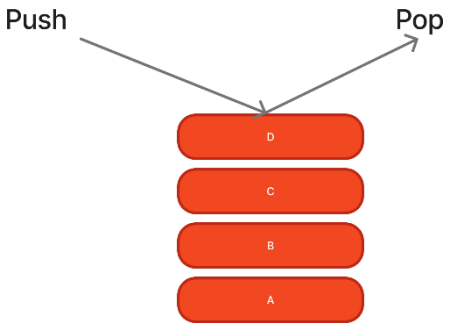
Lecture 04

Outline

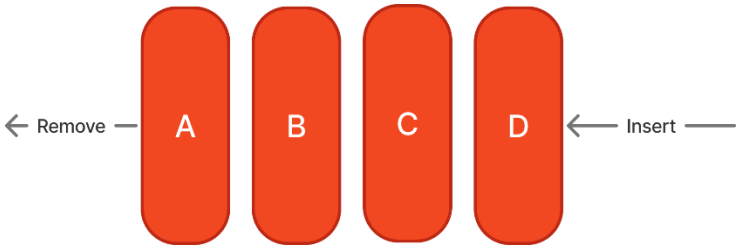
1. Queues – Introduction
    a. Queues in real life
2. Characteristics of queues
    a. FIFO
    b. Insertion and deletion are restricted from the middle
3. Queue Attributes
    a. Front
    b. Rear
4. Queue Operations
    a. Insert (Enqueue)
    b. Remove (Dequeue)
    c. Peek Front
5. Uses of queues
    a. Printer queue
    b. Stores keystroke data as you type at the keyboard
    c. Pipeline
6. Linear Queue Implementation
    a. Queue Array Implementation
    b. Insert Method implementation
    c. Remove Method Implementation
    d. Peek Front Method Implementation
7. Problems with Linear Queues
8. Circular Queue Implementation
    a. Queue Array Implementation
    b. Insert Method implementation
    c. Remove Method Implementation
    d. Peek Front Method Implementation

Queues – Introduction

- Queues in DSA are very similar to the queues in front of a counter.
- The first person who came to the counter get served first.
- Like that, the first element inserted into the queue get removed first.
    - Unlike stacks, you have two different points (front and rear) desperately for insertion and deletion.
    - Like stacks, you can only read one element per time.



Stacks



Queues

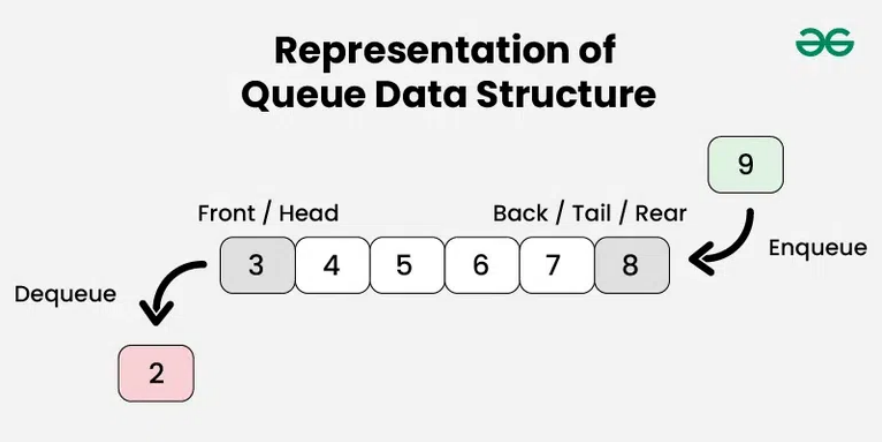
- Stacks are vertical data structures and queues are horizontal data structures.

