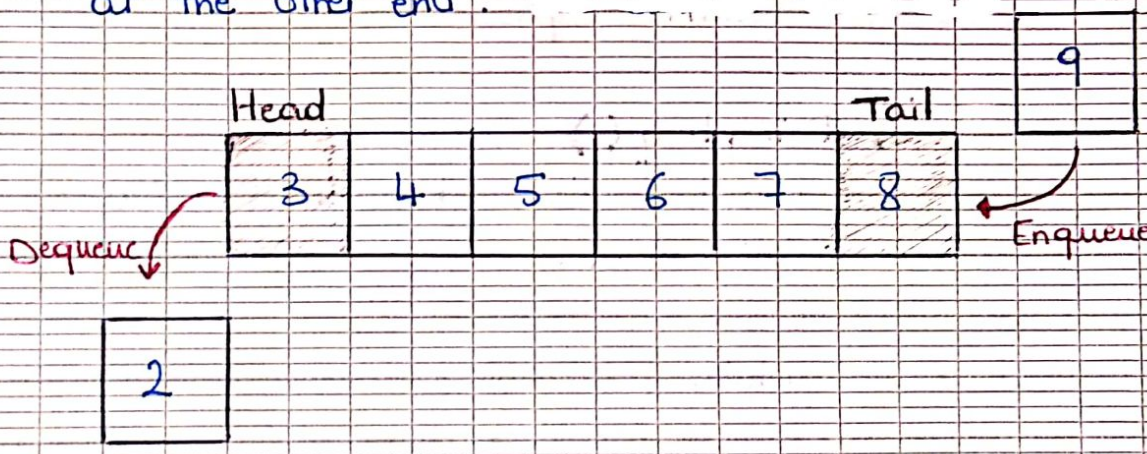
Queue

What is Queue

A queue is a linear data structure that is open at both ends.
We define a queue to be a list in which all additions to the list are made at one end and all deletions from the list are made at the other end.



FIFO (First In First Out)

This strategy state that the first element added to the queue will be the first one to be removed.
It's like standing in a line, where the first person in line is the first to proceed

→ Basic Operations on Queue

- **Enqueue()**: Adds an element to the end of the queue
- **Dequeue()**: Removal of elements from the queue
- **head()**: Returns the item at the head of the queue without removing it
- **tail()**: Returns the item at the tail of the queue without removing it
- **isEmpty()**: Check whether the queue is empty

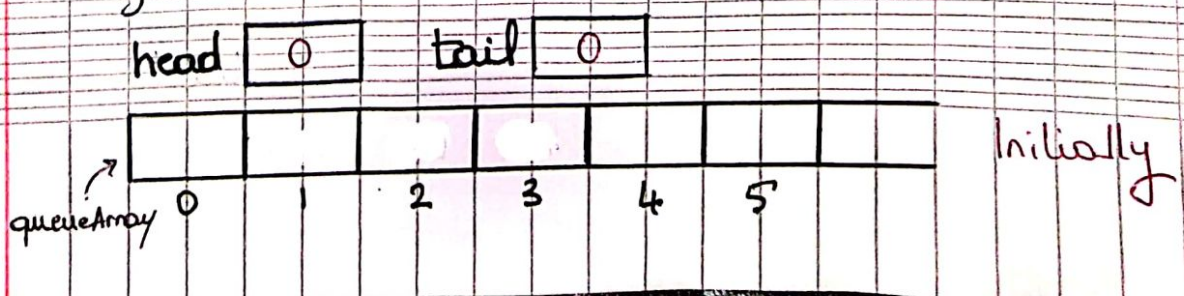
→ Implementation of Queue using Array

We can simply use an Array to implement a queue in this case we will use:

- An array queueArray for sorting the items
- A head that indicates the item at the head
- A tail that indicates the item at the tail

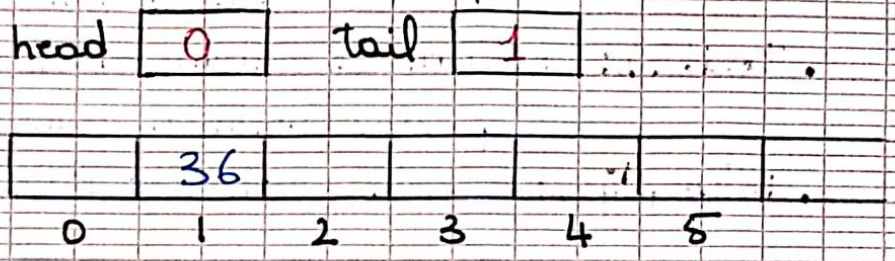
↳ Queue Class (Using Array)

```
public class Queue {
  private int[] queueArray;
  private int head, tail;
  // Constructor to initialize a queue
  public Queue (int size) {
    queueArray = new int [size];
    head = tail = 0;
  }
```



Initially

if we want to add an element to the empty queue that we have initialize, the addition will be at the end so we increament tail to 1 and store the new element in the position 1 and the head will remain 0

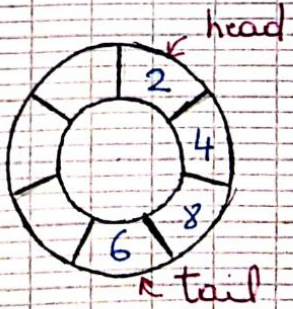


and if we want to add a new element it will be added at index 2 and so on.

