Data Structure Lab8 : Queue 2022-2023

Topics

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()

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 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()) throw new IllegalStateException("Queue is empty");

E element = (E) data[front];

data[front] = null;

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

size--;

return element;

}

public boolean isEmpty() {

return size == 0;

}

public void rotate() {

if (isEmpty()) return; // No action needed if queue is empty

front = (front + 1) % data.length; // Move front pointer to the next element

}

private void resize(int capacity) {

Object[] newData = new Object[capacity];

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

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

}

data = newData;

front = 0;

}

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

public ArrayQueue<E> clone() {

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

clonedQueue.size = this.size;

clonedQueue.data = new Object[this.data.length];

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

clonedQueue.front = this.front;

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 class LinkedQueue<E> {

private Node front, rear;

private int size;

private static class Node {

Object element;

Node next;

Node(Object e) { element = e; }

}

// Method to concatenate Q2 to the original queue (this)

public void concatenate(LinkedQueue<E> Q2) {

if (Q2.isEmpty()) return; // If Q2 is empty, no action needed

// Append Q2's front to the rear of the current queue (this)

if (this.isEmpty()) {

this.front = Q2.front;

} else {

this.rear.next = Q2.front;

}

// Update rear pointer

this.rear = Q2.rear;

// Empty Q2

Q2.front = Q2.rear = null;

Q2.size = 0;

}

public boolean isEmpty() {

return size == 0;

}

// Other methods like enqueue, dequeue, etc., should be implemented as well.

}

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) {

// Create a queue and enqueue people numbered 1 to n

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

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

queue.offer(i);

}

// Simulate the process of elimination

while (queue.size() > 1) {

// Move k-1 people to the back of the queue

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

queue.offer(queue.poll());

}

// Eliminate the k-th person (dequeue)

queue.poll();

}

// Return the last remaining person

return queue.peek();

}

public static void main(String[] args) {

int n = 7; // Number of people

int k = 3; // Every 3rd person is eliminated

int result = josephus(n, k);

System.out.println("The last remaining person is at position: " + result);

}

}

5. Use a queue to simulate Round Robin Scheduling.

import java.util.LinkedList;

import java.util.Queue;

class Process {

int id;

int burstTime; // the remaining burst time for the process

Process(int id, int burstTime) {

this.id = id;

this.burstTime = burstTime;

}

}

public class RoundRobinScheduling {

public static void roundRobin(Queue<Process> processes, int quantum) {

// Process the queue until all processes are finished

while (!processes.isEmpty()) {

Process current = processes.poll();

// If the burst time of the current process is less than or equal to the quantum,

// it will finish executing in this cycle

if (current.burstTime <= quantum) {

System.out.println("Process " + current.id + " finished after " + current.burstTime + " units.");

} else {

// Otherwise, it will run for the time quantum and needs to go back to the queue with the remaining burst time

current.burstTime -= quantum;

processes.offer(current);

System.out.println("Process " + current.id + " ran for " + quantum + " units, remaining time: " + current.burstTime);

}

}

}

public static void main(String[] args) {

// Creating some sample processes

Queue<Process> processes = new LinkedList<>();

processes.offer(new Process(1, 10)); // Process 1 has a burst time of 10 units

processes.offer(new Process(2, 5)); // Process 2 has a burst time of 5 units

processes.offer(new Process(3, 8)); // Process 3 has a burst time of 8 units

int quantum = 3; // Time quantum for each process

roundRobin(processes, quantum); // Run the Round Robin scheduling

}

}