Erick Gonzalez Parada #178145

Andre Francois Duhamel Gutiérrez #177315

Alejandro González Díaz #178645

# Activity #6 Priority queues

## Project objective

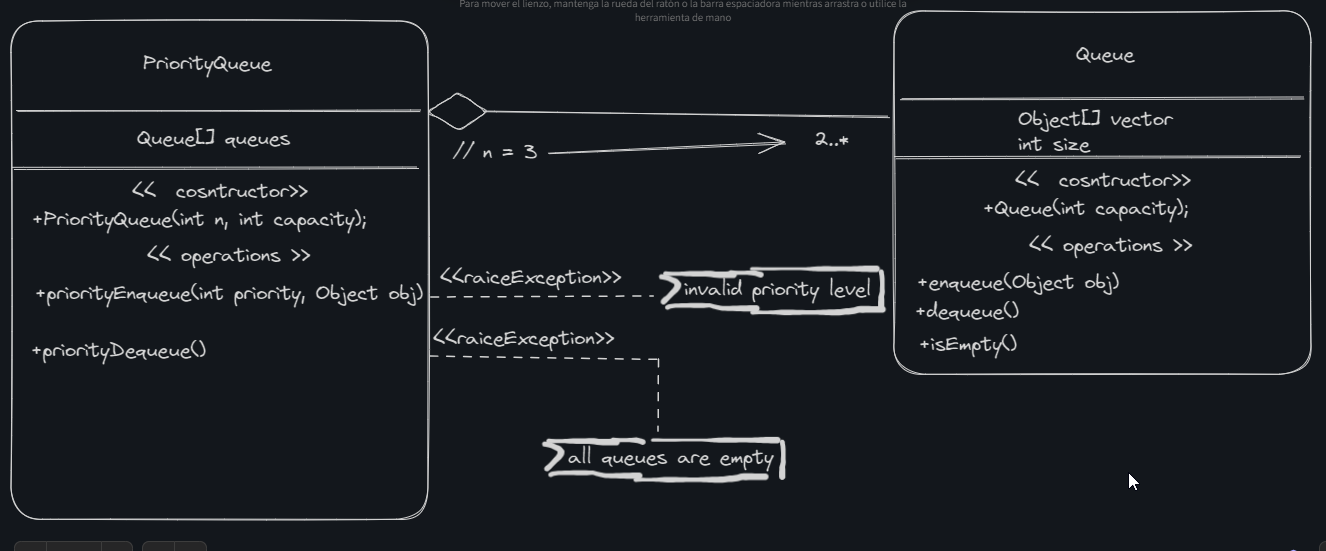
The objective of this activity is to identify the advantages, disadvantages & security aspects of the priority queues, using normal queues and circular queues in its implementation.

[[1]](#_Bibliography)

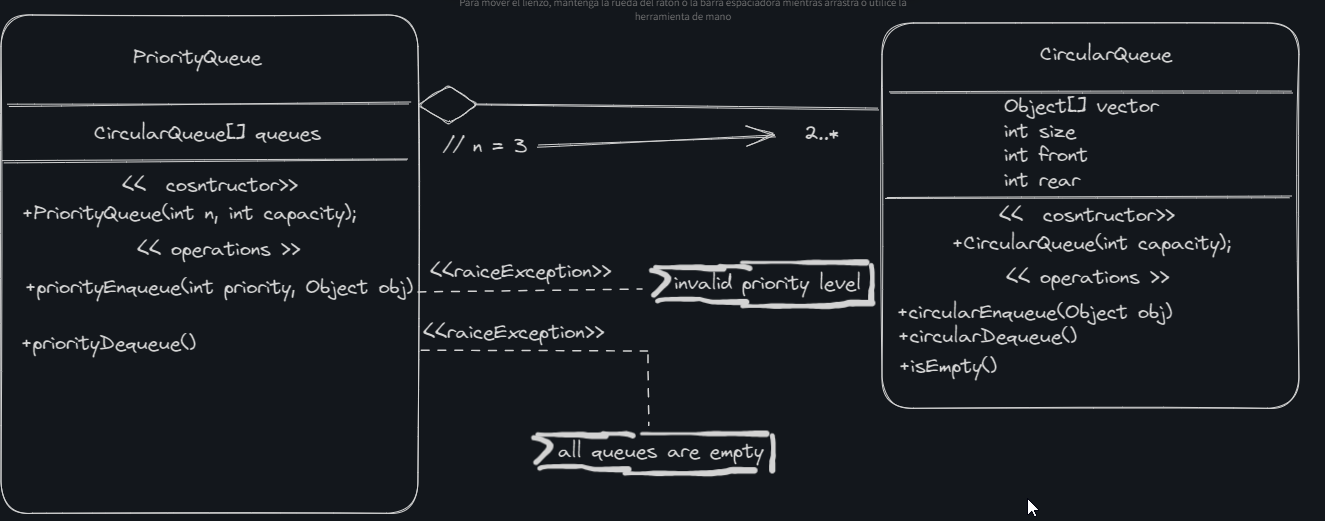
A priority queue is a data structure that orders elements based on a priority level associated with each element. The highest priority elements are kept at the front of the queue and the lowest priority elements are sent to the back. This ensures that elements are always retrieved in sorted priority order. Elements with the same priority are generally served based on their order in the queue.