Stacks

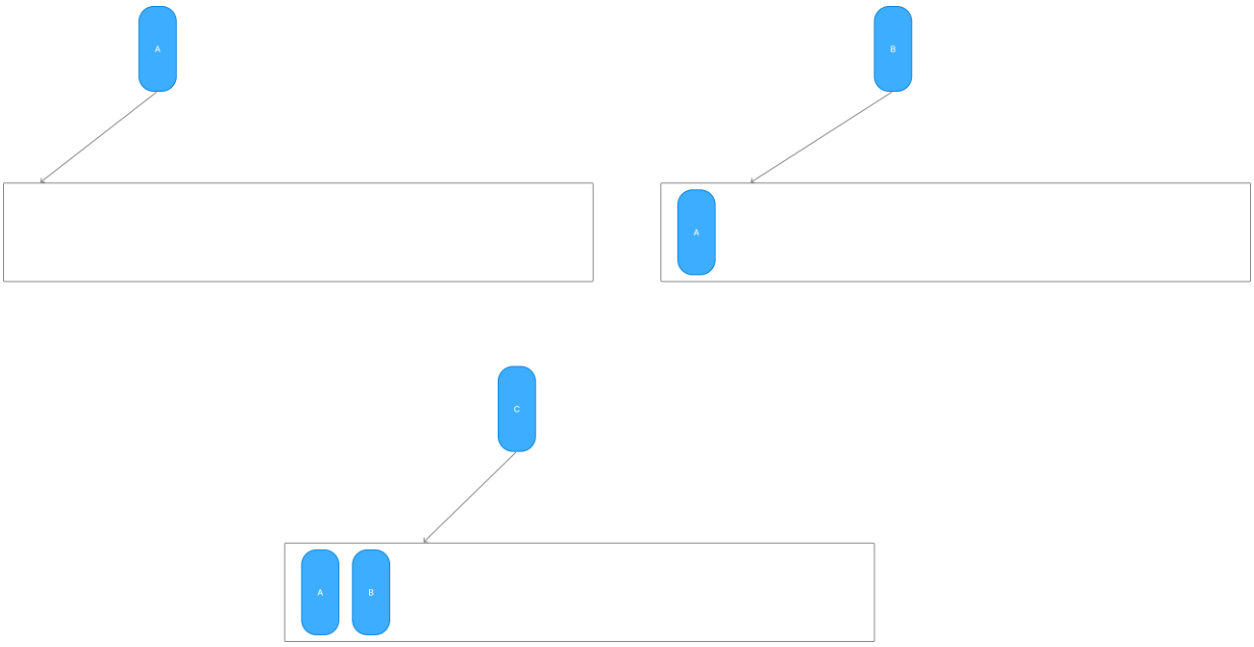
Queues

Characteristics of Queues

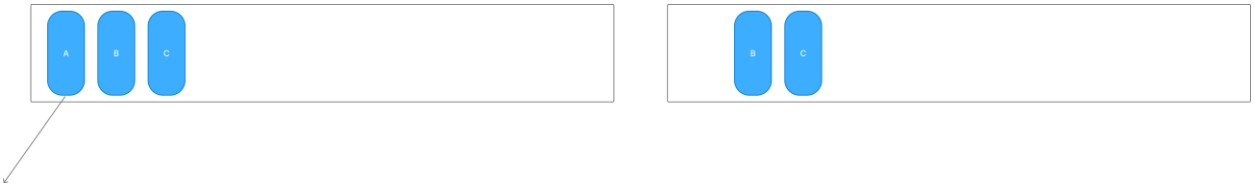
- As we discussed there are two points for a queue. In the following image you can see the head (Front) and the tail (Rear).



- The enqueue (insertion) is only possible from the tail and dequeue (removal) is only possible from the head.
- The following diagram shows the insertion operation.



- And the remove operation is represented by the following diagram.



- As you can see, the queue follows the First In First Out (**FIFO**) principle.
- The access from the middle is restricted.

Queue Attributes

- The queue has two main attributes. The insertion point (rear) and the deletion point (front).
- We access data from the insertion point (rear).
- We insert the data from the deletion point (front).

Queue Operations

- Insertion happens at the rear of the queue.
    - We must only insert if the queue is not full.
    - The insertion point must be updated after the insertion.
- Deletion happens at the front of the queue.
    - We must only try to delete a value if the queue is not empty.
    - The deletion point must be updated after the remove operation.
- Peek happens at the front of the queue
    - We must only try to read a value if the queue is not empty.
    - We should not modify the insertion point or deletion point after the peek operation.

Uses of Queues

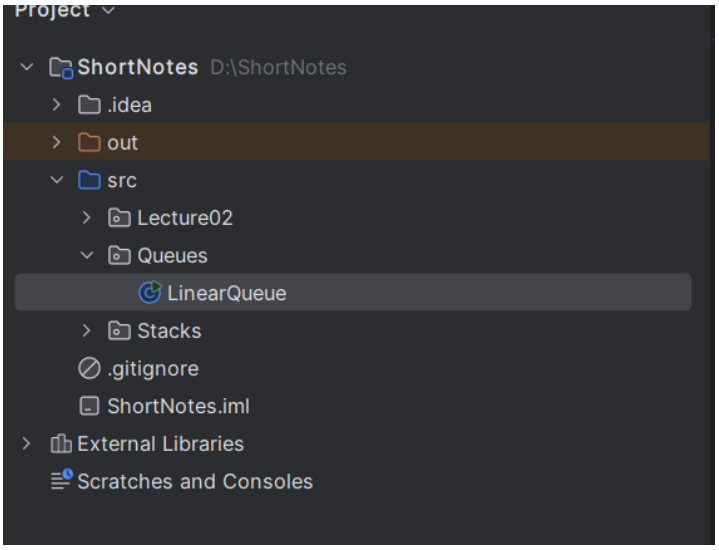
- Printer queue
  The printer must finish the first printing job, in order to return to the next job.

- Stores keystroke data as you type at the keyboard.
  When you type in keyboard it should follow the FIFO principle. Otherwise, you'll get reversed words or random words.

Linear Queue – Implementation

Queue Array Implementation

- First, we need to identify what are the attributes that we need to implement the linear queue.
  - An array to store values
  - The max capacity of the array.
  - Rear index for insertion
  - Front index for access (remove) values.
- First let's define these values.
- We'll create a java class called **LinearStack** to implement our first queue.



- Now let's define the attributes we need to implement the linear queue.
  - We shall define array size as an integer.
  - We'll set the rear value to 0, as the firstly inserted value goes into that index.
  - Also, the front index value is set to 0. Because in the first-to-remove value is in the front.

