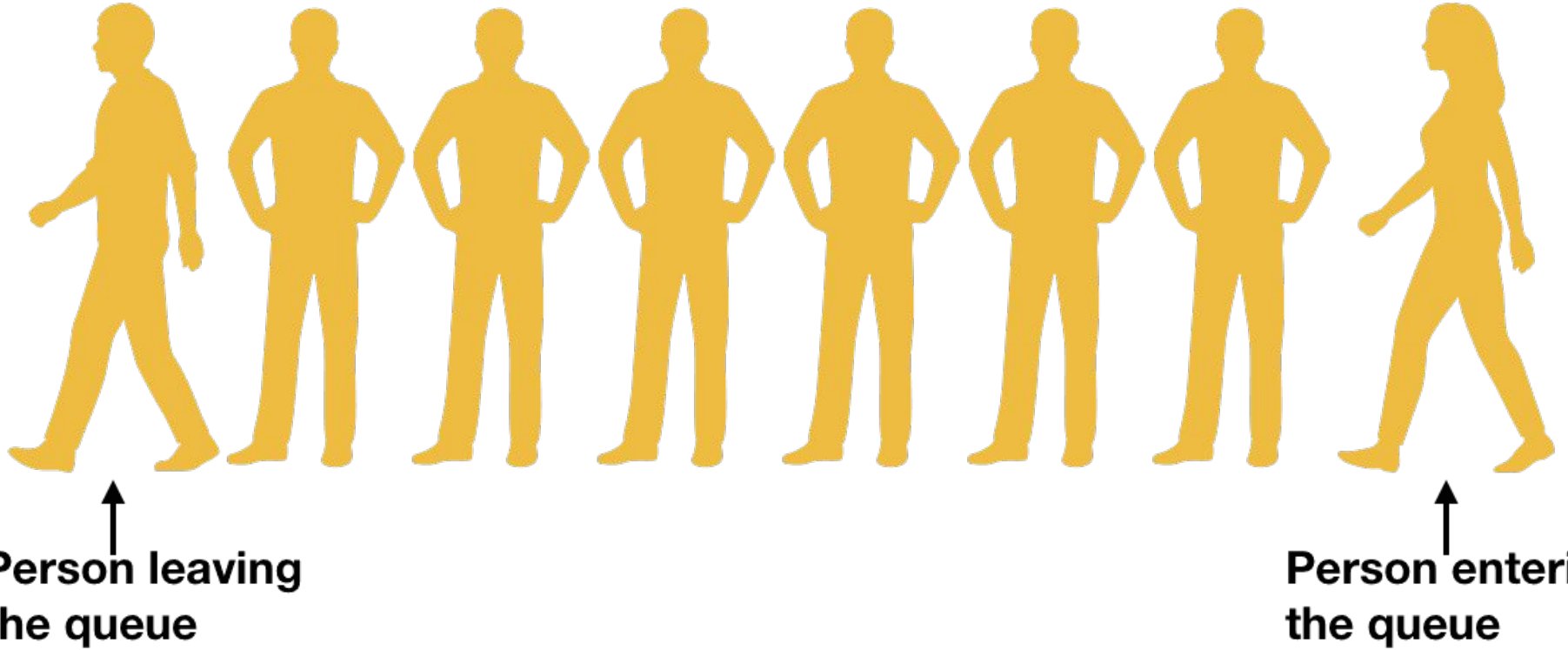


Queue: Front or Head

We delete the elements at the start of the Queue called as Front or Head.

