Name: Patrick Garcia, Havin Lim

Lab Name: Lab28~30

Lab Due Date: November 13th 2022

Written/online sources used: None

Help obtained: None

Add the statement: “We confirm that the above list of sources is complete AND that we have not talked to anyone else (e.g., CSC 207 students) about the solution to this problem.”

LAB 28

EXTRA

Node

```
public class Node {
    public Node below = null;
    public Node above = null;
    int value;

    public Node(int value) {
            this.value = value;
    }
}
```

STACK

```
public class Stack {
    private int capacity;
    public Node top;
    private int size = 0;

    public Stack(int capacity) {
        this.capacity = capacity;
    }

    public boolean isAtCapacity() {
        return capacity == size;
    }

    private void join(Node above, Node below) {
        if (below != null)
            below.above = above;
        if (above != null)
            above.below = below;
    }

    /**
     * Pushes an element into the top of Stack. If Stack is empty, creates another Stack.
     * If it is at capacity, returns false
     * @param v -> value to insert the stack
     * @return boolean
     */
    public boolean push(int v) {
        if(this.isAtCapacity()) {
            return false;
        }
        if(size == 0) {
            Node newNode = new Node(v);
            newNode.above = null;
            newNode.below = null;

            top = newNode;
            size++;

            return true;
        }
        else {
            Node newNode = new Node(v);
            newNode.below = top;
            newNode.above = null;

            top = newNode;
            size++;
```

```
                return true;
            }
        }

        /**
         * Returns the top value and removes it from Stack
         * @return integer -> the top value
         */
        public int pop() {
            int keep = top.value;

            if(top.below == null) {
                top = null;
                return keep;
            }
            top = top.below;
            top.above = null;
            size--;

            return keep;
        }

        /**
         * Checks if Stack is Empty
         * @return boolean
         */
        public boolean isEmpty() {
            return size == 0;
        }

        /**
         * Returns the size of the Stack
         * @return int -> size of stack
         */
        public int size() {
            return size;
        }
    }
```

SET OF STACKS

```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;

public class SetOfStacks {

    private List<Stack> list; //list of stacks
    private int capacity; // Capacity of inner stacks
    private int size; // Total number of elements in the SetOfStackss

    /**
     * The constructor for Set of Stacks
     * @param capacity -> the capacity for each of its inner stacks
     */
    public SetOfStacks(int capacity){
            this.capacity = capacity;
            list = new LinkedList<>();
    }

    /**
     * An instance variable that returns the last stack of the list
     * @return the last Stack of the list
     */
    private Stack getLastStack() {
            if (list.size() == 0)
                    return null;
            return list.get(list.size() - 1);
    }

    /**
     * Inserts an element in the last Stack of the list and creates a new Stack if exceeds
the previous'
     * capacity
     * @param v -> the value to be inserted in the last Stack
     */
    public void push(int v) {
            if(size == 0) {
                    list.add(new Stack(this.capacity));
                    list.get(0).push(v);
                    size++;
            }
            else {
                    if(this.getLastStack().push(v)) {
                            size++;
                    }
                    else {
                            list.add(new Stack(this.capacity));
```

```
                        this.getLastStack().push(v);
                        size++;
                }
        }
}

/**
 * Gets and removes the first element of the last Stack of the list. If the Stack
 * gets empty, it removes it.
 * @return
 */
public int pop() {
        if (size == 0) {
                System.out.println("There are no elements to pop.");
                return 0;
        }

        int keep = this.getLastStack().pop();

        if(this.getLastStack().top == null) {
                list.remove(this.getLastStack());
        }

        return keep;
}

/**
 * Returns the total of elements from all Stacks in the list
 * @return integer
 */
public int size() {
        return size;
}

/**
 * Returns the current capacity of all Stacks
 * @return integer
 */
public int capacity() {
        return capacity;
}

/**
 * Returns the number of Stacks in the list
 * @return integer
 */
public int numInnerStacks() {
        return list.size();
```

```
    }
}
```

