University of Southampton

COMP1201
Tutorial 2
Implementing a Queue

Krasimir Marinov
Email: km1u17@soton.ac.uk
Student ID: 29545528
March 16, 2018

1 Implementing a Queue

1.1 CircularArrayQueue

Write a resizable CircularArrayQueue class which implements MyQueue. It should have a getCapacityLeft() method which tells the user how many items can be added to the array before the array needs to be resized.

```
import java.util.NoSuchElementException;
/**
 * Class CircularArrayQueue
 * Implements MyQueue
 */
public class CircularArrayQueue implements MyQueue {

    // Int for the size of the CircularArrayQueue
    // Int for the head of the CircularArrayQueue - where elements
        are dequeued
    private int head;
    // Int for the tail of the CircularArrayQueue - where elements
        are enqueued
    private int tail;
    // Array of Integers - required as we may have Null
    private Integer[] array;

    /**
     * Constructor for the CircularArrayQueue
     * Sets the default size to be 1024
     */
    public CircularArrayQueue() {
        // Setting the initial size
        this.size = 1024;
        // Creating the array with the given size
        array = new Integer[size];
        // Setting the initial values of the head and the tail
        head = 0;
        tail = 0;
    }

    /**
     * Constructor for CircularArrayQueue with given size as an
```

```
     argument
 */
public CircularArrayQueue(int size) {
    // Setting the size
    this.size = size;
    // Creating the array with the given size
    array = new Integer[size];
    // Setting the initial values of the head and the tial
    head = 0;
    tail = 0;
}

/**
 * Enqueue method
 * We enqueue the given int
 */
@Override
public void enqueue(int in) {
    // If there is no space to insert an element
    if (getCapacityLeft() == 0 && array[tail] != null) {

        // We create a newArray with doubled size
        Integer[] newArr = new Integer[size * 2];

        /**
         * We copy the elements from the previous Array
         * But in the order as they were inserted
         * We keep the order because we dequeue from the
             previous array
         */
        for (int i = 0; i < size; i++) {
            // We dequeue and keep the order
            newArr[i] = this.dequeue();
        }
        // The array is now resized
        array = newArr;

        /**
         * And the tail is equal to the size
         * Then we add the new element to the tail
```

```
         */
        tail = size;
        array[tail] = in;
        /** The head points to the first element */
        head = 0;
        /** We update the size as it is doubled */
        size *= 2;
        /** The tail is also updated to the next index */
        tail = (tail + 1) % size;

    } else {
        /**
         * Else we have enough space
         * We insert the element at the tail
         * And the tail is incremented by 1 and mod is applied
         * In that way we can be sure that the array will be
            filled before resized
         */
        array[tail] = in;
        tail = (tail + 1) % size;
    }
}

/**
 * Dequeue follows the same methodology as enqueue
 * @throws NoSuchElementException
 */
@Override
public int dequeue() throws NoSuchElementException {
    // The element to be returned
    int element;
    // If we have an element we return it
    if (array[head] != null) element = array[head];
    else {
        // Otherwise, we throw new NoSuchElementException
        throw new NoSuchElementException();
    }
    // The head is then set to be null
    array[head] = null;
    /**
```

```
     * And the head is advanced to show the next element
     * Again mod is applied because it is a circular array
     */
    head = (head + 1) % size;
    // We return the element
    return element;
}
/**
 * Method noItems()
 * Return the number of elements
 */
@Override
public int noItems() {
    // If the head is equal to the tail and it is null then the
       circular array is empty
    if (head == tail && array[head] == null) return 0;
       // Otherwise
    else {
        /**
         * If the head is equal to the tail and the element
            pointed by them is not null
         * Then, the array is full and we return the size
         */
        if (head == tail && array[head] != null) return size;
        else {
            // Otherwise, we calculate the number of elements
            return (size - head + tail) % size;
        }
    }
}
/**
 * Method isEmpty
 * Return true if it is empty
 * Otherwise, it returns false
 */
@Override
public boolean isEmpty() {
    /**
     * If the head is equal to the tail and the element pointed
        by them is null
```

```
         * Then, the array is empty and we return true
         */
        if (head == tail && array[head] == null) return true;
        // Otherwise, we return false
        else return false;
    }
    /**
     * Method getCapacityLeft
     * Returns the number of elements we can insert before the
         CircularArray is full
     */
    public int getCapacityLeft() {
        return (size - noItems()) % size;
    }
}
```

After testing the **CircularArrayQueue** against the junit test program - **CircularArrayQueueTest** , our class passes all 6 tests successfully.