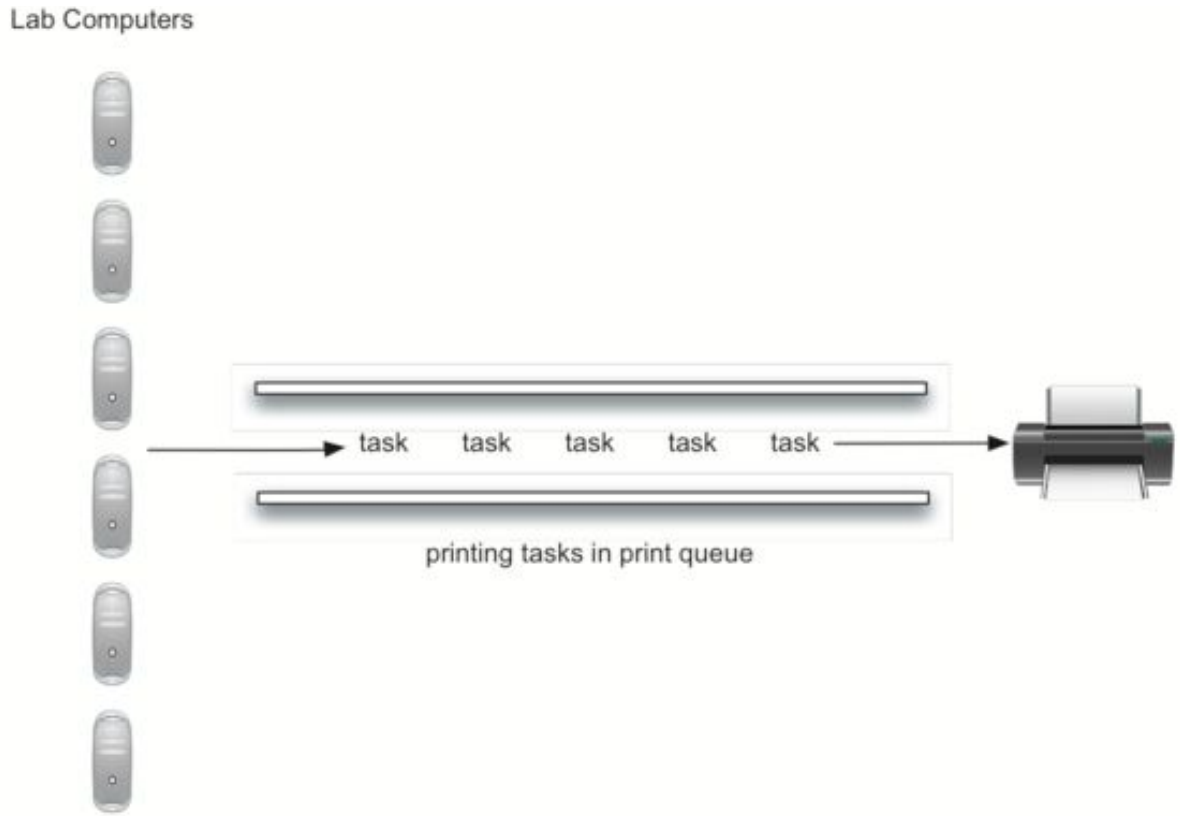


Queue: Real Life Examples



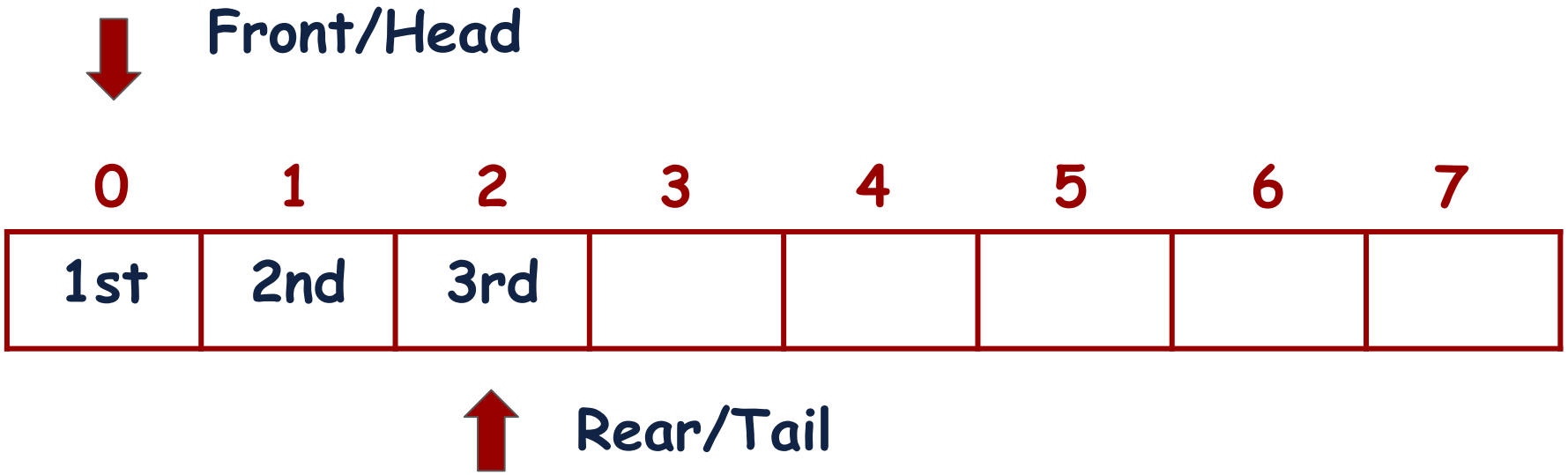
Queue: Real Life Examples

Queue: Real Life Examples



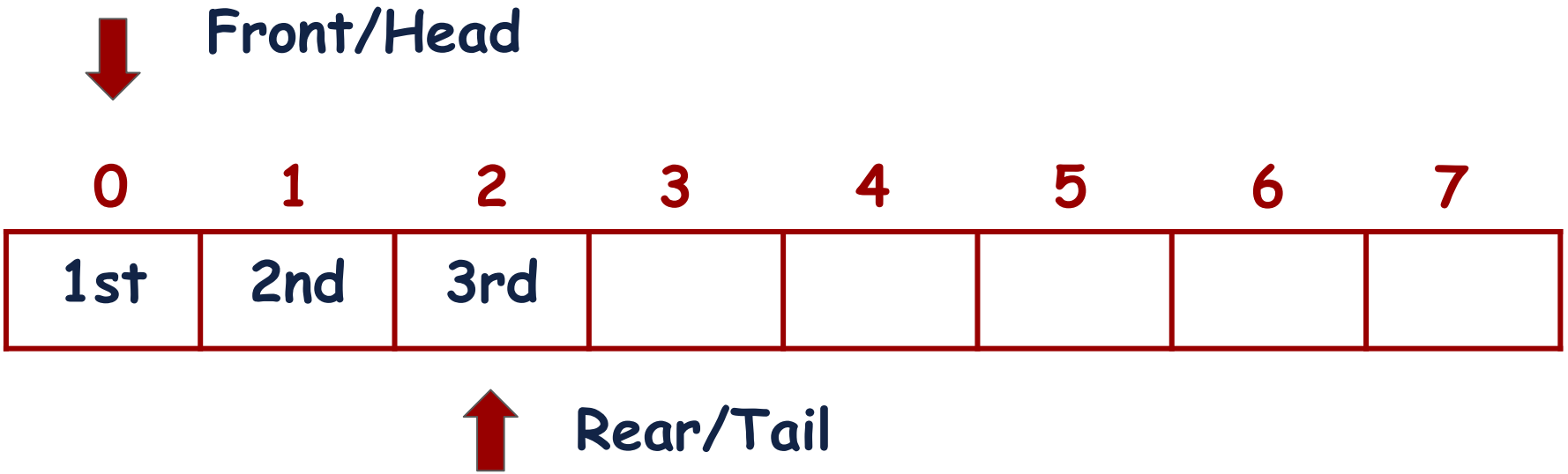
Queue: FIFO

Queue follows FIFO (First In First Out) principle.



Queue: Operations on Queue

We have to add the elements at the Tail of the Queue and we have to remove the elements at the Head of the Queue.



Queue: Operations on Queue

We have to add the elements at the Tail of the Queue and we have to remove the elements at the Head of the Queue.

In Technical Terms:

- Add == Enqueue
- Delete == Dequeue

Queue: Operations on Queue

We have to add the elements at the Tail of the Queue and we have to remove the elements at the Head of the Queue.

In Technical Terms:

- Add == Enqueue
- Delete == Dequeue

Queue: Operations on Queue

We can now apply other operations of the Queue as well

- View whether the Queue is empty or not
- View all the available elements of the Queue

Queue: Implementation using Array

Make a Class named Queue and add Enqueue, Dequeue, isEmpty and View functions.

