Practical Number-7: Implementation of Queue Using Array in Java

Aim:

To study the concept of **Queue data structure** and implement it using **Array in Java** to perform basic operations like Enqueue, Dequeue, Peek and Display.

Theory:

A **Queue** is a linear data structure that follows the principle of:

FIFO (First In First Out)

This means the element inserted first is removed first.

Real Life Examples:

- People standing in a queue at a ticket counter
- Printer job scheduling
- CPU process scheduling

Types of Queue:

1. Linear Queue. 2. Circular Queue

3. Priority Queue. 4.Double Ended Queue (Deque)

In this experiment, we are implementing a **Linear Queue using Array**.

Queue Operations:

Operation	Description
Enqueue	Insert element at rear
Dequeue	Remove element from front
Peek	Display front element
isEmpty	Check if queue is empty
isFull	Check if queue is full

Representation Using Array:

In array implementation:

- front → Points to first element
- rear → Points to last element
- Initially: front = -1, rear = -1

Overflow Condition:

When:

rear == maxSize - 1

Underflow Condition:

When:

front == -1 OR front > rear

Advantages of Array Implementation:

- Simple to implement
- Fast access (O(1) time complexity)
- Memory allocated at once

Limitations:

- Fixed size (cannot grow dynamically)
- Wastage of space in linear queue
- Cannot reuse deleted space

Algorithm:

Enqueue

1. Check if queue is full.
2. If full → Display overflow.
3. If front == -1 → Set front = 0.
4. Increment rear.
5. Insert element at rear.

Dequeue

1. Check if queue is empty.
2. If empty → Display underflow.
3. Print element at front.
4. Increment front.

Display

1. If queue empty → Print message.
2. Else print elements from front to rear.

Code(Java):

```
import java.util.Scanner;
class QueueArray {
    int maxSize;
    int front, rear;
    int[] queue;
QueueArray(int size) {
        maxSize = size;
        queue = new int[maxSize];
        front = -1;
        rear = -1;
    }
boolean isFull() {
        return (rear == maxSize - 1);
    }
boolean isEmpty() {
        return (front == -1 || front > rear);
    }
 void enqueue(int value) {
        if (isFull()) {
            System.out.println("Queue Overflow!");
        } else {
            if (front == -1) {
                front = 0;
            }
            rear++;
            queue[rear] = value;
```

```
            System.out.println(value + " inserted into queue.");
        }
    }
void dequeue() {
        if (isEmpty()) {
            System.out.println("Queue Underflow!");
        } else {
            System.out.println(queue[front] + " removed from queue.");
            front++;
        }
    }
 void peek() {
        if (isEmpty()) {
            System.out.println("Queue is Empty!");
        } else {
            System.out.println("Front element: " + queue[front]);
        }
    }
void display() {
        if (isEmpty()) {
            System.out.println("Queue is Empty!");
        } else {
            System.out.print("Queue elements: ");
            for (int i = front; i <= rear; i++) {
                System.out.print(queue[i] + " ");
            }
            System.out.println();
        }
    }
```

```
}
public class QueueImplementation {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
System.out.print("Enter size of queue: ");
        int size = sc.nextInt();
QueueArray q = new QueueArray(size);
        int choice, value;
do {
            System.out.println("\n1. Enqueue");
            System.out.println("2. Dequeue");
            System.out.println("3. Peek");
            System.out.println("4. Display");
            System.out.println("5. Exit");
            System.out.print("Enter choice: ");
            choice = sc.nextInt();
switch (choice) {
                case 1:
                    System.out.print("Enter value: ");
                    value = sc.nextInt();
                    q.enqueue(value);
                    break;
                case 2:
                    q.dequeue();
                    break;
                case 3:
                    q.peek();
                    break;
                case 4:
```

```
                q.display();
                break;
            case 5:
                System.out.println("Program terminated.");
                break;
            default:
                System.out.println("Invalid choice!");
        }

    } while (choice != 5)
sc.close();
  }
}
```

Time Complexity:

- Enqueue → O(1)
- Dequeue → O(1)
- Peek → O(1)
- Display → O(n)

Conclusion:

The Queue data structure was successfully implemented using an array in Java.
The program demonstrates FIFO behavior and handles overflow and underflow conditions properly.