TESTER CODE

```
public class SetOfStacksTester {


    public static void main (String[] args) {
        SetOfStacks stack = new SetOfStacks(7);

        System.out.println("Pushing the following elements into the Stack:");
        for (int i=0; i < 20; i++) {
            System.out.print(" " + i);
            stack.push(i);
        }

        System.out.println("\nNumber of Elements in Stack: " + stack.size());
        System.out.println("Capacity of each inner Stack: " + stack.capacity());
        System.out.println("Number of Stacks in the list: " + stack.numInnerStacks());

        System.out.println("\nPopping the following elements from the Stack:");
        for (int i=0; i < 20; i ++) {
            System.out.print(" " + stack.pop());
        }

        System.out.println("\nNumber of Elements in Stack: " + stack.size());
        System.out.println("Capacity of each inner Stack: " + stack.capacity());
        System.out.println("Number of Stacks in the list: " + stack.numInnerStacks());
    }
}
```

OUTPUT

```
Pushing the following elements into the Stack:
 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Number of Elements in Stack: 20
Capacity of each inner Stack: 7
Number of Stacks in the list: 3

Popping the following elements from the Stack:
 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Number of Elements in Stack: 20
Capacity of each inner Stack: 7
Number of Stacks in the list: 0
```

LAB 29

DEQUEUE CLASS

```
package Lab29;

import java.util.NoSuchElementException;

// from Weiss chapter 16 and Jerod Weinman
// modified by:  Your names here!!

/**
 * Dequeue (Double Ended Queue) class
 *
 * Like a queue, but with access allowed at both ends.
 *
 ******** PUBLIC OPERATIONS *******
 * void addFront( x )    --> Insert x at front
 * void addRear( x )     --> Insert x at rear
 * AnyType getFront()  --> Return ("least" recently inserted) item at front
 * AnyType getRear()   --> Return ("most" recently inserted) item at rear
 * AnyType removeFront() --> Return and remove item from front
 * AnyType removeRear( ) --> Return and remove item from rear
 * boolean isEmpty()   --> Return true if empty; else false
 * void makeEmpty()    --> Remove all items
 *
 ******** ERRORS - THROWS EXCEPTION *******
 * getFront, getRear, removeFront, removeRear on empty dequeue
 */

public class Dequeue<AnyType>
{
    ////////////////////////////////////////////////////////////////////////
    // PRIVATE FIELDS
    ////////////////////////////////////////////////////////////////////////

    private AnyType[] theArray;
    private int front, back, currentSize;
    private int sizeFront, sizeBack;
    private int capacity;

    private static final int DEFAULT_CAPACITY = 5;

    ////////////////////////////////////////////////////////////////////////
    // PUBLIC METHODS
    ////////////////////////////////////////////////////////////////////////

    public int length() {
```

```
        return theArray.length;
}

/**
 * Construct and initialize a new empty Dequeue object
 */
@SuppressWarnings("unchecked")
public Dequeue()
{
        theArray = (AnyType[]) new Object[DEFAULT_CAPACITY];

        makeEmpty();
}

/**
 * Indicate whether this dequeue is empty.
 *
 * @return true if no items are in the dequeue; else false
 */
public boolean isEmpty()
{
        if (this.currentSize == 0)
                return true;

        return false;
}

/**
 * Removes all items from this dequeue.
 */
@SuppressWarnings("unchecked")
public void makeEmpty()
{
        theArray = (AnyType[]) new Object[DEFAULT_CAPACITY];
        sizeFront = 0;
        sizeBack = 0;
        currentSize = sizeFront+sizeBack;
        capacity = DEFAULT_CAPACITY;

        front = 0;
        back = theArray.length - 1;

}


/**
 * Get the item at front the front of this Dequeue
 *
```

```
 * @return item at the front of this Dequeue
 * @throws NoSuchElementException  with message "Empty" if isEmpty()
 */
public AnyType getFront() throws NoSuchElementException
{
	return theArray[front];
}


/**
 * Get the item at front the rear of this Dequeue
 *
 * @return item at the rear of this Dequeue
 * @throws NoSuchElementException  with message "Empty" if isEmpty()
 */
public AnyType getRear() throws NoSuchElementException
{
	return theArray[back];
}

/**
 * Remove the item at the front of this Dequeue
 *
 * @return item at the front of this Dequeue
 * @throws NoSuchElementException  with message "Empty" if isEmpty()
 */
public AnyType removeFront() throws NoSuchElementException
{
	if(this.isEmpty())
	{
		throw new NoSuchElementException("null argument");
	}
	AnyType temp = this.getFront();
	int i = front;

	while(theArray[i] != null) {
		//	System.out.println("Current i = " + theArray[i]);
		//	System.out.println("Current i+1 = " + theArray[i+1]);
		theArray[i] = theArray[i+1];
		i++;
	}
	currentSize--;
	sizeFront--;
	return temp;
}

/**
 * Remove the least at the rear of this Dequeue
```

```
 *
 * @return item at the rear of this Dequeue
 * @throws NoSuchElementException  with message "Empty" if isEmpty()
 */
public AnyType removeRear() throws NoSuchElementException
{
	if(this.isEmpty())
	{
		throw new NoSuchElementException("null argument");
	}
	AnyType temp = this.getRear();
	int i = back;

	while(theArray[i] != null) {
		theArray[i] = theArray[i-1];
		i--;
	}
	currentSize--;
	sizeBack--;

	return temp;

}

/** Insert item at the front of this Dequeue.
 */
public void addFront( AnyType x )
{
	move(true);
	theArray[front] = x;

	currentSize++;
	sizeFront++;

	if((theArray[sizeFront] != null)) {
		expand();
		back = theArray.length - 1;
	}
}

/** Insert item at the rear of this Dequeue.
 */
public void addRear( AnyType x )
{
	move(false);
	theArray[back] = x;

	currentSize++;
```

```
        sizeBack++;


        if((theArray[theArray.length - sizeBack - 1] != (null))) {
                expand();
                back = theArray.length - 1;
        }

    }


    ////////////////////////////////////////////////////////////////////////
    // PRIVATE METHODS
    ////////////////////////////////////////////////////////////////////////
    /**
     * Moves the elements in the front of the array forward or the elements from the back
     * of the array backward.
     * @param i -> boolean, if true -> moves forward; if false -> moves backwards
     */
    private void move(boolean i) {
        if(i) {
                //    System.out.println("move-true");
                for(int j = sizeFront - 1; j > -1; j--) {
                        //    System.out.println("  copy element " + theArray[j] + " to element "
+ theArray[j+1]);
                        theArray[j+1] = theArray[j];
                        //    System.out.println("  " + theArray[j] + " " + theArray[j+1]);
                }
        }
        else {
                //    System.out.println("move-false");
                for(int j = theArray.length - sizeBack; j < theArray.length; j++) {
                        //    System.out.println("  copy element " + theArray[j] + " to element "
+ theArray[j-1]);
                        theArray[j-1] = theArray[j];
                        //    System.out.println("  " + theArray[j] + " " + theArray[j-1]);
                }
        }
    }

    /**
     * Expands the array to twice its size
     */
    @SuppressWarnings("unchecked")
    private void expand()
    {
        //        System.out.println("It is expanding");
        AnyType[] keep = (AnyType[]) new Object[theArray.length];
```