Queue: Implementation using Array

```
const int MAX_SIZE = 10;
class Queue
{
    int myqueue[MAX_SIZE], front, rear;

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

```
bool isFull()
{
  if (rear == MAX_SIZE - 1)
  {
       return true;
  }
  return false;
}

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

Queue: Implementation using Array

```
bool enQueue(int value)
    {
        if (isFull())
        {
            cout << "Queue is full!!";
            cout << endl;
            return false;
        }
        else
        {
            if (isEmpty())
            {
                front = 0;
            }
            rear = rear + 1;
            myqueue[rear] = value;
            return true;
        }
    }
```

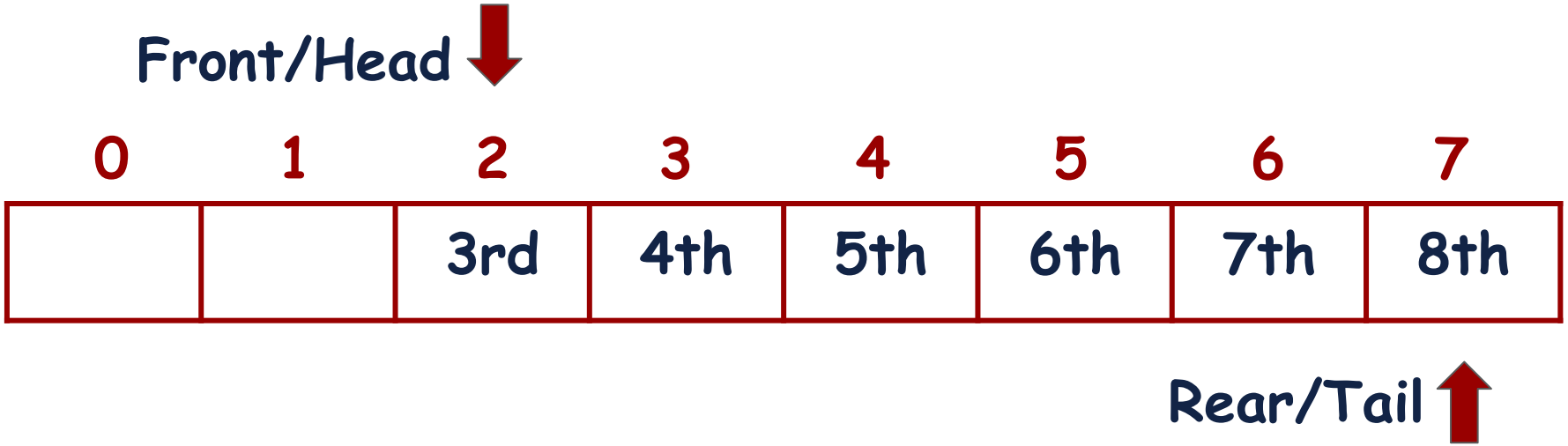
```
int deQueue()
    {
        int value;
        if (isEmpty())
        {
            cout << "Line is empty!!" << endl;
            return -1;
        }
        else
        {
            value = myqueue[front];
            if (front >= rear)
            { // only one element in queue
                front = -1;
                rear = -1;
            }
            else
                front++;
            return value;
        }
    }
```

Queue: Implementation using Array

Do you see any problem with the implementation of Queue using Array?

Queue: Implementation using Array

Although there is space at the start of the array but when we'll try to add the element, it will give us the error that Queue is Full.

