## 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).

import java.util.Arrays;

public class ArrayQueue<E> {

private static final int DEFAULT\_CAPACITY = 10;

private E[] data;

private int front = 0;

private int size = 0;

// Constructor

public ArrayQueue() {

data = (E[]) new Object[DEFAULT\_CAPACITY];

}

// Get the size of the queue

public int size() {

return size;

}

// Check if the queue is empty

public boolean isEmpty() {

return size == 0;

}

// Enqueue an element into the queue

public void enqueue(E element) {

if (size == data.length) {

resize(2 \* data.length); // Resize if full

}

int avail = (front + size) % data.length; // Next available spot

data[avail] = element;

size++;

}

// Dequeue an element from the queue

public E dequeue() {

if (isEmpty()) {

throw new IllegalStateException("Queue is empty");

}

E result = data[front];

data[front] = null; // Help garbage collection

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

size--;

return result;

}

// Peek at the front element of the queue

public E first() {

if (isEmpty()) {

throw new IllegalStateException("Queue is empty");

}

return data[front];

}

// Rotate the queue

public void rotate() {

if (isEmpty()) {

return; // No need to rotate an empty queue

}

front = (front + 1) % data.length; // Move the front pointer forward

}

// Resize the internal array

private void resize(int newCapacity) {

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

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

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

}

data = newData;

front = 0; // Reset front

}

// Main method for testing

public static void main(String[] args) {

ArrayQueue<Integer> queue = new ArrayQueue<>();

queue.enqueue(10);

queue.enqueue(20);

queue.enqueue(30);

System.out.println("Queue after enqueue operations: " + Arrays.toString(queue.data));

queue.rotate(); // Rotate the queue

System.out.println("Queue after first rotate: " + Arrays.toString(queue.data) + ", Front: " + queue.front);

queue.rotate(); // Rotate the queue again

System.out.println("Queue after second rotate: " + Arrays.toString(queue.data) + ", Front: " + queue.front);

System.out.println("Front element after rotates: " + queue.first());

}

}

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

import java.util.Arrays;

public class ArrayQueue<E> implements Cloneable {

private static final int DEFAULT\_CAPACITY = 10;

private E[] data;

private int front = 0;

private int size = 0;

// Constructor

public ArrayQueue() {

data = (E[]) new Object[DEFAULT\_CAPACITY];

}

// Get the size of the queue

public int size() {

return size;

}

// Check if the queue is empty

public boolean isEmpty() {

return size == 0;

}

// Enqueue an element into the queue

public void enqueue(E element) {

if (size == data.length) {

resize(2 \* data.length); // Resize if full

}

int avail = (front + size) % data.length; // Next available spot

data[avail] = element;

size++;

}

// Dequeue an element from the queue

public E dequeue() {

if (isEmpty()) {

throw new IllegalStateException("Queue is empty");

}

E result = data[front];

data[front] = null; // Help garbage collection

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

size--;

return result;

}

// Peek at the front element of the queue

public E first() {

if (isEmpty()) {

throw new IllegalStateException("Queue is empty");

}

return data[front];

}

// Resize the internal array

private void resize(int newCapacity) {

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

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

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

}

data = newData;

front = 0; // Reset front

}

// Clone method

@Override

public ArrayQueue<E> clone() {

try {

// Perform a shallow copy

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

// Deep copy the internal array

cloned.data = Arrays.copyOf(this.data, this.data.length);

return cloned;

} catch (CloneNotSupportedException e) {

throw new AssertionError("Clone not supported", e);

}

}

// Main method for testing

public static void main(String[] args) {

ArrayQueue<Integer> queue = new ArrayQueue<>();

queue.enqueue(10);

queue.enqueue(20);

queue.enqueue(30);

System.out.println("Original queue: " + Arrays.toString(queue.data));

// Clone the queue

ArrayQueue<Integer> clonedQueue = queue.clone();

System.out.println("Cloned queue: " + Arrays.toString(clonedQueue.data));

// Modify the original queue

queue.enqueue(40);

queue.dequeue();

System.out.println("Modified original queue: " + Arrays.toString(queue.data));

System.out.println("Cloned queue after modification of original: " + Arrays.toString(clonedQueue.data));

}

}

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> {

// Node class

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 E getElement() {

return element;

}

public Node<E> getNext() {

return next;

}

public void setNext(Node<E> next) {

this.next = next;

