## Topics

1. Create Queue Interface

public interface Queue<E> {

boolean isEmpty();

int size();

void enqueue(E element);

E dequeue();

E peek();

}

1. Create Queue Using Array

public class ArrayQueue<E> implements Queue<E> {

private static final int DEFAULT\_CAPACITY = 10;

private E[] elements;

private int front;

private int rear;

private int size;

public ArrayQueue() {

this(DEFAULT\_CAPACITY);

}

@SuppressWarnings("unchecked")

public ArrayQueue(int capacity) {

elements = (E[]) new Object[capacity];

front = 0;

rear = -1;

size = 0;

}

@Override

public boolean isEmpty() {

return size == 0;

}

@Override

public int size() {

return size;

}

@Override

public E first() {

if (isEmpty()) {

throw new IllegalStateException("Queue is empty");

}

return elements[front];

}

@Override

public void enqueue(E element) {

if (size == elements.length) {

throw new IllegalStateException("Queue is

full");

}

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

elements[rear] = element;

size++;

}

@Override

public E dequeue() {

if (isEmpty()) {

throw new IllegalStateException("Queue is empty");

}

E removedElement = elements[front];

elements[front] = null;

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

size--;

return removedElement;

}

}

1. Create Queue Using Linked Lists

public class LinkedQueue<E> implements Queue<E> {

private Node<E> front;

private Node<E> rear;

private int size;

private static class Node<E> {

private E element;

private Node<E> next;

public Node(E element, Node<E> next) {

this.element = element;

this.next = next;

}

}

public LinkedQueue() {

front = null;

rear = null;

size = 0;

}

@Override

public boolean isEmpty() {

return size == 0;

}

@Override

public int size() {

return size;

}

@Override

public void enqueue(E element) {

Node<E> newNode = new Node<>(element, null);

if (isEmpty()) {

front = newNode;

} else {

rear.next = newNode;

}

rear = newNode;

size++;

}

@Override

public E dequeue() {

if (isEmpty()) {

throw new IllegalStateException("Queue is empty");

}

E removedElement =

front.element;

front = front.next;

if (front == null) {

rear = null;

}

size--;

return removedElement;

}

@Override

public E peek() {

if (isEmpty()) {

throw new IllegalStateException("Queue is empty");

}

return front.element;

}

}

1. Implement Basic Methods of Queue

* isEmpty()
* size()
* first()
* enqueue(E e)
* dequeue()

public interface Queue<E> {

boolean isEmpty();

int size();

E first();

void enqueue(E element);

E dequeue();

}

## Homework

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 class ArrayQueue<E> implements Queue<E> {

private E[] elements;

private int front;

private int rear;

private int size;

// constructor and other methods

public void rotate() {

if (size > 0) {

E element = dequeue();

enqueue(element);

}

}

}

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

public class ArrayQueue<E> implements Queue<E> {

// other class variables and methods

@Override

public ArrayQueue<E> clone() {

try {

@SuppressWarnings("unchecked")

ArrayQueue<E> clonedQueue = (ArrayQueue<E>) super.clone();

clonedQueue.elements = elements.clone();

return clonedQueue;

} catch (CloneNotSupportedException e) {

throw new AssertionError(); // Should not happen

}

}

}

1. 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 class LinkedQueue<E>

implements Queue<E> {

// other class variables and methods

public void concatenate(LinkedQueue<E> Q2) {

if (Q2.isEmpty()) {

return;

}

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;

}

}

1. 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) {

// Dequeue k-1 people and enqueue them back

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

int person = queue.remove();

queue.add(person);

}

// Dequeue the k-th person (eliminate)

queue. Remove();

}

1. Use a queue to simulate Round Robin Scheduling.

public class ArrayQueue<E> implements Queue<E> {

private static final int DEFAULT\_CAPACITY = 10;

private E[] elements;

private int front;

private int rear;

private int size;

public ArrayQueue() {

this(DEFAULT\_CAPACITY);

}

@SuppressWarnings("unchecked")

public ArrayQueue(int capacity) {

elements = (E[]) new Object[capacity];

front = 0;

rear = -1;

size = 0;

}

@Override

public boolean isEmpty() {

return size == 0;

}

@Override

public int size() {

return size;

}

@Override

public E first() {

if (isEmpty()) {

throw new IllegalStateException("Queue is empty");

}

return elements[front];

}

@Override

public void enqueue(E element) {

if (size == elements.length) {

throw new IllegalStateException("Queue is

full");

}

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

elements[rear] = element;

size++;

}

@Override

public E dequeue() {

if (isEmpty()) {

throw new IllegalStateException("Queue is empty");

}

E removedElement = elements[front];

elements[front] = null;

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

size--;

return removedElement;

}

}