## Solution design

### First UML diagram for normal queues.



### Second UML diagram for circular queues.



### Approach

We as a team decided to take the simple approach where the priority comes from the index of a vector of Queues i.e. queue[0] get’s the highest priority and queue[n-1] gets the lowest priority.

## Main steps

In both cases we just need it to achieve that the class PriorityQueue with its constructor creates as many priorities we need, note that if we want 5 priorities we only need to modify the int n to be assigned to 5, then the rest of the process acts like a normal queue data structure, nevertheless priorityEnqueue must know the priority in which we want to access and insert an element and priorityDequeue would empty starting from the highest priority to the lowest.

## Security aspects

In both cases we have the following consideration:

Buffer Overflow/Underflow: When you enqueue an element into a full queue or dequeue an element from an empty queue, it can lead to buffer overflow and underflow errors respectively. In the provided code, these cases are handled by throwing exceptions when such operations are attempted.

Data Integrity: Since the queue is implemented using an array, the data is stored in contiguous memory locations. This makes it efficient but also susceptible to data corruption if not handled properly. However, in Java, this risk is mitigated by the JVM’s memory management.

Thread Safety: If our queue is being accessed by multiple threads, we need to ensure that the enqueue and dequeue operations are thread-safe. This can be achieved by synchronizing these methods or using a lock.

Memory Management: In the provided code, the size of the queue (array) is fixed at the time of creation. If we’re dealing with a large amount of data or if the required size of the queue can change dynamically, we might need a dynamic data structure or implement a mechanism to resize the array.

## Analysis

### Code (normal queues)

Note that the classes are separated in different files:

public class PriorityQueue{

Queue[] queues;

//bob el constructor

public PriorityQueue(int n, int capacity) {

queues = new Queue[n];

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

queues[i] = new Queue(capacity);

}

}

//methods

public void priorityEnqueue(int priority, Object obj) throws Exception {

if (priority < 0 || priority >= queues.length) {

throw new Exception("Invalid priority level");

}

queues[priority].enqueue(obj);

}

public Object priorityDequeue() throws Exception {

for (Queue queue: queues) {

if (!queue.isEmpty()) {

return queue.dequeue();

}

}

throw new Exception("All queues are empty");

}

}

class Queue {

Object[] vector;

int size;

public Queue(int capacity) {

vector = new Object[capacity];

size = 0;

}

public void enqueue(Object obj) {

if (size < vector.length) {

vector[size] = obj;

size++;

}

}

public Object dequeue() {

if (size > 0) {

Object obj = vector[0];

System.arraycopy(vector, 1, vector, 0, size - 1);

size--;

return obj;

}

return null;

}

public boolean isEmpty() {

return size == 0;

}

}

public class Main {

public static void main(String[] args) {

PriorityQueue pq = new PriorityQueue(3, 10);

/\*

enqueueing as following

high priority is simply given by the first parameter

the second parameter is an Object instance, so it can be

anything

\*/

try {

pq.priorityEnqueue(0, "High priority vector 1");

pq.priorityEnqueue(1, "Medium priority vector 1");

pq.priorityEnqueue(2, "Low priority vector 1");

pq.priorityEnqueue(0, "High priority vector 2");

pq.priorityEnqueue(1, "Medium priority vector 2");

pq.priorityEnqueue(2, "Low priority vector 2");

pq.priorityEnqueue(0, 4);

} catch (Exception e) {

System.out.println(e.getMessage());

}

// Dequeue elements from the queues

for (int i = 0; i < 6; i++) {

try {

System.out.println(pq.priorityDequeue());

} catch (Exception e) {

System.out.println(e.getMessage());