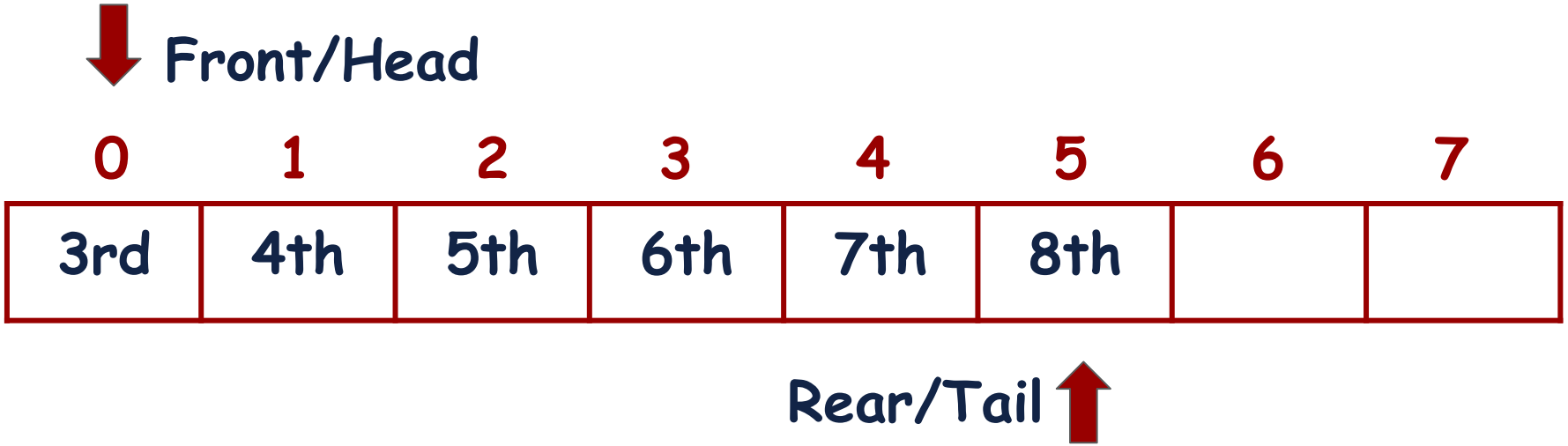


Queue: Implementation using Array

How can we Solve this problem of **Queue is full** issue although there is empty space present in the Array?

Queue: Implementation using Array

Whenever we delete an element we push all the elements to one previous index and decrement 1 in the rear.



Queue: Implementation using Array

```
int deQueue()
    {
        int value;
        if (isEmpty())
        {
            cout << "Queue is empty!!" << endl;
            return -1;
        }
        else
        {
            value = myqueue[front];
            for(int x = 0; x < rear; x++)
            {
                myqueue[x] = myqueue[x+1];
            }
            rear--;
            return value;
        }
    }
```

Queue: Implementation using Array

But this solution comes with extra computational cost (i.e., extra time will be required).

Queue: Implementation using LinkedList

So why go through all the fuss when we can implement the queue using LinkedList.