  - A variable to store the number of variables. We'll initialize it as 0.
  - In the constructor we're going to pass the array size as an argument.
- The code shall look like this.

```
package Queues;

public class LinearQueue {
    private int[] queue;
    private int maxCap;
    private int rear;
    private int front;
    private int nItems;

    public LinearQueue(int maxCap){
        this.maxCap = maxCap;
        queue = new int[maxCap];
        rear = 0;
        front = 0;
        nItems = 0;
    }
```

Insert Method Implementation

- Before inserting a value, we must check whether the list is full or not.
- If it's full we cannot insert a value into it.
- So, let's first create a method called **isFull()** to check whether the list is full or not.

Assume you have a queue of size 3.

Step	Value of Rear		What happens?	Queue
	Before the operation	After the operation		
Insert 10	0	1	queue[0] = 10 rear = 0 + 1	10
Insert 20	1	2	queue[1] = 20 rear = 1 + 1	10 20
Insert 30	2	3	queue[2] = 30 rear = 2 + 1	10 20 30

- As you can see, after filling in the last available index, rear value becomes the length of the list.
- So, we can simply implement isFull() method by

    - `rear == maxCap`

- The code shall look like this for is empty operation.

```
public boolean isFull(){
    return maxCap == rear;
}
```

- What should happen after calling the insert function?
    - First, ensure the queue is not full.
    - After that assign the value to-be-inserted to the rear index.
    - Do a postfix increment to the rear variable.
    - Also increment the item-count
- The code should look like this.

```
public void insert(int item){
    if (isFull()){
        System.out.println("The queue is full!");
    }
    else{
        queue[rear++] = item;
        nItems++;
    }
}
```

Remove Method Implementation

- To remove an item for list first we need to validate that the queue is not empty.
- To do that we'll implement a method called isEmpty().
- We can simply do that using,
    - `return nItems == 0`
- As you can see only when a value gets added to the queue, nItems get incremented.
- Let's first implement the isEmpty() function.

```
public boolean isEmpty(){
    return nItems == 0;
}
```

- After that we should follow these actions.
    - First, validate whether the stack is not empty.
    - Only if it's not empty, reduce the item count (nItems) by one.
    - After that return the front index value.
    - Increment front index value by one.

- The code should look like this.

```
public int remove(){
    if (isEmpty()){
        System.out.println("The queue is empty!");
        return -1;
    }
    else {
        nItems --;
        return queue[front++];
    }
}
```

Peek Method Implementation

- The peek method is almost the same as the remove method. The only difference is it does not modify our queue.
- We'll implement the peek method just by removing the increment and decrement operations from the push.

```
public int peek(){
    if (isEmpty()){
        System.out.println("The queue is empty!");
        return -1;
    }
    else {
        return queue[front];
    }
}
```

Extra: Print and Main

To print the queue and test our class implementation we'll use the following code

```
public void display(){
    for (int i = front; i < rear; i++){
        System.out.print(queue[i]);
        System.out.print(" ");
    }
    System.out.println();
}
```

```
public static void main(String[] args) {
    LinearQueue queue1 = new LinearQueue(5);

    queue1.insert(10);
    queue1.insert(20);
    queue1.insert(30);
    queue1.insert(40);
    queue1.insert(50);
    queue1.display();
    System.out.println();

    queue1.remove();
    queue1.remove();
    queue1.display();
    System.out.println();

    queue1.insert(60);
    queue1.display();
    System.out.println();

    queue1.remove();
    queue1.display();
    System.out.println();


}
```

supportbyDV

Whole Code

```
package Queues;

public class LinearQueue {
    private int[] queue;
    private int maxCap;
    private int rear;
    private int front;
    private int nItems;

    public LinearQueue(int maxCap){
        this.maxCap = maxCap;
        queue = new int[maxCap];
        rear = 0;
        front = 0;
        nItems = 0;
    }

    public boolean isEmpty(){
        return nItems == 0;
    }

    public boolean isFull(){
        return maxCap == rear;
    }

    public void insert(int item){
        if (isFull()){
            System.out.println("The queue is full!");
        }
        else{
            queue[rear++] = item;
            nItems++;
        }
    }

    public int remove(){
        if (isEmpty()){
            System.out.println("The queue is empty!");
            return -1;
        }
        else {
            nItems --;
            return queue[front++];
        }
    }

    public int peek(){
        if (isEmpty()){
            System.out.println("The queue is empty!");
            return -1;
        }
        else {
            return queue[front];
        }
    }
```

```
    public void display(){
        for (int i = front; i < rear; i++){
            System.out.print(queue[i]);
            System.out.print(" ");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        LinearQueue queue1 = new LinearQueue(5);

        queue1.insert(10);
        queue1.insert(20);
        queue1.insert(30);
        queue1.insert(40);
        queue1.insert(50);
        queue1.display();
        System.out.println();

        queue1.remove();
        queue1.remove();
        queue1.display();
        System.out.println();

        queue1.insert(60);
        queue1.display();
        System.out.println();

        queue1.remove();
        queue1.display();
        System.out.println();


    }

}
```

