1. Augment the ArrayQueue implementation with a new rotate( ) method having semantics identical to the combination, enqueue(dequeue( )). But, your implementation should be more efficient than making two separate calls (for example, because there is no need to modify the size).

public void rotate() {

if (isEmpty()) {

return; // No need to rotate an empty queue

}

E element = elements[front];

front = (front + 1) % elements.length;

elements[rear] = element;

rear = (rear + 1) % elements.length;

}

2.Implement the clone( ) method for the ArrayQueue class.

@SuppressWarnings("unchecked")

public ArrayQueue<E> clone() {

ArrayQueue<E> clonedQueue = new ArrayQueue<>(capacity);

clonedQueue.elements = (E[]) new Object[elements.length];

System.arraycopy(elements, 0, clonedQueue.elements, 0, elements.length);

clonedQueue.front = front;

clonedQueue.rear = rear;

clonedQueue.size = size;

return clonedQueue;

}

3.Implement a method with signature concatenate(LinkedQueue Q2) for the LinkedQueue class that takes all elements of Q2 and appends them to the end of the original queue. The operation should run in O(1) time and should result in Q2 being an empty

queue.

public void concatenate(LinkedQueue<E> Q2) {

if (Q2.isEmpty()) {

return; // No need to concatenate an empty queue

}

if (isEmpty()) {

front = Q2.front;

} else {

rear.next = Q2.front;

}

rear = Q2.rear;

size += Q2.size;

Q2.front = null;

Q2.rear = null;

Q2.size = 0;

}

4.Use a queue to solve the Josephus Problem.

import java.util.LinkedList;

import java.util.Queue;

public class JosephusProblem {

public static int josephus(int n, int k) {

Queue<Integer> queue = new LinkedList<>();

// Enqueue all people from 1 to n

for (int i = 1; i <= n; i++) {

queue.add(i);

}

while (queue.size() > 1) {

// Skip k-1 people

for (int i = 0; i < k - 1; i++) {

int eliminated = queue.remove();

queue.add(eliminated);

}

// Eliminate the kth person

queue.remove();

}

// Return the last remaining person

return queue.remove();

}

public static void main(String[] args) {

int n = 7; // Number of people

int k = 3; // Elimination interval

int survivor = josephus(n, k);

System.out.println("The last person remaining is: " + survivor);

}

}

5.Use a queue to simulate Round Robin Scheduling.

import java.util.LinkedList;

import java.util.Queue;

public class RoundRobinScheduling {

public static void simulateRoundRobin(int[] processIDs, int[] burstTimes, int timeSlice) {

int n = processIDs.length;

Queue<Integer> queue = new LinkedList<>();

for (int i = 0; i < n; i++) {

queue.add(processIDs[i]);

}

int totalTime = 0;

while (!queue.isEmpty()) {

int currentProcess = queue.remove();

int remainingBurstTime = burstTimes[currentProcess];

if (remainingBurstTime <= timeSlice) {

// Process completes within the time slice

totalTime += remainingBurstTime;

System.out.println("Process " + currentProcess + " completed. Total time: " + totalTime);

} else {

// Process needs more time

totalTime += timeSlice;

System.out.println("Time slice expired for process " + currentProcess + ". Total time: " + totalTime);

burstTimes[currentProcess] -= timeSlice;

queue.add(currentProcess);

}

}

}

public static void main(String[] args) {

int[] processIDs = {1, 2, 3, 4, 5};

int[] burstTimes = {10, 4, 6, 8, 2};

int timeSlice = 3;

simulateRoundRobin(processIDs, burstTimes, timeSlice);

}

}

-----------------------------------------------------------------------------------------------------------------------