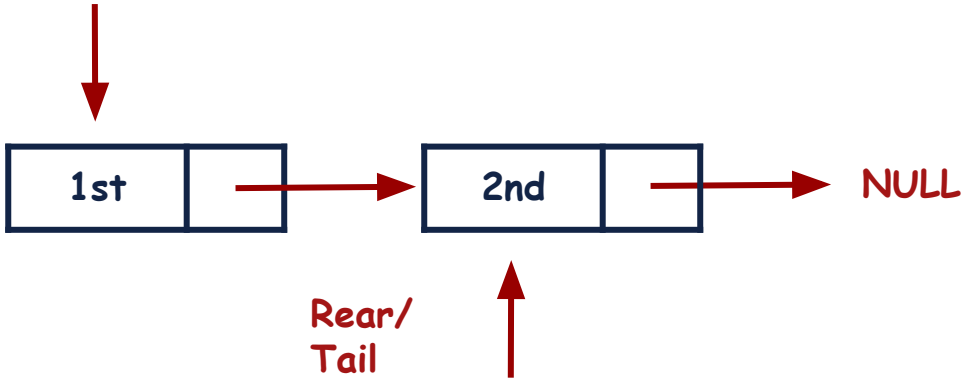
GOOD

T

coming

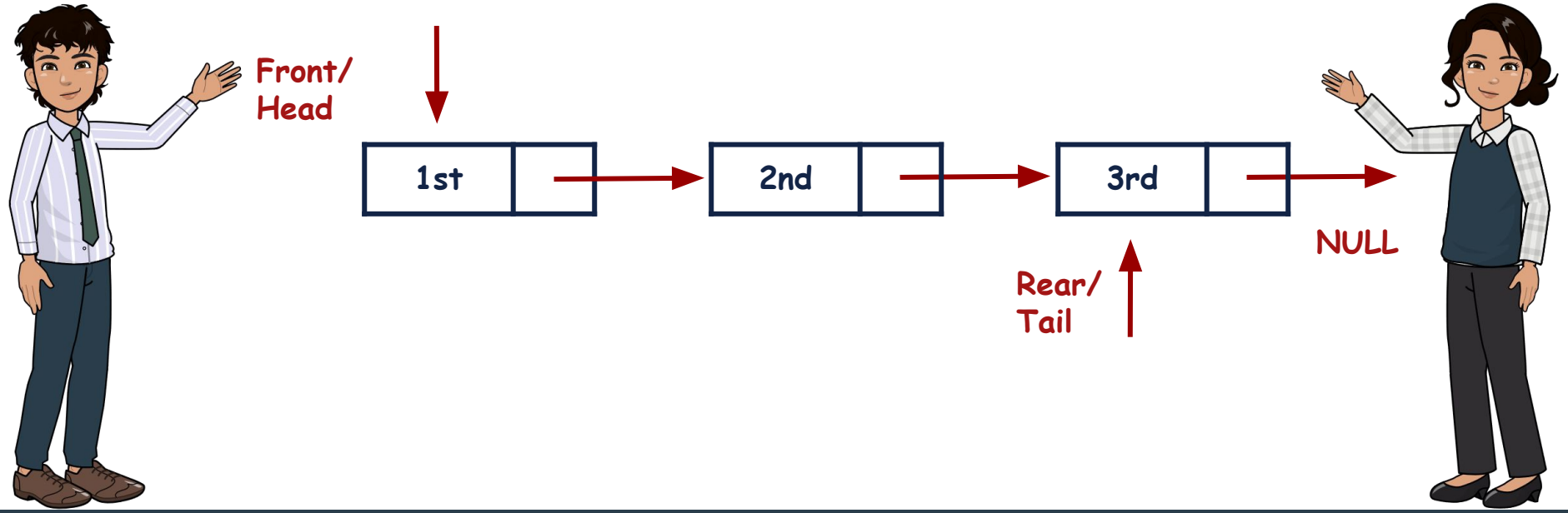
Queue: Implementation using LinkedList

So why go through all the fuss when we can implement the queue using **LinkedList**.



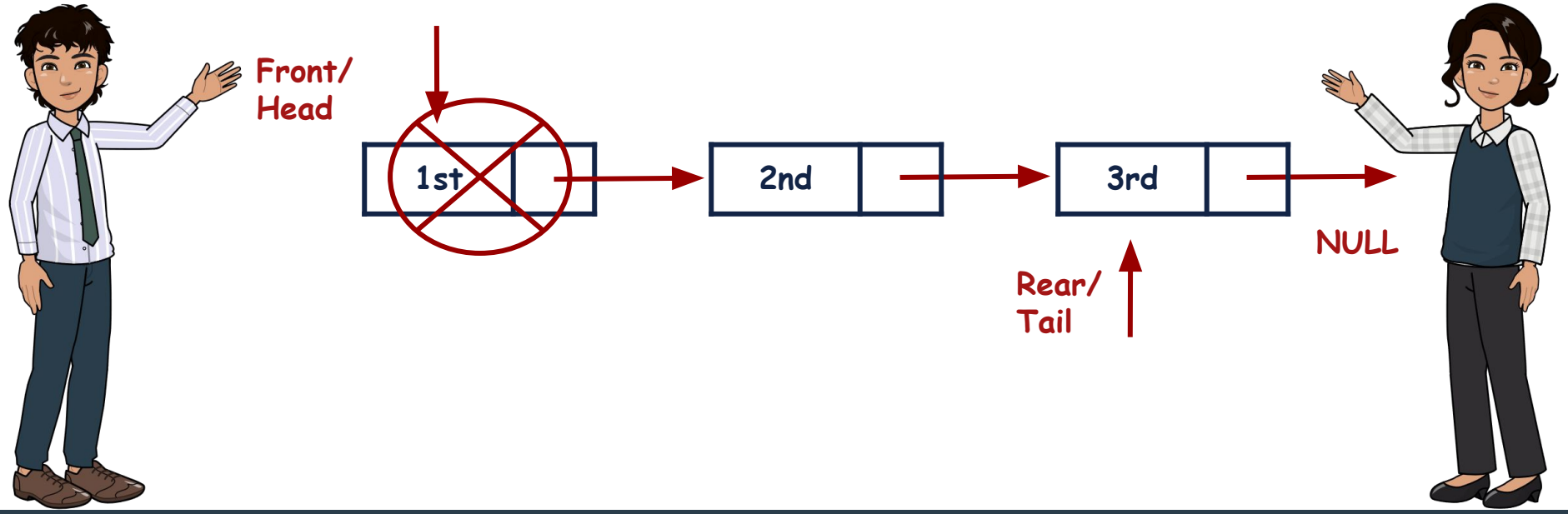
Queue: Implementation using LinkedList

When we **Enqueue** an element we just add at the end of the linkedlist and update the rear pointer.