supportbyDV

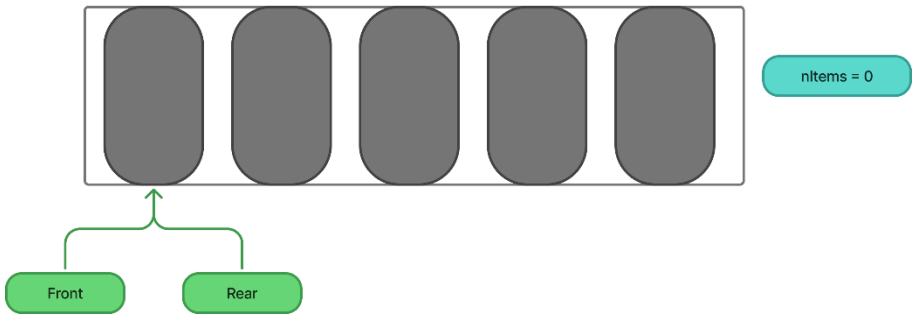
Output

```
10 20 30 40 50

30 40 50

The queue is full!
30 40 50

40 50


Process finished with exit code 0
```

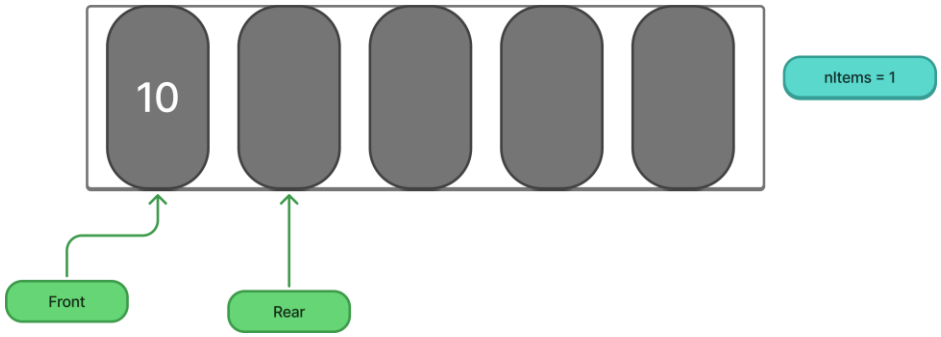
Problems with Linear Queues

As you can see there are some problems with linear queues. Let's revisit our last code.

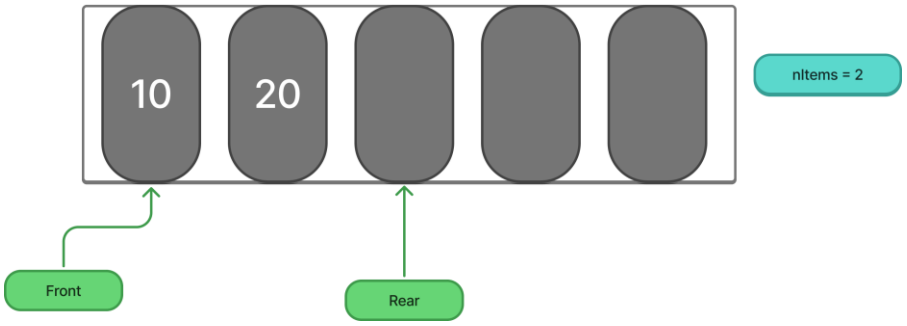
```
LinearQueue queue1 = new LinearQueue(5);
```