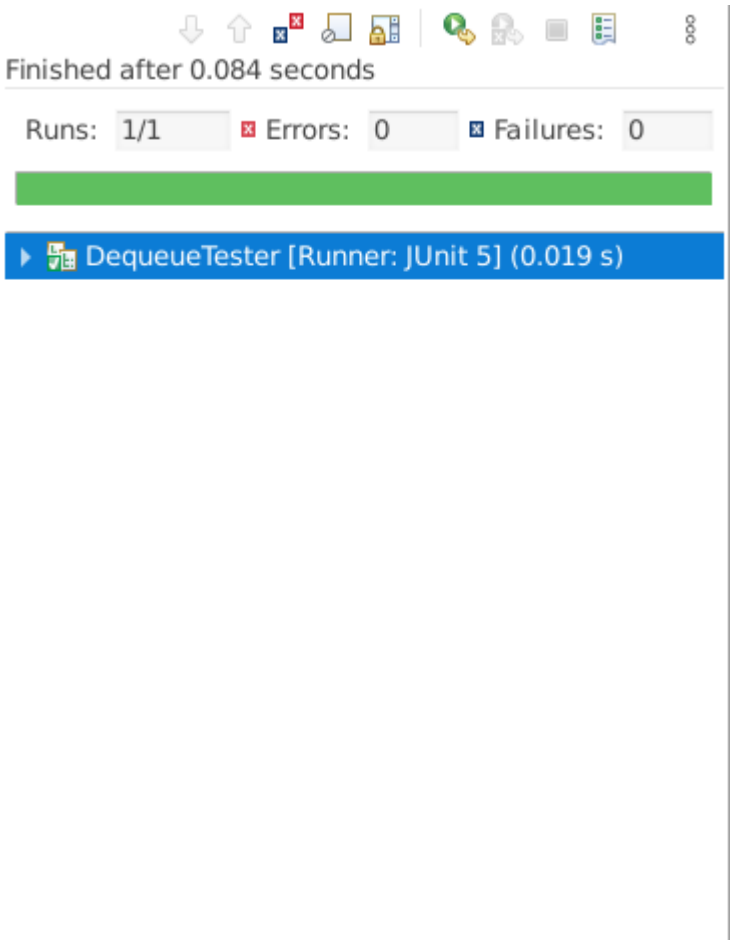
```
        for(int i = 0; i < theArray.length; i++) {
                //                  System.out.println("  move element " + theArray[i] + " to array
keep");
                keep[i] = theArray[i];
        }

        capacity *= 2;
        //      System.out.println("  new capacity = " + capacity);
        theArray = (AnyType[]) new Object[capacity];
        //adding to the front
        for(int i = 0; i < sizeFront; i++)
                theArray[i] = keep[i];

        //adding to the back
        int sk = keep.length - 1;
        for(int i = theArray.length-1; i >= theArray.length - sizeBack; i--) {
                theArray[i] = keep[sk];
                sk--;
        }

    }


    /**
     * Internal method to increment with wraparound.
     *
     * @param x any index in theArray's range.
     * @return x+1, or 0 if x is at the end of theArray
     */
    private int increment(int x)
    {
        if (++x == theArray.length)
                x = 0;

        return x;
    }


    /**
     * Internal method to decrement with wraparound.
     *
     * @param x any index in theArray's range.
     * @return x-1, or theArray.length-1 if x is at the beginning of theArray
     */
    private int decrement(int x)
    {
        if(--x == theArray.length)
                return 0;
```

```
        return x;
    }


    /** Internal method to expand theArray.
     */
    private void doubleDequeue()
    {

        // Create a new array of double capacity
        AnyType[] newArray = (AnyType[]) new Object[theArray.length * 2];

        // Copy elements that are logically in the dequeue
        for (int i=0 ; i<currentSize ; i++, front=increment(front) )
            newArray[i] = theArray[front];

        // settings for a "new" array ... use as an example for makeEmpty
        theArray = newArray;
        front = 0;
        back = currentSize - 1;

    }

}
```

