## 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()

Sure! Here's the implementation of a Queue interface, followed by implementations of a Queue using an array and a Queue using linked lists.

1. Queue Interface:

java

Copy

public interface Queue<E> {

boolean isEmpty();

int size();

E first();

void enqueue(E e);

E dequeue();

}

1. Queue Using Array:

java

Copy

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

private static final int DEFAULT\_CAPACITY = 10;

private E[] data;

private int front;

private int rear;

private int size;

public ArrayQueue() {

this(DEFAULT\_CAPACITY);

}

public ArrayQueue(int capacity) {

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

front = 0;

rear = 0;

size = 0;

}

public boolean isEmpty() {

return size == 0;

}

public int size() {

return size;

}

public E first() {

if (isEmpty()) {

throw new NoSuchElementException("The queue is empty.");

}

return data[front];

}

public void enqueue(E e) {

if (size == data.length) {

throw new IllegalStateException("The queue is full.");

}

data[rear] = e;

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

size++;

}

public E dequeue() {

if (isEmpty()) {

throw new NoSuchElementException("The queue is empty.");

}

E element = data[front];

data[front] = null;

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

size--;

return element;

}

}

1. Queue Using Linked Lists:

java

Copy

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

private static class Node<E> {

private E data;

private Node<E> next;

public Node(E data) {

this.data = data;

this.next = null;

}

}

private Node<E> front;

private Node<E> rear;

private int size;

public LinkedQueue() {

front = null;

rear = null;

size = 0;

}

public boolean isEmpty() {

return size == 0;

}

public int size() {

return size;

}

public E first() {

if (isEmpty()) {

throw new NoSuchElementException("The queue is empty.");

}

return front.data;

}

public void enqueue(E e) {

Node<E> newNode = new Node<>(e);

if (isEmpty()) {

front = newNode;

} else {

rear.next = newNode;

}

rear = newNode;

size++;

}

public E dequeue() {

if (isEmpty()) {

throw new NoSuchElementException("The queue is empty.");

}

E element = front.data;

front = front.next;

size--;

if (isEmpty()) {

rear = null;

}

return element;

}

}

In the ArrayQueue implementation, the queue is implemented using an array. The front and rear variables keep track of the indices of the front and rear elements of the queue, respectively. The size of the queue is maintained using the size variable. The circular array implementation is used to handle wrap-around when adding elements to the queue.

In the LinkedQueue implementation, the queue is implemented using a singly linked list. The front and rear variables point to the front and rear nodes of the queue, respectively. The size of the queue is maintained using the size variable.

Both implementations provide the basic methods of a queue: isEmpty() to check if the queue is empty, size() to get the size of the queue, first() to get the element at the front of the queue, enqueue() to add an element to the rear of the queue, and dequeue() to remove and return the element at the front of the queue.

## 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).

To augment the ArrayQueue implementation with a more efficient rotate() method, we can directly update the front and rear indices instead of performing separate enqueue and dequeue operations. This avoids unnecessary array assignments and size modifications.

Here's the updated implementation of the ArrayQueue class with the rotate() method:

java

Copy

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

private static final int DEFAULT\_CAPACITY = 10;

private E[] data;

private int front;

private int rear;

private int size;

public ArrayQueue() {

this(DEFAULT\_CAPACITY);

}

public ArrayQueue(int capacity) {

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

front = 0;

rear = 0;

size = 0;

}

public boolean isEmpty() {

return size == 0;

}

public int size() {

return size;

}

public E first() {

if (isEmpty()) {

throw new NoSuchElementException("The queue is empty.");

}

return data[front];

}

public void enqueue(E e) {

if (size == data.length) {

throw new IllegalStateException("The queue is full.");

}

data[rear] = e;

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

size++;

}

public E dequeue() {

if (isEmpty()) {

throw new NoSuchElementException("The queue is empty.");

}

E element = data[front];

data[front] = null;

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

size--;

return element;

}

public void rotate() {

if (isEmpty()) {

return;

}

rear = (rear - 1 + data.length) % data.length;

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

}

}

In the rotate() method, we update the rear and front indices by subtracting 1 from their current values and taking the modulo operation with data.length to handle wrap-around. This effectively moves the indices one position to the left, making the element at the front of the queue become the new rear element.