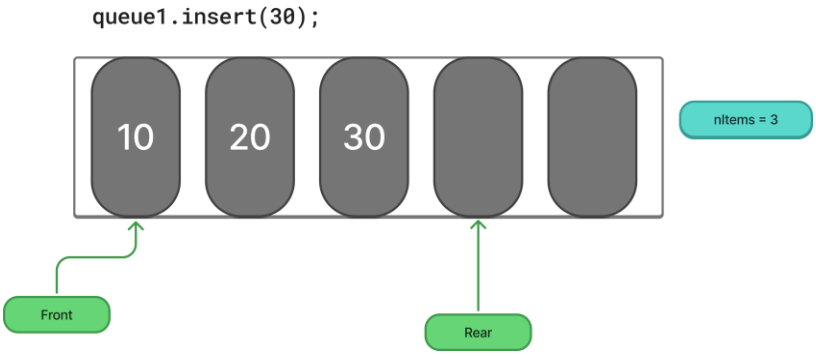

```
queue1.insert(10);
```

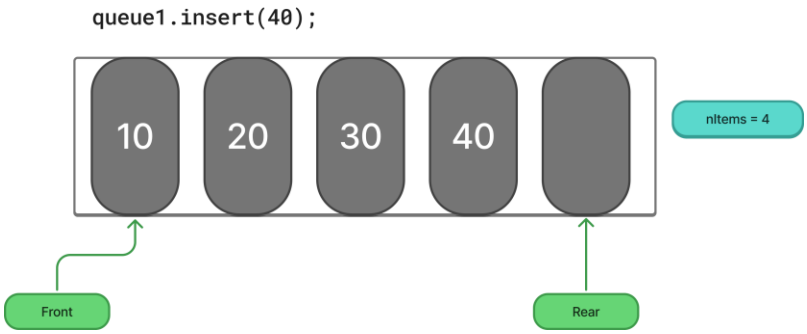


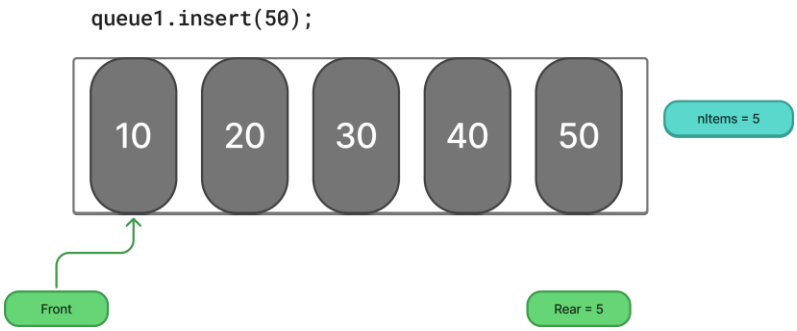

```
queue1.insert(20);
```

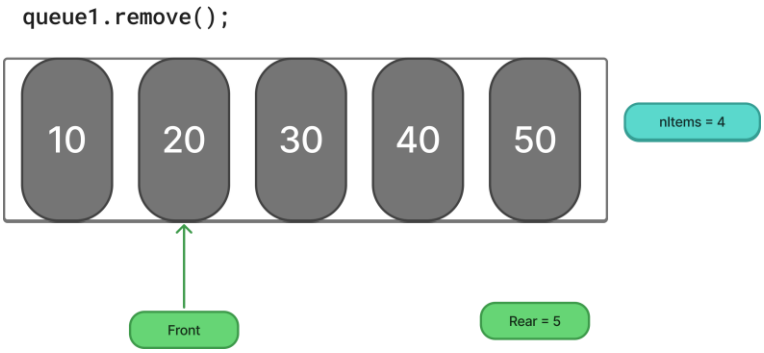


supportbyDV

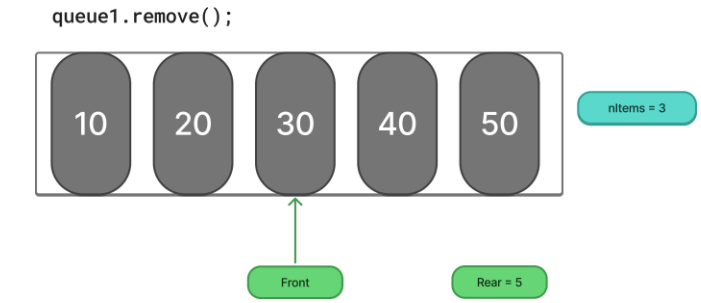


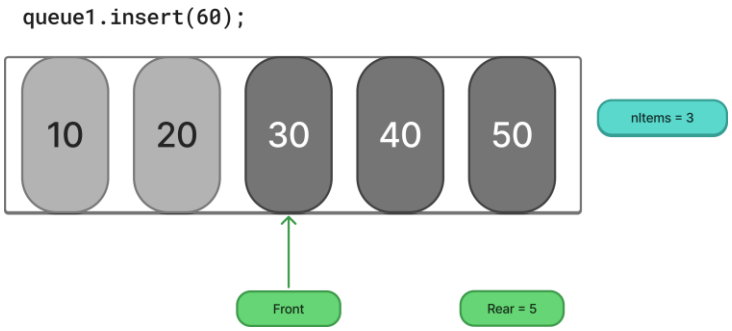






Prepared By Dinindu Vishwajith



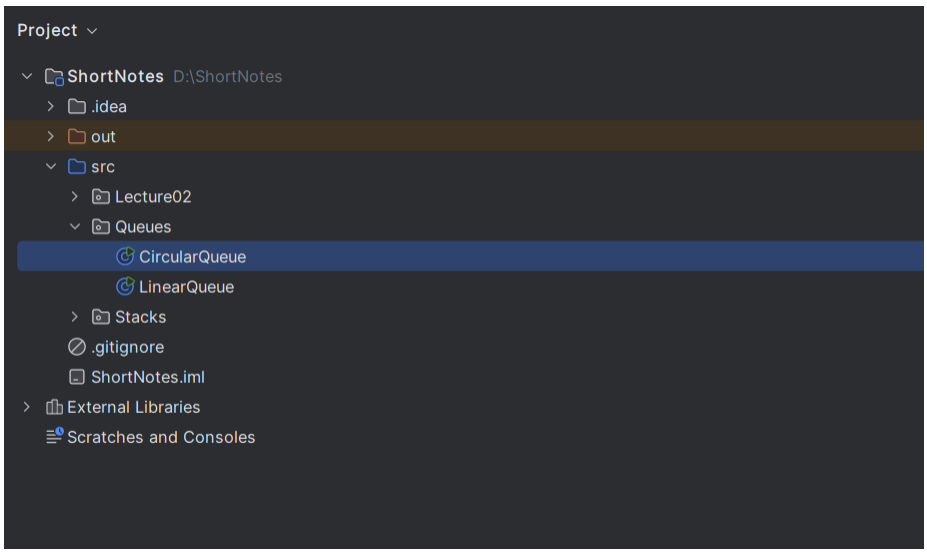


- We cannot add 60 to the queue even though we have two empty spaces. It's because our insertion point became 5. To solve this problem, we can use circular queues.