However, we notice that the index 0 remain empty so it would be better to use it. This lead us to the idea of using array in a circular way.

→ **Circular Queue Concept**

In a circular queue, when the tail reaches the end of the array, it wraps around the beginning, creating a circular arrangement. This ensures efficient use of space and avoid wasting memory



To implement this circular behavior, we use the modulo operator (%). When the tail reaches the end of the array, tail = (tail+1) % array.length brings it back to the beginning.
This circular arrangment allows the queue to utilize the enlire array without the need for resizing

↳ isEmpty () / isFull ()

```
// Check is the queue isEmply
public boolean isEmply (){
    return head == tail ;
}

// Check is the queue isFull
public boolean isFull (){
    return (tail + 1) % queueArray.length ==head;
}
```

↳ enqueue () Operation

- Approach

1- Check the queue is full or not
2- If full, print overflow and exist
3- If queue not full, increament tail and add the element

• Code:

```
public void enqueue (int item){
    if (!isFull()) {
        tail = (tail +1) % queueArray.length;
        // Circular increment
        queueArray [tail] = item;
    }
    else // full queue : Unable to enqueue
        System.out.println ("The queue is full");
}
```

↳ dequeue() Operation

• Approach:

1- Check queue is empty or not
2- If not empty, print element at the head and increment head
3- If empty print underflow

• Code

```
public int dequeue (){
    if (isEmpty()){ // Empty queue : Unable to dequeue
        System.out.println("The queue is empty");
        System.exist(1);
    }
    else
        head = (head +1) % queueArray.length;
    return queueArray [head];
}
```

- head() / tail()

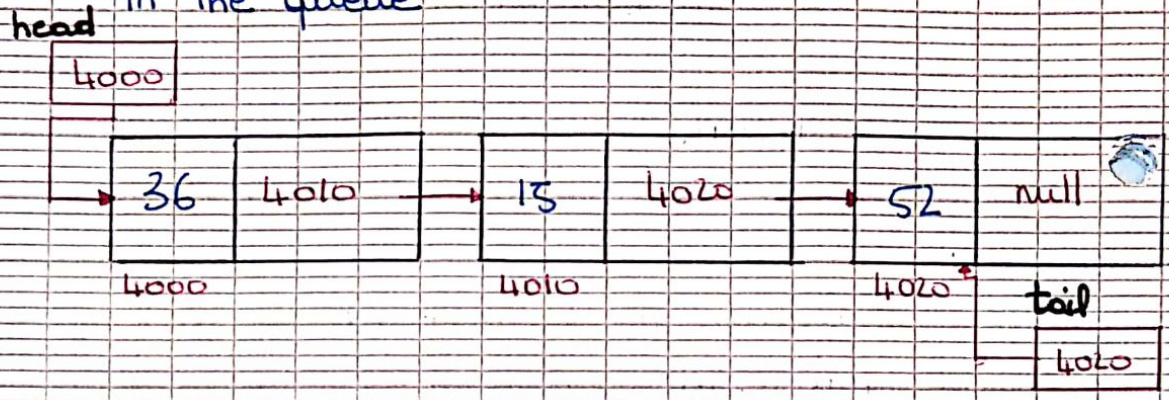
```
// Return the element at the head of the queue
public int head () {
    if (isEmpty()) {
        System.out.println("The queue is empty");
        System.exist(1);
    }
    return queueArray[(head+1) % queueArray.length];
}

// Return the element at the tail of the queue
public int tail () {
    if (isEmpty()) {
        System.out.println("The queue is empty");
        System.exist(1);
    }
    return queueArray[tail];
```

→ **Implementation of Queue Using LinkedList**

As with stacks, we can implement a queue using LinkedLists

In this case we use two links, head and tail to refer to the first and last nodes in the queue



!! We will use the same classes NodeData and Node used in the implementation of the LinkedList class

↳ Queue Class (using LinkedList)

