Queue (Implementation)

How is Queue implemented in Java? What are some of the main methods that each Queue have? That is what we are going to cover in this lesson.

We'll cover the following

Implementation of Queues

Queues are implemented in many ways. They can be represented by using an array, a linked list, or even a stack. That being said, an array is most commonly used because it's the easiest way to implement Queues. As discussed in the previous lesson, a typical Queue must contain the following standard methods:

- `enqueue (datatype V)`
- `datatype dequeue()`
- `boolean isEmpty()`
- `boolean isFull()`
- `datatype top()`

Before we take a look at these methods one by one, let's construct a Queue class with an integer data type and create an instance. We will make a class with 5 data members to hold the following information:

- The **array** that will contain all the elements
- The **maxSize** is the size of this array
- The **front** element of the Queue
- The **back** element of the Queue
- The **currentSize** of elements in the Queue

The code given below shows how to construct the Queue class:

main.java

Queue.java

```
public class Queue<V> {
    private int maxSize;
    private V[] array;
    private int front;
    private int back;
    private int currentSize;

    /*
    Java does not allow generic type arrays. So we have used an
    array of Object type and type-casted it to the generic type V.
    This type-casting is unsafe and produces a warning.
    Comment out the line below and execute again to see the warning.
    */
    @SuppressWarnings("unchecked")
    public Queue(int maxSize) {
        this.maxSize = maxSize;
        array = (V[]) new Object[maxSize];
        front = 0;
        back = -1;
        currentSize = 0;
    }

    public int getMaxSize() {
        return maxSize;
    }

    public int getCurrentSize() {
        return currentSize;
    }

}

```

Adding Helper Functions

Now before adding the `enqueue` and `dequeue` methods into this class, we need to implement some helper methods to keep the code simple and understandable. Here's the list of helper functions that we will implement in the code below:

- `isEmpty()`
- `isFull()`
- `top()`

Now run the following code and see if the helper function outputs correctly.

main.java

Queue.java

```
public class Queue<V> {
    private int maxSize;
    private V[] array;
    private int front;
    private int back;
    private int currentSize;

    /*
    Java does not allow generic type arrays. So we have used an
    array of Object type and type-casted it to the generic type V.
    This type-casting is unsafe and produces a warning.
    Comment out the line below and execute again to see the warning.
    */
    @SuppressWarnings("unchecked")
    public Queue(int maxSize) {
        this.maxSize = maxSize;
        array = (V[]) new Object[maxSize];
        front = 0;
        back = -1;
        currentSize = 0;
    }

    public int getMaxSize() {
        return maxSize;
    }

    public int getCurrentSize() {
        return currentSize;
    }

    public boolean isEmpty() {
```

For the above code, `isEmpty()` should return `true` and `isFull()` should return `false`. Now, we will extend this code with the `enqueue` and `dequeue` methods to add and remove elements, respectively.

main.java

Queue.java

```
public class Queue<V> {
    private int maxSize;
    private V[] array;
    private int front;
    private int back;
    private int currentSize;

    /*
    Java does not allow generic type arrays. So we have used an
    array of Object type and type-casted it to the generic type V.
    This type-casting is unsafe and produces a warning.
    Comment out the line below and execute again to see the warning.
    */
    @SuppressWarnings("unchecked")
    public Queue(int maxSize) {
        this.maxSize = maxSize;
        array = (V[]) new Object[maxSize];
        front = 0;
        back = -1;
        currentSize = 0;
    }

    public int getMaxSize() {
        return maxSize;
    }

    public int getCurrentSize() {
        return currentSize;
    }

    public boolean isEmpty() {
        return currentSize == 0;
    }

    public boolean isFull() {
        return currentSize == maxSize;
    }

    public V top() {
        return array[front];
    }

    public void enqueue(V value) {
        if (isFull())
            return;
        back = (back + 1) % maxSize; //to keep the index in range
```

```
47          array[back] = value;
48          currentSize++;
49      }
50
51      public V dequeue() {
52          if (isEmpty())
53              return null;
54
55          V temp = array[front];
56          front = (front + 1) % maxSize; //to keep the index in range
57          currentSize--;
58
59          return temp;
60      }
61  }
62
```

If you look at the output of the code, you can see that the elements are enqueued in the back and dequeued from the front. This means that our queue works perfectly.

In the last few chapters, we discussed some fundamental data structures and went through their implementations. Soon, we will take a look at some advanced data structures which are mainly derived using the basic data structures that we have studied so far.

For now, let's solve some challenges!

Back

What is a Queue?

Next

Challenge 1: Generate Binary Number...

Completed