Circular Queue – Implementation

- There is no significant difference between linear and circular queue attributes implementation.
- However, when it comes to insertion and deletion there are some differences.
- First, let's look at the basic queue array implementation.

We'll create a java class called **CircularQueue**.



- As we implemented LinearQueue attributes, let's initialize the CircularQueue attributes as well.

```
package Queues;

public class CircularQueue {
    private int maxSize;
    private int[] queueArray;
    private int front;
    private int rear;
    private int nItems;

    public CircularQueue(int size) {
        maxSize = size;
        queueArray = new int[maxSize];
        front = 0;
        rear = -1;
        nItems = 0;
    }
```

Implementing Insert

- Regardless of what kind of queue we are using, we can not insert a value to the queue, if the queue is full.
- So, let's first implement the isFull() method.

```
public boolean isFull() {
    return nItems == maxSize;
}
```

- After validating it we have to,
    - After the validation we have to check whether the rear value is the last possible index or not.
        - rear == maxCap – 1
    - If it's the last possible index, we must reset it to -1.
        - if (rear == maxCap - 1){
          rear = -1;
          }
    - After that we'll do a postfix increment to the rear value and assign the item to queue.

```
public void insert(int item) {
    if (isFull()) {
        System.out.println("Queue is full");
        return;
    }
    if (rear == maxSize - 1) {
        rear = -1;
    }
    queueArray[++rear] = item;
    nItems++;
}
```

Implementing Remove

- Same as the linear queues, we cannot remove a value from an empty list. So, we'll first create the isEmpty() method to check whether the queue is empty or not.

```
public boolean isEmpty() {
    return nItems == 0;
}
```

- This is the validation.

```
public int remove() {
    if (isEmpty()) {
        System.out.println("Queue is empty");
        return -1;
    }
```

- We'll get the return value to a variable first.

```
int removedItem = queueArray[front];
```

- And same as the insert operation we reset the value to 0 when front reached the last possible index.
- This is what the remove method looks like.

```
public int remove() {
    if (isEmpty()) {
        System.out.println("Queue is empty");
        return -1;
    }
    int removedItem = queueArray[front];
    if (front == maxSize - 1) {
        front = 0;
    } else {
        front++;
    }
    nItems--;
    return removedItem;
}
```

- There is no difference in peek method.

```
public int peekFront() {
    if (isEmpty()) {
        System.out.println("Queue is empty");
        return -1;
    }
    return queueArray[front];
}
```

Extra – Print the Queue and Main Method

```
public void displayQueue() {
    if (isEmpty()) {
        System.out.println("Queue is empty");
        return;
    }
    System.out.print("Queue elements: ");
    int count = 0;
    int index = front;
    while (count < nItems) {
        System.out.print(queueArray[index] + " ");
        if (index == maxSize - 1) {
            index = 0;
        } else {
            index++;
        }
        count++;
    }
    System.out.println();
}
```

```
public static void main(String[] args) {
    CircularQueue queue = new CircularQueue(5);
    queue.insert(10);
    queue.insert(20);
    queue.insert(30);
    queue.insert(40);
    queue.insert(50);
    queue.displayQueue();
    System.out.println("Front element: " + queue.peekFront());
    queue.remove();
    queue.remove();
    queue.displayQueue();
    queue.insert(60);
    queue.insert(70);
    queue.displayQueue();
}
```

supportbyDV

Whole Code

```
package Queues;

public class CircularQueue {
    private int maxSize;
    private int[] queueArray;
    private int front;
    private int rear;
    private int nItems;

    public CircularQueue(int size) {
        maxSize = size;
        queueArray = new int[maxSize];
        front = 0;
        rear = -1;
        nItems = 0;
    }

    public void insert(int item) {
        if (isFull()) {
            System.out.println("Queue is full");
            return;
        }
        if (rear == maxSize - 1) {
            rear = -1;
        }
        queueArray[++rear] = item;
        nItems++;
    }

    public int remove() {
        if (isEmpty()) {
            System.out.println("Queue is empty");
            return -1;
        }
        int removedItem = queueArray[front];
        if (front == maxSize - 1) {
            front = 0;
        } else {
            front++;
        }
        nItems--;
        return removedItem;
    }

    public int peekFront() {
        if (isEmpty()) {
            System.out.println("Queue is empty");
            return -1;
        }
        return queueArray[front];
    }

    public boolean isEmpty() {
        return nItems == 0;
    }
```