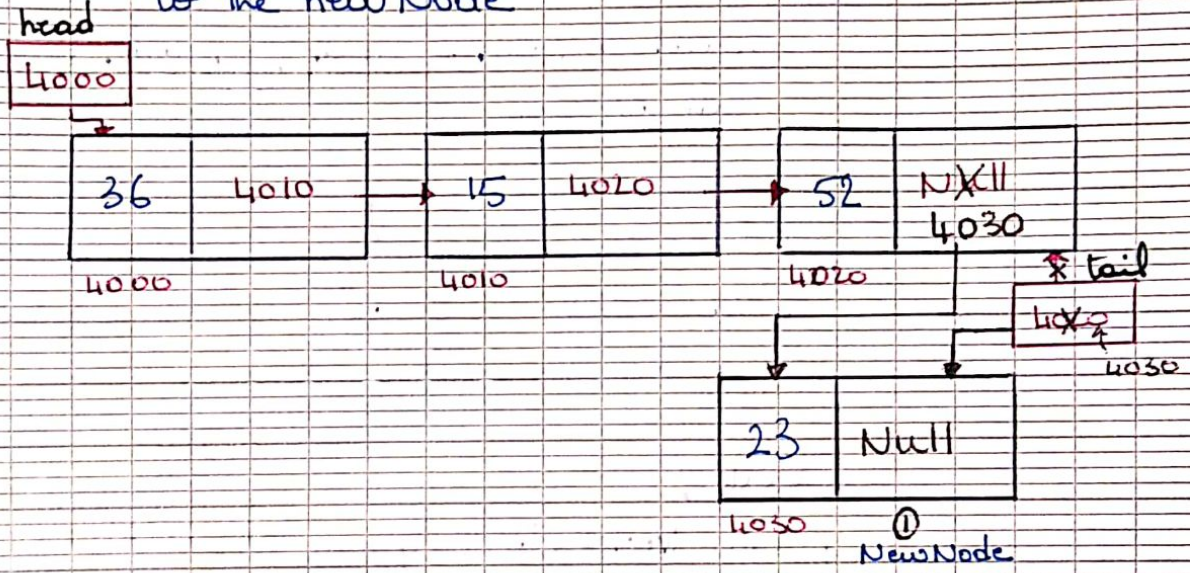
```
public class Queue {
  private Node head, tail;

// Constructor to initialize the queue
  public Queue () {
     head = tail = null;
  }
// Check if the queue is Empty
  public boolean isEmpty () {
    return (head == null);
  }
```

Operation enqueue ():

- Approach:

1. Create a new node with the given data
2. Check if the queue is empty
3. If the queue is empty, set both the head and tail pointers to the newNode
4. If the queue is not empty, set the 'next' pointer of the current 'tail' node to the new node, and update the 'tail' pointer to the newNode



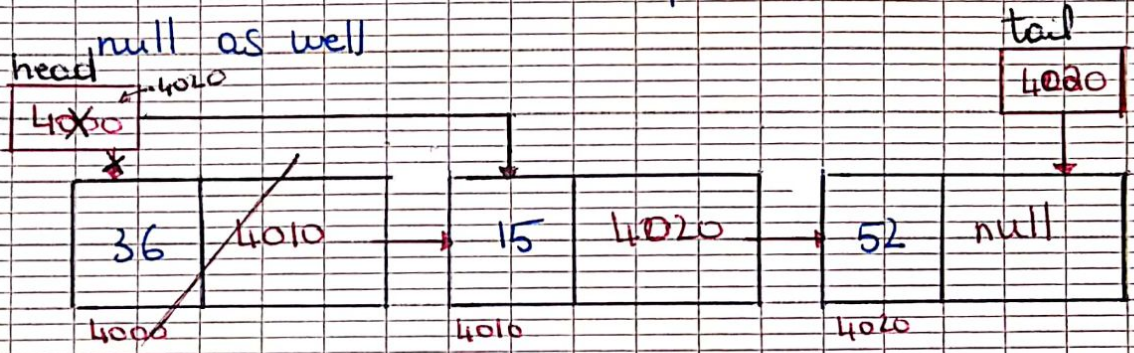
- Code

```
public void enqueue (NodeData item){
    Node newNode = new Node (item);
    if (isEmpty ()){
      head = newNode;
      tail = newNode;
    }
    else{
      tail.next = newNode;
      tail = newNode;
}  }
```

Operation dequeue()

- Approach:

1. Check if the queue is empty. If the queue is empty, there's nothing to dequeue
2. If the queue is not empty, retrieve the data from the head node
3. Move the 'head' pointer to the next node in the queue
4. If after dequeuing the last element, the head becomes null, update the tail to null as well



- Code:

```
public NodeData dequeue(){
    if (isEmpty()){
        System.out.println("The queue is empty");
        System.exit(1);
    }
    NodeData nd = head.data;
    head = head.next;
    if (head == null) // if the queue become empty
        tail == null;
} return nd;
```

↳ head() / tail()

```
// Return the element at the head without
   removing it
   public NodeData head(){
      if(isEmpty()){
         System.out.println("The queue is empty")
         System.exist(1);
      }
      return head.data;
}

// Return the element at the tail without removing it
   public NodeData Tail(){
      if (isEmpty()){
         System.out.println("The queue is empty")
         System.exist(1);
      }
      return tail.data;
   }
```

↳ display()

```
// Display the content of the queue
   public void display(){
    Node current = head;
    while ( current != null) {
      System.out.println (current.data.value + " ");
     current = current.next;
    }
}
```