TESTER

```
package Lab30;

import static org.junit.jupiter.api.Assertions.*;

import java.util.NoSuchElementException;

import org.junit.jupiter.api.Test;

import Lab29.Dequeue;

class DequeueTester {

	@Test
	void test() {
		Dequeue<Integer> d = new Dequeue<Integer>();

		assertEquals(true, d.isEmpty());
		d.addFront(5);
		assertEquals(false, d.isEmpty());
		d.addRear(3);
		d.addFront(10);
		d.addRear(20);
		d.addFront(12);
		d.addFront(30);
		d.removeRear();
		d.removeFront();
		// 12 10 5 3
		assertEquals(12,d.getFront());
		assertEquals(3,d.getRear());

		d.makeEmpty();
		assertEquals(true, d.isEmpty());
		d.addFront(523);
		assertEquals(false, d.isEmpty());
		d.removeFront();
		assertEquals(true, d.isEmpty());
		d.addRear(5000);
		assertEquals(false, d.isEmpty());
		d.removeRear();
		assertEquals(true, d.isEmpty());

		boolean thrown = false;
		try {
			d.removeFront();
		} catch(NoSuchElementException nsee) {
			thrown = true;
		}
```

```
            assertTrue(thrown);

            d.makeEmpty();
            assertEquals(true, d.isEmpty());


    }

}
```