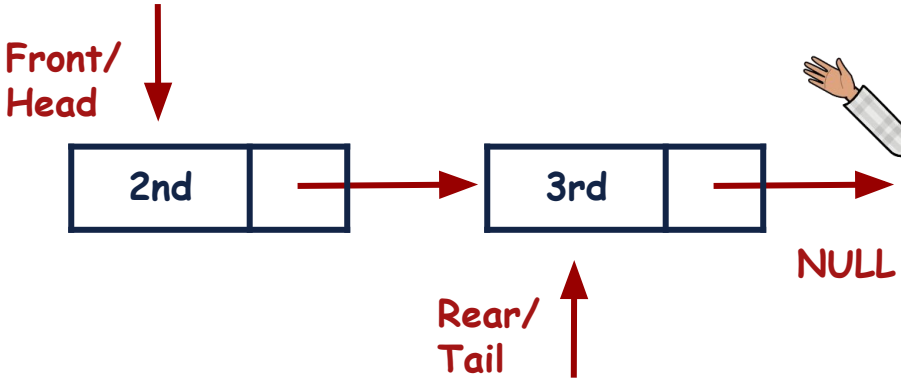
Queue: Implementation using LinkedList

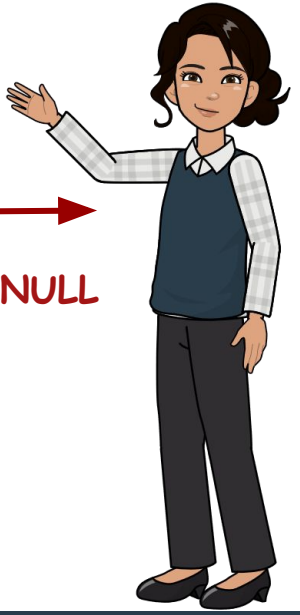
When we **Dequeue**, we just remove the node at the start of the linkedlist and update the front pointer.



Queue: Implementation using LinkedList

When we Dequeue, we just remove the node at the start of the linkedlist and update the front pointer.





Queue: Implementation using LinkedList

```
struct node
{
     int data;
     node *next;
};

class Queue
{
     node *front;
     node *rear;

public:
     Queue()
     {
          front = NULL;
          rear = NULL;
     }
```

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

Queue: Implementation using LinkedList

```
bool enQueue(int item)
    {
        node *record = new node();
        record->data = item;
        record->next = NULL;
        if (front == NULL)
        {
            front = record;
            rear = record;
        }
        else
        {
            rear->next = record;
            rear = record;
        }
        return true;
    }
```

```
int deQueue()
    {
        if (isEmpty())
        {
            cout << "Queue is Empty" << endl;
            return 0;
        }
        else
        {
            node *temp = front;
            int item = temp->data;

            front = front->next;
            delete temp;
            return item;
        }
    }
```

Queue: Implementation using LinkedList

```
void displayQueue()
    {
        node *temp = front;
        if (isEmpty())
        {
            cout << "Queue is Empty" << endl;
        }
        else
        {
            while (temp != NULL)
            {
                cout << temp->data << "\t";
                temp = temp->next;
            }
            cout << endl;
        }
    }
};
```

Learning Objective

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

Self Assessment

There are n people in a line queuing to buy tickets, where the 0th person is at the front of the line and the (n - 1)th person is at the back of the line.
You are given a 0-indexed integer array tickets of length n where the number of tickets that the ith person would like to buy is tickets[i].

Each person takes exactly 1 second to buy a ticket. A person can only buy 1 ticket at a time and has to go back to the end of the line (which happens instantaneously) in order to buy more tickets. If a person does not have any tickets left to buy, the person will leave the line.

Return the time taken for the person at position k (0-indexed) to finish buying tickets.

Self Assessment

Example 1:

Input: tickets = [2,3,2], k = 2
Output: 6
Explanation:
- In the first pass, everyone in the line buys a ticket and the line becomes [1, 2, 1].
- In the second pass, everyone in the line buys a ticket and the line becomes [0, 1, 0].
The person at position 2 has successfully bought 2 tickets and it took 3 + 3 = 6 seconds.

Self Assessment

Example 2:

Input: tickets = [5,1,1,1], k = 0
Output: 8
Explanation:
- In the first pass, everyone in the line buys a ticket and the line becomes [4, 0, 0, 0].
- In the next 4 passes, only the person in position 0 is buying tickets.
The person at position 0 has successfully bought 5 tickets and it took 4 + 1 + 1 + 1 + 1 = 8 seconds.

Self Assessment

Constraints:

- n == tickets.length
- 1 <= n <= 100
- 1 <= tickets[i] <= 100
- 0 <= k < n