Note that the rotate() method assumes that the queue is not empty. If the queue is empty, the method does nothing.

By directly updating the indices, we avoid the overhead of separate enqueue and dequeue operations, resulting in a more efficient implementation of the rotate() method.

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

To implement the clone() method for the ArrayQueue class, we need to create a deep copy of the queue. This means that a new instance of ArrayQueue should be created, and the elements of the original queue should be copied to the new instance.

Here's the updated implementation of the ArrayQueue class with the clone() method:

java

Copy

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

// ... existing code ...

public ArrayQueue<E> clone() {

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

clonedQueue.front = front;

clonedQueue.rear = rear;

clonedQueue.size = size;

clonedQueue.data = Arrays.copyOf(data, data.length);

return clonedQueue;

}

}

In the clone() method, we create a new instance of ArrayQueue called clonedQueue with the same capacity as the original queue. Then, we copy the values of the front, rear, and size variables to the clonedQueue. Finally, we use Arrays.copyOf() to create a deep copy of the data array and assign it to clonedQueue.data. This ensures that the elements of the original queue are copied to the new instance.

Now, you can use the clone() method on an instance of ArrayQueue to create a separate, independent copy of the queue. For example:

java

Copy

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

originalQueue.enqueue(1);

originalQueue.enqueue(2);

originalQueue.enqueue(3);

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

In this example, originalQueue and clonedQueue will be separate instances of ArrayQueue, but the elements stored in originalQueue will be copied to clonedQueue. Any modifications made to one queue will not affect the other.

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.

To implement the concatenate(LinkedQueue Q2) method for the LinkedQueue class that appends all elements of Q2 to the end of the original queue, we can use the following approach:

java

Copy

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

// ... existing code ...

public void concatenate(LinkedQueue<E> Q2) {

if (Q2.isEmpty()) {

return;

}

if (isEmpty()) {

front = Q2.front;

rear = Q2.rear;

} else {

rear.next = Q2.front;

rear = Q2.rear;

}

size += Q2.size;

Q2.front = null;

Q2.rear = null;

Q2.size = 0;

}

}

In the concatenate(LinkedQueue Q2) method, we first check if Q2 is empty. If it is, we don't need to perform any concatenation and simply return.

Next, we handle the case where the original queue (this) is empty. In this case, we update the front and rear references of the original queue to point to the elements of Q2. This effectively makes the original queue a copy of Q2.

If the original queue is not empty, we update the next reference of the current rear node to point to the front of Q2. Then, we update the rear reference of the original queue to point to the rear of Q2.

Finally, we update the size of the original queue by adding the size of Q2. We also reset the front, rear, and size of Q2 to make it an empty queue.

This approach ensures that the concatenation operation runs in O(1) time complexity, as it only involves updating a few references and counters. Additionally, Q2 becomes an empty queue after the concatenation operation.

1. Use a queue to solve the Josephus Problem.

The Josephus Problem is a famous theoretical problem that involves a group of people standing in a circle. Starting from a specified position, every nth person is eliminated until only one person remains. The task is to determine the position of the last person standing.

To solve the Josephus Problem using a queue, we can simulate the elimination process by repeatedly enqueueing and dequeuing elements from the queue until only one element remains.

Here's an example implementation in Java:

java

Copy

import java.util.LinkedList;

import java.util.Queue;

public class JosephusProblemSolver {

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

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

// Enqueue all people into the queue

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

queue.offer(i);

}

while (queue.size() > 1) {

// Dequeue k-1 people and enqueue them back

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

int person = queue.poll();

queue.offer(person);

}

// Dequeue the k-th person (eliminate)

queue.poll();

}

// Return the position of the last person standing

return queue.peek();

}

public static void main(String[] args) {

int n = 7; // Number of people

int k = 3; // Elimination count

int lastPersonStanding = findLastPersonStanding(n, k);

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

}

}

In this example, the findLastPersonStanding method takes two parameters: n represents the number of people in the circle, and k represents the elimination count (every k-th person is eliminated).