LAB 30

Circular List Class

```
package Lab30;

import java.util.AbstractCollection;
import java.util.Iterator;

class CircularList<AnyType> extends AbstractCollection<AnyType>{
   private Node<AnyType> first;
   private int size;

   /**
    * The constructor for CircularList, sets size to 0;
    */
   public CircularList(){
         size = 0;
   }

   //Inner Class NODE
   @SuppressWarnings("hiding")
   public class Node<AnyType> {
         public AnyType data;
         public Node<AnyType> next;
         public Node<AnyType> prev;

         public Node(AnyType data) {
               this.data = data;
               next = null;
               prev = null;
         }
   }

   /**
    * Add any type element to Linked List
    * @param data -> any type
    * @return -> boolean
    */
   @Override
   public boolean add(AnyType data) {
         if(this.isEmpty()) {
               Node<AnyType> FirstNewNode = new Node<>(data);

               FirstNewNode.next = FirstNewNode;
               FirstNewNode.prev = FirstNewNode;
               first = FirstNewNode;
               size++;
```

```
            return true;
        }
        else {
            Node<AnyType> newNode = new Node<>(data);

            newNode.prev = first.prev;
            first.prev.next = newNode;
            first.prev = newNode;
            newNode.next = first;
            size++;

            return true;
        }
    }//end ADD method

    /**
     * Method that returns the data of the node at the index
     * @param index -> integer
     * @return -> AnyType
     */
    public AnyType get(int index) {
        Node<AnyType> temp = first;

        if(index<0) {
            for(int i = 0; i<index*(-1); i++){
                temp = temp.prev;
            }
        }
        else {
            for(int i = 0; i<index; i++){
                temp = temp.next;
            }
        }

        return (AnyType) temp.data;
    }//end of get method


    /**
     * Removes a data from the list and returns true if successful, false if otherwise.
     */
    public boolean remove(Object data) {
        Node<AnyType> temp = first;

        for(int i = 0; i < size; i++) {
            if(temp.data.equals(data)) {
                remove(temp);
```

```
                    return true;
                }

                temp = temp.next;
            }

            return false;
        }

        private AnyType remove(Node<AnyType> temp) {
            AnyType element = temp.data;

            temp.prev.next = temp.next;
            temp.next.prev = temp.prev;
            temp.next = null;
            temp.prev = null;

            size--;
            return element;
        }

        /**
         * Checks if list is empty
         */
        public boolean isEmpty() {
            return (this.size == 0);
        }

        @Override
        public Iterator<AnyType> iterator() {
            // TODO Auto-generated method stub
            return null;
        }

        /**
         * Checks the size of the list
         */
        @Override
        public int size() {
            return this.size;
        }

    }
```

TESTER CLASS

```
package Lab30;

public class CircularListTester {
   public static void main(String[]args) {
         CircularList<Integer> list = new CircularList<>();
         System.out.println("The size of our list: " + list.size());

         System.out.println("\nAdding 5, 7, and 13 to the list...");
         list.add(5);
         list.add(7);
         list.add(13);
         System.out.println("Current size of our list: " + list.size());

         System.out.println("\nRemoving 7 from the list...");
         list.remove(7);
         System.out.println("Current size of our list: " + list.size());

         System.out.println("\nRemoving the non-existing element 24 from the list");
         System.out.println("Should return FALSE = " + list.remove(24));
         System.out.println("Current size of our list: " + list.size());

         System.out.println("\nEXTRA TESTING - Is it circular?");
         System.out.println("\nAdding 1 to 10 to the list");
         for(int i = 1; i<11; i++)
               list.add(i);
         System.out.println("Current size of our list: " + list.size());
         System.out.println("Getting the -2th element (should be 9) = " + list.get(-2));
         System.out.println("Getting the 63th element (should be 1) = " + list.get(62));

   }
}
```

OUTPUT

```
The size of our list: 0

Adding 5, 7, and 13 to the list...
Current size of our list: 3

Removing 7 from the list...
Current size of our list: 2

Removing the non-existing element 24 from the list
Should return FALSE = false
Current size of our list: 2

EXTRA TESTING - Is it circular?
```

Adding 1 to 10 to the list
Current size of our list: 12
Getting the -2th element (should be 9) = 9
Getting the 63th element (should be 9) = 1