1.2 CircularArrayRing

The CircularArrayRing<E>class which implements the Ring<T>interface and extends AbstractCollection<E>. A ring is a list that forgets. That is, it is a data structure which remembers the last N entries.

```
import java.util.AbstractCollection;
import java.util.Iterator;
import java.util.NoSuchElementException;
/**
 * Class CircularArrayRing<E>
 * A list that forgets
 * Data structure which remembers the last N entries
 * Extends AbstractCollection<E>
 * Implements Ring<E>
 */
public class CircularArrayRing<E> extends AbstractCollection<E>
    implements Ring<E> {
    // Int for the size of the CircularArrayQueue
    private int size;
    // Array of E (Generics) that can hold different Objects
```

```java
private E[] array;
// Int for the tail of the CircularArrayRing
private int tail;
/**
 * Constructor for the CircularArrayRing
 * Sets the default size to be 12
 */
public CircularArrayRing() {
    // Sets the default size to be 12
    this.size = 12;
    // Cast to E[] Type
    array = (E[]) new Object[size];
    // The tail is set to be 0
    tail = 0;
}
/**
 * Constructor for the CircularArrayRing
 * Sets the size to be the given size in the argument
 */
public CircularArrayRing(int size) {
    // Sets the size
    this.size = size;
    // Cast to E[] Type
    array = (E[]) new Object[size];
    // The tail is set to be 0
    tail = 0;
}
/**
 * Size method
 * Returns the number of elements of the CircularArrayRing
 */
@Override
public int size() {
    // We return the size if we do have a element
    if (array[tail % size] != null) return size;
    return tail;

}
/**
 * Iterator method
```

```
 * Returns an Iterator<E> object
 */
@Override
public Iterator iterator() {
    // We create a new Iterator
    Iterator<E> iterator = new Iterator<E>() {
        /**
         * We set the index of the current Element
         * It is (tail-1) because the iterator iterates
             backwards from the most recent element added
         */
        int currentE = (tail - 1 + size) % size;
        // Count for the elements we have iterated through
        int elementsCounted = 0;
        /**
         * Method hasNext((
         * Returns true if we have next element
         * Otherwise, false
         */
        @Override
        public boolean hasNext() {
            // If we haven't added any element then we return
                false
            if (tail == 0 && array[tail] == null) return false;
            // If the current Element is null then we return
                false
            if (array[currentE] == null) return false;
            /**
             *  If the current element is not null but we have
                 reached the size of the Array
             *  Then, the CircularArrayRing is full and we
                 return false
             */
            if (array[currentE] != null && elementsCounted ==
                size()) return false;
            // Otherwise, we return the next element
            else return true;
        }
        /**
         * Method next()
```

```
         * Returns the nextElement
         */
        @Override
        public E next() {
            // If we have a next Element
            if (hasNext()) {
                // We get it
                E element = array[currentE];
                // The iterator iterates backwards
                currentE = (currentE - 1 + size) % size;
                // The counter for the elemenrs iterated of the
                    Array is also incremented
                elementsCounted++;
                // We return the element
                return element;
            } // Else we throw a NoSuchElementException
            else throw new NoSuchElementException("No such
                element");
        }
        /**
         * Method Remove throws an UnsupportedOperationException
         */
        @Override
        public void remove() throws
            UnsupportedOperationException {
            throw new UnsupportedOperationException("Remove is
                not supported!!!");
        }
    };
    /** We return the iterator */
    return iterator;
}
/**
 * Method get() gets the last added variables first
 * Get(0) gets the last item you added
 * Get(1) gets the previous item and so on...
 * Throws an IndexOutOfBoundsException if the index is either
    larger than the number of items added or larger than the
    ring size
 */
```

```
    @Override
    public E get(int index) throws IndexOutOfBoundsException {
        // If the index is negative or is either larger than the
            number of items added or larger than the ring size
        if (index < 0 || index > size - 1 || index > size() - 1)
            // It throws an new IndexOutOfBoundsException
            throw new IndexOutOfBoundsException("Ring not big
                enough");
        // Otherwise, it returns the element
        return array[(tail - index - 1 + size) % size];
    }
    /*
     * Method add
     * Adds an element to the CircularArrayRing
     */
    @Override
    public boolean add(Object a) {
        // We insert the element
        array[tail] = (E) a;
        // The tail is incremented and mod is applied in order to
            fill the array
        tail = (tail + 1) % size;
        // We return true
        return true;
    }
}
```

After testing the **CircularArrayRing** against the **CircularArrayRingTest** junit test program, our class passes all 5 tests successfully.

1.3 Suggestions

Students could also be asked to implement a Queue with a regular array. So then, they could see the differences between the two data structures and the flaws of the implementation using a regular array. In that way they could understand that with a circular Queue, they just have to increase the pointer and resize if necessary. As a result, there are less operations on an update and that gives a better performance.