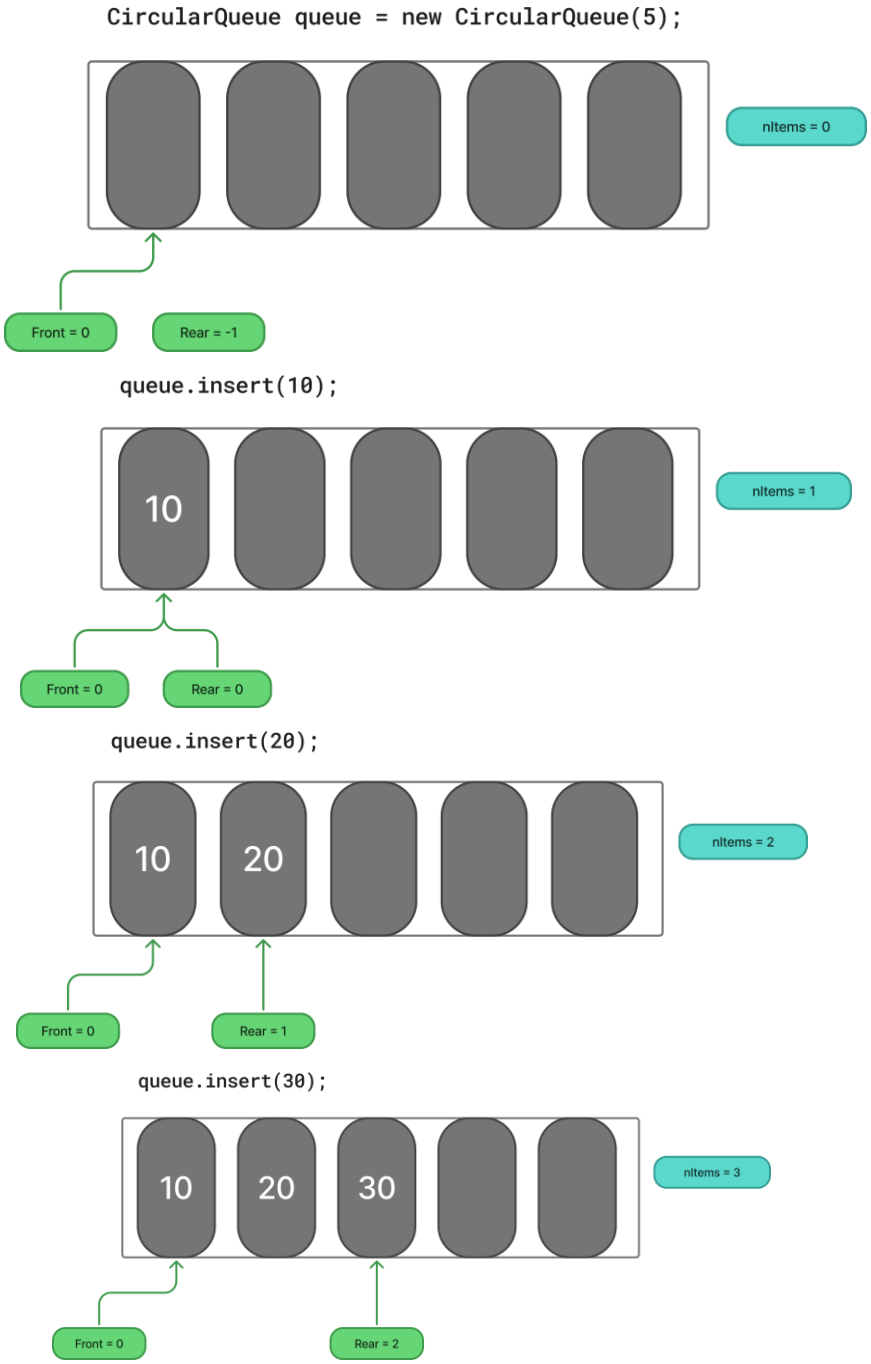
```
    public boolean isFull() {
        return nItems == maxSize;
    }

    public void displayQueue() {
        if (isEmpty()) {
            System.out.println("Queue is empty");
            return;
        }
        System.out.print("Queue elements: ");
        int count = 0;
        int index = front;
        while (count < nItems) {
            System.out.print(queueArray[index] + " ");
            if (index == maxSize - 1) {
                index = 0;
            } else {
                index++;
            }
            count++;
        }
        System.out.println();
    }

    public static void main(String[] args) {
        CircularQueue queue = new CircularQueue(5);
        queue.insert(10);
        queue.insert(20);
        queue.insert(30);
        queue.insert(40);
        queue.insert(50);
        queue.displayQueue();
        System.out.println("Front element: " + queue.peekFront());
        queue.remove();
        queue.remove();
        queue.displayQueue();
        queue.insert(60);
        queue.insert(70);
        queue.displayQueue();
    }
}
```

