1. **Reverse the First K Elements of a Queue.**

Code:-

```
import java.util.*;

public class ReverseFirstKElements {
    public static void reverseFirstKElements(Queue<Integer> queue, int k) {
        if (queue == null || queue.size() < k || k <= 0) {
            System.out.println("Invalid input.");
            return;
        }

        Stack<Integer> stack = new Stack<>();

        // Step 1: Push the first K elements into the stack
        for (int i = 0; i < k; i++) {
            stack.push(queue.poll());
        }

        // Step 2: Pop elements from the stack and enqueue them back
        while (!stack.isEmpty()) {
            queue.add(stack.pop());
        }

        // Step 3: Move the remaining elements to the back of the queue
        int size = queue.size();
        for (int i = 0; i < size - k; i++) {
            queue.add(queue.poll());
        }
    }

    public static void main(String[] args) {
        Queue<Integer> queue = new LinkedList<>();
        queue.add(1);
        queue.add(2);
        queue.add(3);
        queue.add(4);
        queue.add(5);

        int k = 3;

        System.out.println("Original Queue: " + queue);
        reverseFirstKElements(queue, k);
        System.out.println("Modified Queue: " + queue);
    }
}
```

2. **Implement a Circular Queue.**

Code:-

```
class CircularQueue {
    private int[] queue;
    private int front;
    private int rear;
    private int size;
    private int capacity;

    // Constructor to initialize the circular queue
    public CircularQueue(int capacity) {
        this.capacity = capacity;
        this.queue = new int[capacity];
        this.front = -1;
        this.rear = -1;
        this.size = 0;
    }

    // Method to add an element to the queue
    public boolean enqueue(int value) {
        if (isFull()) {
            System.out.println("Queue is full! Overflow condition.");
            return false;
        }
        if (isEmpty()) {
            front = 0;
        }
        rear = (rear + 1) % capacity;
        queue[rear] = value;
        size++;
        return true;
    }

    // Method to remove an element from the queue
    public int dequeue() {
        if (isEmpty()) {
            System.out.println("Queue is empty! Underflow condition.");
            return -1; // Indicating error
        }
        int removedValue = queue[front];
        if (front == rear) { // Only one element was in the queue
            front = -1;
            rear = -1;
        } else {
            front = (front + 1) % capacity;
        }
        size--;
        return removedValue;
```

```
    }

    // Method to get the front element of the queue
    public int peek() {
        if (isEmpty()) {
            System.out.println("Queue is empty! No elements to peek.");
            return -1; // Indicating error
        }
        return queue[front];
    }

    // Method to check if the queue is empty
    public boolean isEmpty() {
        return size == 0;
    }

    // Method to check if the queue is full
    public boolean isFull() {
        return size == capacity;
    }

    // Method to get the size of the queue
    public int getSize() {
        return size;
    }

    // Main method to test the CircularQueue class
    public static void main(String[] args) {
        CircularQueue circularQueue = new CircularQueue(5);

        // Testing enqueue operation
        circularQueue.enqueue(10);
        circularQueue.enqueue(20);
        circularQueue.enqueue(30);
        circularQueue.enqueue(40);
        circularQueue.enqueue(50);
        System.out.println("Enqueue result (should fail): " + circularQueue.enqueue(60)); //
Overflow

        // Testing peek operation
        System.out.println("Front element: " + circularQueue.peek()); // Should print 10

        // Testing dequeue operation
        System.out.println("Dequeued element: " + circularQueue.dequeue()); // Should print
10
        System.out.println("Dequeued element: " + circularQueue.dequeue()); // Should print
20
```

```
        // Adding more elements
        circularQueue.enqueue(60);
        circularQueue.enqueue(70);

        // Testing circular behavior
        while (!circularQueue.isEmpty()) {
            System.out.println("Dequeued element: " + circularQueue.dequeue());
        }

        // Underflow test
        System.out.println("Dequeue result (should fail): " + circularQueue.dequeue()); //
Underflow
    }
}
```

3. Find the First Negative Integer in Every Window of Size KKK in a Queue.

Code:-

```
import java.util.*;

public class FirstNegativeInWindow {
    public static void main(String[] args) {
        int[] arr = {12, -1, -7, 8, 15, 30, 16, 28};
        int k = 3;
        List<Integer> result = findFirstNegativeInWindow(arr, k);
        System.out.println(result);
    }

    public static List<Integer> findFirstNegativeInWindow(int[] arr, int k)
{
        List<Integer> result = new ArrayList<>();
        Queue<Integer> negatives = new LinkedList<>(); // Queue to store
indices of negative numbers

        for (int i = 0; i < arr.length; i++) {
            // Add current element index to the queue if it is negative
            if (arr[i] < 0) {
                negatives.add(i);
            }
```

```
            // Remove elements that are out of the current window
            if (!negatives.isEmpty() && negatives.peek() < i - k + 1) {
                negatives.poll();
            }

            // Add the first negative number in the current window to the
result
            if (i >= k - 1) {
                if (!negatives.isEmpty()) {
                    result.add(arr[negatives.peek()]);
                } else {
                    result.add(0);
                }
            }
        }

        return result;
    }
}
```

