1. Create Queue Interface
2. Create Queue Using Array
3. Create Queue Using Linked Lists
4. Implement Basic Methods of Queue

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

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).
2. Implement the clone( ) method for the ArrayQueue class.
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.
4. Use a queue to solve the Josephus Problem.
5. Use a queue to simulate Round Robin Scheduling.
6. إنشاء واجهة Queue:

public interface Queue<E> {

boolean isEmpty();

int size();

E first();

void enqueue(E element);

E dequeue();

}

يتم تعريف واجهة Queue بواجهات الطرق الأساسية لعمليات الإدخال والإخراج من العناصر في القائمة.

1. إنشاء قائمة باستخدام المصفوفة:

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

private static final int DEFAULT\_CAPACITY = 10;

private E[] data;

private int front;

private int size;

public ArrayQueue() {

this(DEFAULT\_CAPACITY);

}

public ArrayQueue(int capacity) {

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

front = 0;

size = 0;

}

public boolean isEmpty() {

return size == 0;

}

public int size() {

return size;

}

public E first() {

if (isEmpty()) {

return null;

}

return data[front];

}

public void enqueue(E element) {

if (size == data.length) {

resize(2 \* data.length);

}

int back = (front + size) % data.length;

data[back] = element;

size++;

}

public E dequeue() {

if (isEmpty()) {

return null;

}

E removedElement = data[front];

data[front] = null;

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

size--;

if (size > 0 && size == data.length / 4) {

resize(data.length / 2);

}

return removedElement;

}

private void resize(int capacity) {

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

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

newData[i] = data[(front + i) % data.length];

}

data = newData;

front = 0;

}

}

تم تنفيذ قائمة باستخدام المصفوفة. يتم استخدام المصفوفة لتخزين العناصر ومؤشر front لتتبع العنصر الأمامي في القائمة ومتغير size لتتبع حجم القائمة.

1. إنشاء قائمة باستخدام القوائم المرتبطة:

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

private SinglyLinkedList<E> list;

public LinkedQueue() {

list = new SinglyLinkedList<>();

}

public boolean isEmpty() {

return list.isEmpty();

}

public int size() {

return list.size();

}

public E first() {

return list.first();

}

public void enqueue(E element) {

list.addLast(element);

}

public E dequeue() {

return list.removeFirst();

}

}

تم تنفيذ قائمة باستخدام القوائم المرتبطة. يتم استخدام قائمة مرتبطة لتخزين العناصر، ويتم تنفيذ العمليات الأساسية للقائمة باستخدام طرق القائمة المرتبطة.

1. تنفيذ الطرق الأساسية للقائمة:

public interface Queue<E> {

// الطرق الأساسية الأخرى

boolean isEmpty();

int size();

E first();

void enqueue(E element);

E dequeue();

}

الطرق المنفذة تشمل:

* isEmpty(): تحقق مما إذا كانت القائمة فارغة أم لا.
* size(): تعيد حجم القائمة.

1. تعزيز تنفيذ ArrayQueue بطريقة rotate() جديدة:

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

// الطرق الأخرى

public void rotate() {

if (size > 1) {

E element = dequeue();

enqueue(element);

}

}

}

تم تنفيذ طريقة rotate() التي تقوم بإزاحة العنصر الأمامي إلى الخلف من خلال إزالته وإعادة إضافته إلى نهاية القائمة. هذا يؤدي إلى تأثير مماثل للعملية enqueue(dequeue()) ولكن بكفاءة أعلى.

1. تنفيذ طريقة clone() لفئة ArrayQueue:

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

// الطرق الأخرى

@Override

public ArrayQueue<E> clone() {

ArrayQueue<E> clonedQueue = new ArrayQueue<>(data.length);

clonedQueue.front = this.front;

clonedQueue.size = this.size;

System.arraycopy(this.data, 0, clonedQueue.data, 0, this.data.length);

return clonedQueue;

}

}

تم تنفيذ طريقة clone() التي تنشئ نسخة مستقلة من قائمة ArrayQueue الحالية. يتم إنشاء قائمة جديدة ونسخ البيانات والمؤشرات من القائمة الحالية إلى القائمة المستنسخة.

1. تنفيذ طريقة concatenate(LinkedQueue Q2) لفئة LinkedQueue:

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

// الطرق الأخرى

public void concatenate(LinkedQueue<E> Q2) {

if (Q2.isEmpty()) {

return;

}

if (isEmpty()) {

this.list = Q2.list;

} else {

this.list.tail.setNext(Q2.list.head);

this.list.tail = Q2.list.tail;

}

Q2.list = new SinglyLinkedList<>();

}

}

تم تنفيذ طريقة concatenate() التي تأخذ جميع العناصر من قائمة Q2 وتضيفها في نهاية القائمة الأصلية. العملية تتم في وقت ثابت O(1) وتجعل Q2 فارغة بعد الانتهاء.

1. استخدام قائمة لحل مشكلة يوسفوس (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<>();

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

queue.add(i);

}

while (queue.size() > 1) {

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

int removed = queue.remove();

queue.add(removed);

}

queue.remove();

}

return queue.remove();

}

public static void main(String[] args) {

int n = 7;

int k = 3;

int survivor = josephus(n, k);

System.out.println("Survivor: " + survivor);

}

}

تم استخدام الطابور (Queue) لحل مشكلة يوسفوس. تتكون المشكلة من دائرة من الأشخاص حيث يتم استبعاد كل شخص k ويتم تكرار العملية حتى يتبقى شخص واحد كباقي الناجين. يتم تنفيذ هذا باستخدام دورة في الطاب