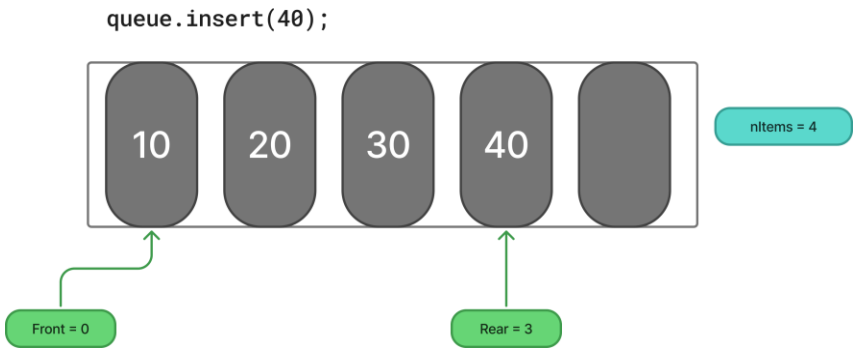
supportbyDV

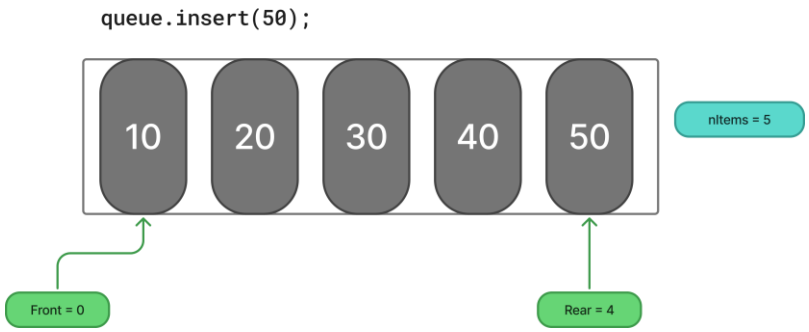
Output

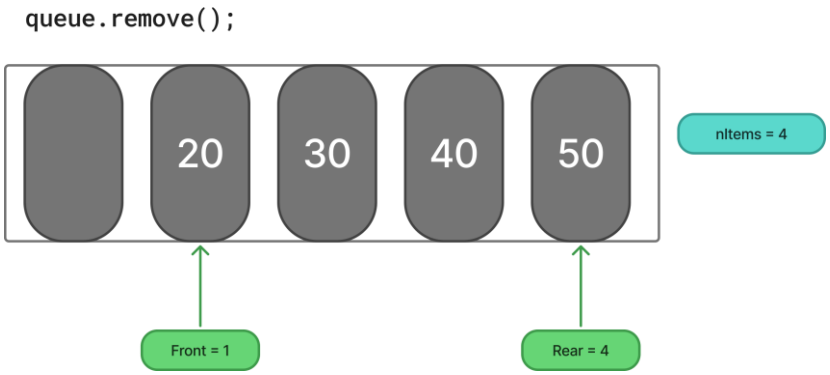
```
Queue elements: 10 20 30 40 50
Front element: 10
Queue elements: 30 40 50
Queue elements: 30 40 50 60 70

Process finished with exit code 0
```

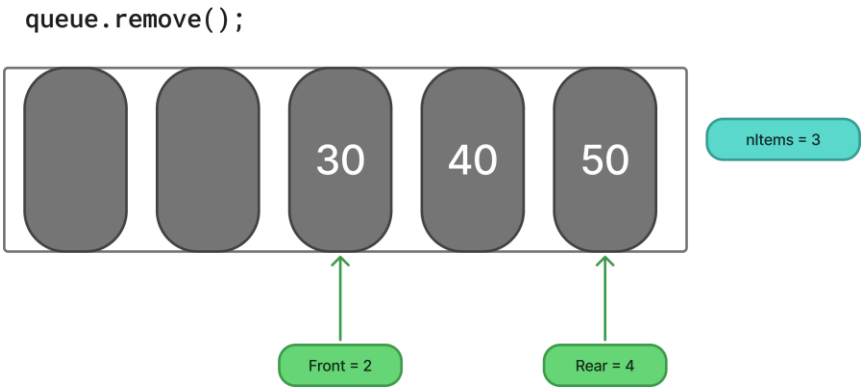
- Let's trace this



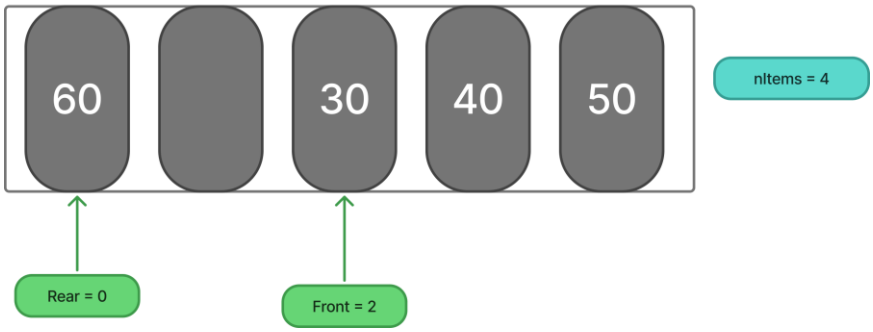

queue.insert(40);
10
20
30
40
nItems = 4
Front = 0
Rear = 3


queue.insert(50);
10
20
30
40
50
nItems = 5
Front = 0
Rear = 4


queue.remove();
20
30
40
50
nItems = 4
Front = 1
Rear = 4


queue.remove();
30
40
50
nItems = 3
Front = 2
Rear = 4

```
queue.insert(60);
```




```
queue.insert(70);
```