}

}

private Node<E> head = null; // Head of the queue

private Node<E> tail = null; // Tail of the queue

private int size = 0; // Number of elements in the queue

// Constructor

public LinkedQueue() {}

// Get the size of the queue

public int size() {

return size;

}

// Check if the queue is empty

public boolean isEmpty() {

return size == 0;

}

// Enqueue an element

public void enqueue(E element) {

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

if (isEmpty()) {

head = newNode;

} else {

tail.setNext(newNode);

}

tail = newNode;

size++;

}

// Dequeue an element

public E dequeue() {

if (isEmpty()) {

return null;

}

E element = head.getElement();

head = head.getNext();

size--;

if (isEmpty()) {

tail = null; // Queue is now empty

}

return element;

}

// Peek at the front element

public E first() {

if (isEmpty()) {

return null;

}

return head.getElement();

}

// Concatenate method

public void concatenate(LinkedQueue<E> Q2) {

if (Q2.isEmpty()) {

return; // Nothing to concatenate

}

// Link the tail of the original queue to the head of Q2

if (this.isEmpty()) {

// If the original queue is empty, adopt Q2's nodes

this.head = Q2.head;

this.tail = Q2.tail;

} else {

// Link the tail of the original queue to the head of Q2

this.tail.setNext(Q2.head);

this.tail = Q2.tail;

}

// Update size

this.size += Q2.size();

// Clear Q2

Q2.head = null;

Q2.tail = null;

Q2.size = 0;

}

// Main method for testing

public static void main(String[] args) {

LinkedQueue<Integer> Q1 = new LinkedQueue<>();

LinkedQueue<Integer> Q2 = new LinkedQueue<>();

Q1.enqueue(1);

Q1.enqueue(2);

Q1.enqueue(3);

Q2.enqueue(4);

Q2.enqueue(5);

Q2.enqueue(6);

System.out.println("Q1 size before concatenate: " + Q1.size()); // Output: 3

System.out.println("Q2 size before concatenate: " + Q2.size()); // Output: 3

Q1.concatenate(Q2);

System.out.println("Q1 size after concatenate: " + Q1.size()); // Output: 6

System.out.println("Q2 size after concatenate: " + Q2.size()); // Output: 0

// Print Q1 elements

while (!Q1.isEmpty()) {

System.out.print(Q1.dequeue() + " ");

}

// Output: 1 2 3 4 5 6

}

}

1. Use a queue to solve the Josephus Problem.

import java.util.LinkedList;

import java.util.Queue;

public class JosephusProblem {

// Method to solve the Josephus problem

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

// Create a queue and initialize it with people (1 to n)

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

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

queue.add(i);

}

// Simulate the elimination process

while (queue.size() > 1) {

// Rotate the queue (dequeue and re-enqueue k-1 people)

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

queue.add(queue.remove());

}

// Eliminate the k-th person

queue.remove();

}

// The last remaining person is the survivor

return queue.peek();

}

// Main method for testing

public static void main(String[] args) {

int n = 7; // Number of people in the circle

int k = 3; // Every k-th person is eliminated

int survivor = solveJosephus(n, k);

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

}

}

1. Use a queue to simulate Round Robin Scheduling.

import java.util.LinkedList;

import java.util.Queue;

class Process {

String name; // Process name

int burstTime; // Time required to complete the process

public Process(String name, int burstTime) {

this.name = name;

this.burstTime = burstTime;

}

}

public class RoundRobinScheduling {

// Method to simulate Round Robin Scheduling

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

System.out.println("Starting Round Robin Scheduling...");

while (!processes.isEmpty()) {

// Get the next process from the queue

Process current = processes.poll();

// Simulate the process execution

if (current.burstTime > quantum) {

System.out.println(current.name + " executed for " + quantum + " units.");

current.burstTime -= quantum; // Reduce remaining burst time

processes.add(current); // Re-enqueue the process

} else {

System.out.println(current.name + " executed for " + current.burstTime + " units and completed.");

}

}

System.out.println("All processes completed.");

}

public static void main(String[] args) {

// Create a queue for the processes

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

// Add processes to the queue

processes.add(new Process("P1", 10));

processes.add(new Process("P2", 4));

processes.add(new Process("P3", 7));

processes.add(new Process("P4", 12));

int quantum = 3; // Time slice (quantum)

// Simulate Round Robin Scheduling

simulateRoundRobin(processes, quantum);

}

}