4. **Interleave the First Half and Second Half of a Queue.**

Code:-

```
import java.util.LinkedList;
import java.util.Queue;

public class InterleaveQueue {

    public static void interleaveQueue(Queue<Integer> queue) {
        if (queue.size() % 2 != 0) {
            throw new IllegalArgumentException("Queue size must be
even");
        }

        int halfSize = queue.size() / 2;
        Queue<Integer> firstHalf = new LinkedList<>();

        // Move the first half of the queue to a new queue
```

```
        for (int i = 0; i < halfSize; i++) {
            firstHalf.add(queue.poll());
        }

        // Interleave the two halves
        while (!firstHalf.isEmpty()) {
            queue.add(firstHalf.poll()); // Add from the first half
            queue.add(queue.poll());    // Add from the original second half
        }
    }

    public static void main(String[] args) {
        Queue<Integer> queue = new LinkedList<>();
        queue.add(1);
        queue.add(2);
        queue.add(3);
        queue.add(4);
        queue.add(5);
        queue.add(6);

        System.out.println("Original Queue: " + queue);
        interleaveQueue(queue);
        System.out.println("Interleaved Queue: " + queue);
    }
}
```

5. LRU Cache Implementation Using a Queue.

Code:-

```
import java.util.*;

public class LRUCache {
    private final int capacity;
    private final Map<Integer, Integer> map; // To store key-value pairs
    private final LinkedList<Integer> queue; // To maintain the order of usage

    public LRUCache(int capacity) {
        this.capacity = capacity;
        this.map = new HashMap<>();
        this.queue = new LinkedList<>();
    }
```

```
    public int get(int key) {
        if (!map.containsKey(key)) {
            return -1; // Key not present
        }

        // Move the accessed key to the front of the queue
        queue.remove((Integer) key);
        queue.addFirst(key);

        return map.get(key);
    }

    public void put(int key, int value) {
        if (map.containsKey(key)) {
            // Key already exists, update its value and move to the front
            map.put(key, value);
            queue.remove((Integer) key);
            queue.addFirst(key);
        } else {
            if (map.size() >= capacity) {
                // Remove the least recently used key
                int lruKey = queue.removeLast();
                map.remove(lruKey);
            }
            // Add the new key-value pair
            map.put(key, value);
            queue.addFirst(key);
        }
    }

    public static void main(String[] args) {
        LRUCache lruCache = new LRUCache(3);

        lruCache.put(1, 100);
        lruCache.put(2, 200);
        lruCache.put(3, 300);

        System.out.println(lruCache.get(1)); // Outputs 100

        lruCache.put(4, 400); // Evicts key 2

        System.out.println(lruCache.get(2)); // Outputs -1 (not found)
        System.out.println(lruCache.get(3)); // Outputs 300
        System.out.println(lruCache.get(4)); // Outputs 400
    }
}
```

6. Generate Binary Numbers from 1 to NNN Using a Queue.

Code:-

```
import java.util.LinkedList;
import java.util.Queue;

public class BinaryNumberGenerator {

  // Method to generate binary numbers from 1 to N
  public static void generateBinaryNumbers(int N) {
    if (N <= 0) {
      System.out.println("Invalid input: N should be greater than 0.");
      return;
    }

    Queue<String> queue = new LinkedList<>();
    queue.add("1"); // Start with the first binary number

    for (int i = 1; i <= N; i++) {
      String current = queue.poll(); // Get the front element
      System.out.println(current); // Print the binary number

      // Generate the next two binary numbers and add to the queue
      queue.add(current + "0");
      queue.add(current + "1");
    }
  }

  // Main method to test the BinaryNumberGenerator class
  public static void main(String[] args) {
    int N = 10; // Change this value to generate binary numbers up to N
    System.out.println("Binary numbers from 1 to " + N + ":");
    generateBinaryNumbers(N);
  }
}
```

7.Shortest Path in a Binary Maze Using BFS and a Queue.

Code:-

```
import java.util.*;

public class ShortestPathBinaryMaze {
  // Directions for moving up, down, left, right
  private static final int[][] DIRECTIONS = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

  public static void main(String[] args) {
    int[][] maze = {
```

```java
            {1, 0, 0, 0},
            {1, 1, 0, 1},
            {0, 1, 0, 0},
            {1, 1, 1, 1}
        };

        int[] start = {0, 0};
        int[] destination = {3, 3};

        int result = shortestPathInBinaryMaze(maze, start, destination);
        System.out.println("Shortest Path Length: " + result);
    }

    public static int shortestPathInBinaryMaze(int[][] maze, int[] start, int[] destination) {
        int rows = maze.length;
        int cols = maze[0].length;

        // Edge case: if the start or destination is not traversable
        if (maze[start[0]][start[1]] == 0 || maze[destination[0]][destination[1]] == 0) {
            return -1;
        }

        // Queue for BFS: stores the coordinates and current distance
        Queue<int[]> queue = new LinkedList<>();
        queue.add(new int[]{start[0], start[1], 0}); // {row, col, distance}

        // Visited array to track visited cells
        boolean[][] visited = new boolean[rows][cols];
        visited[start[0]][start[1]] = true;

        while (!queue.isEmpty()) {
            int[] current = queue.poll();
            int row = current[0];
            int col = current[1];
            int distance = current[2];

            // Check if the destination is reached
            if (row == destination[0] && col == destination[1]) {
                return distance;
            }

            // Explore all possible directions
            for (int[] direction : DIRECTIONS) {
                int newRow = row + direction[0];
                int newCol = col + direction[1];

                // Check if the new cell is within bounds and traversable
                if (newRow >= 0 && newRow < rows && newCol >= 0 && newCol < cols
```

```
                && maze[newRow][newCol] == 1 && !visited[newRow][newCol]) {
              queue.add(new int[]{newRow, newCol, distance + 1});
              visited[newRow][newCol] = true;
            }
          }
        }

        // Return -1 if no path exists
        return -1;
    }
}
```