We create a Queue<Integer> called queue and enqueue all people into the queue, assigning them numbers from 1 to n.

The elimination process is simulated by repeatedly dequeuing k-1 people from the front of the queue and enqueueing them back at the rear. Then, the k-th person is dequeued (eliminated) from the queue.

This process continues until only one person remains in the queue. Finally, we return the position of the last person standing (the element at the front of the queue).

In the example main method, we define n = 7 (7 people) and k = 3 (every 3rd person is eliminated). The result is printed, indicating the position of the last person standing.

Note that the implementation assumes a 1-based index for the positions of the people in the circle.

1. Use a queue to simulate Round Robin Scheduling.
2. Round Robin Scheduling is a CPU scheduling algorithm commonly used in operating systems. In this algorithm, each process is assigned a fixed time slice called a time quantum. The processes are executed in a cyclical manner, where each process is given a chance to run for the duration of the time quantum before being preempted and the next process is scheduled.
3. To simulate Round Robin Scheduling using a queue, we can use a queue to represent the ready queue, which holds the processes that are ready to be executed.
4. Here's an example implementation in Java:
5. java
6. Copy
7. import java.util.LinkedList;
8. import java.util.Queue;
9. public class RoundRobinScheduler {
10. public static void schedule(int[] processIds, int[] burstTimes, int timeQuantum) {
11. Queue<Integer> readyQueue = new LinkedList<>();
12. // Enqueue all process IDs into the ready queue
13. for (int processId : processIds) {
14. readyQueue.offer(processId);
15. }
16. int currentTime = 0;
17. while (!readyQueue.isEmpty()) {
18. int currentProcess = readyQueue.poll();
19. int remainingBurstTime = burstTimes[currentProcess];
20. // Simulate execution of the current process for the time quantum
21. if (remainingBurstTime > timeQuantum) {
22. currentTime += timeQuantum;
23. remainingBurstTime -= timeQuantum;
24. System.out.println("Process " + currentProcess + " executed for time quantum of " + timeQuantum);
25. } else {
26. currentTime += remainingBurstTime;
27. remainingBurstTime = 0;
28. System.out.println("Process " + currentProcess + " executed for remaining burst time of " + remainingBurstTime);
29. }
30. // Re-enqueue the process if it still has remaining burst time
31. if (remainingBurstTime > 0) {
32. readyQueue.offer(currentProcess);
33. }
34. }
35. }
36. public static void main(String[] args) {
37. int[] processIds = {1, 2, 3, 4, 5}; // Process IDs
38. int[] burstTimes = {10, 4, 6, 3, 8}; // Burst times for each process
39. int timeQuantum = 4;
40. schedule(processIds, burstTimes, timeQuantum);
41. }
42. }
43. In this example, the schedule method takes three parameters: processIds is an array of process IDs, burstTimes is an array of burst times for each process, and timeQuantum is the time quantum for the Round Robin schedule.
44. We create a Queue<Integer> called readyQueue and enqueue all process IDs into the queue.
45. The simulation starts with an initial currentTime of 0. In each iteration of the while loop, we dequeue a process from the readyQueue and simulate its execution for the time quantum.
46. If the remaining burst time of the process is greater than the time quantum, we subtract the time quantum from the remaining burst time, update the currentTime, and print a message indicating that the process was executed for the time quantum.
47. If the remaining burst time is less than or equal to the time quantum, we execute the process for the remaining burst time, set the remaining burst time to 0, update the currentTime, and print a message indicating that the process was executed for the remaining burst time.
48. After executing the process, we check if it still has remaining burst time. If it does, we re-enqueue the process back into the readyQueue to give it another chance to run in the future.
49. The simulation continues until the readyQueue is empty, indicating that all processes have been executed.
50. In the example main method, we define an array of process IDs, an array of burst times for each process, and a time quantum of 4. The schedule method is called with these parameters to simulate the Round Robin Scheduling. The execution details are printed to the console.