}

}

}

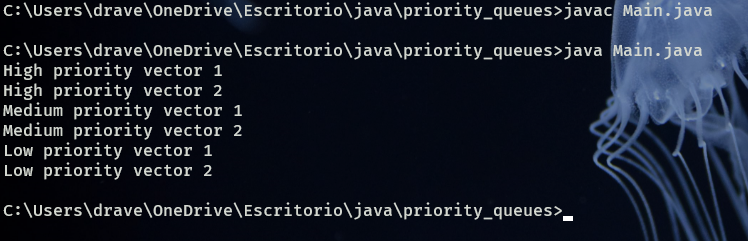
}

### Case 1

This is the case we the main is not modified i.e. as it is shown above, for this we have the following output:

### Output

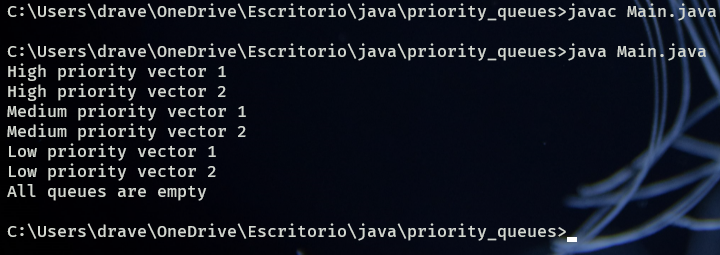
Note that the output is printing the dequeued elements



### Case 2

Modifying the main class to dequeue one more element

### Output



### Code (circular queues)

Note that the classes are separated in different files:

public class PriorityQueue {

CircularQueue[] queues;

public PriorityQueue(int n, int capacity) {

queues = new CircularQueue[n];

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

queues[i] = new CircularQueue(capacity);

}

}

public void priorityEnqueue(int priority, Object obj) throws Exception {

if (priority < 0 || priority >= queues.length) {

throw new Exception("Invalid priority level");

}

queues[priority].circularEnqueue(obj);

}

public Object priorityDequeue() throws Exception {

for (CircularQueue queue : queues) {

if (!queue.isEmpty()) {

return queue.circularDequeue();

}

}

throw new Exception("All queues are empty");

}

}

public class CircularQueue {

Object[] vector;

int size;

int front;

int rear;

public CircularQueue (int capacity) {

vector = new Object[capacity];

size = 0;

front = 0;

rear = 0;

}

public void circularEnqueue(Object obj) throws Exception {

if (size >= vector.length) {

throw new Exception("Queue is full");

}

vector[rear] = obj;

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

size++;

}

public Object circularDequeue() throws Exception {

if (size <= 0) {

throw new Exception("Queue is empty");

}

Object obj = vector[front];

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

size--;

return obj;

}

public boolean isEmpty() {

return size == 0;

}

}

public class Main {

public static void main(String[] args) {

PriorityQueue pq = new PriorityQueue(3, 6);

// Enqueue elements to different priority queues

try {

pq.priorityEnqueue(0, "Priority 1 Object 1");

pq.priorityEnqueue(0, "Priority 1 Object 2");

pq.priorityEnqueue(2, "Priority 3 Object 1");

pq.priorityEnqueue(1, "Priority 2 Object 1");

pq.priorityEnqueue(1, "Priority 2 Object 2");

pq.priorityEnqueue(1, "Priority 2 Object 3");

} catch (Exception e) {

System.out.println(e.getMessage());

}

// Dequeue elements from the queues

for (int i = 0; i < 6; i++) {

try {

System.out.println(pq.priorityDequeue());

} catch (Exception e) {

System.out.println(e.getMessage());

}

}

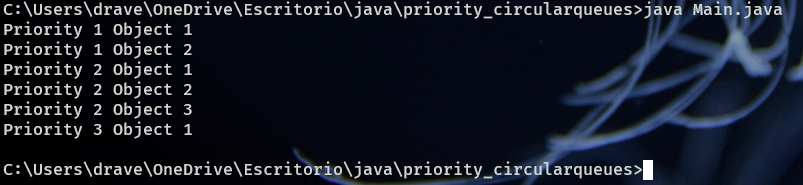
}

}

### Case 1

We should have the same output as before we didn’t change much but the circular queue:

### Output



### Case 2

Now let’s enqueue an element on the highest priority and dequeue to see the circular in action

### Output



## Conclusions

The priority queue is a useful data structure when we have to manage data that is more or less important to programmers. That fact of using it in a circular queue helps us to recycle the same data structure and just change the data in a graceful manner. Also, the fact of using a priority queue allows us to access the highest priority data in a very fast way of just O(1). Nevertheless, the queue and dequeue functions are slow having a time complexity of O(log n).

The objective of the activity was achieved, as we managed to see the advantages of a priority queue and how it helps us to access high priority data in a fast way.

## Bibliography

1. [*Java PriorityQueue*. (s. f.).](#_Project_objective) <https://www.programiz.com/javaprogramming/priorityqueue#google_vignette>