Deque.java:

```
package proj3;   // Gradescope needs this
/**
*  Class description, name, honor code go here.
* This class implements a deque using a doubly linked list.
* It provides methods to add and remove elements from both ends,
* check the size, and ensure capacity. It also includes methods
* to clone the deque, check equality, and trim capacity.
* @author Kaito Nguyen
* @version May 11, 2025, CSC 151, Spring 2025
* I confirm that I follow the Honor Code.
*/

public class Deque {
   public int DEFAULT_CAPACITY = 10;
   /**
    * Data is stored in a doubly linked list.
    * The capacity of the deque is the number of elements that can be stored in the
deque.
    * The size of the deque is the number of elements currently in the deque.
    */
   private DoublyLinkedList data; // the data in the deque                      //
   private int capacity;         // the current capacity of the deque          //
   private int size;            // the number of elements in the deque        //


   /**
    * Creates a new deque with initial capacity 10.
    */
   public Deque()
   {
       data = new DoublyLinkedList();
       capacity = DEFAULT_CAPACITY;
       size = 0;
   }

   /**
    * Creates a new deque.
    * @param initialCapacity the initial capacity of the deque.
    */
   public Deque(int initialCapacity)
   {
```

```
    if (initialCapacity < 0) {
        System.out.println("Invalid capacity");
        return;
    }
    data = new DoublyLinkedList();
    capacity = initialCapacity;
    size = 0;
}

/**
 * inserts a string on the left of the deque.
 *
 * If the deque's capacity has been reached, the deque will
 * expand to twice its current capacity plus 1.
 *
 * @param value the string to add.
 */
public void addLeft(String value)
{
    if (value == null) {
        System.out.println("Invalid value");
        return;
    }
    if (size == capacity) {
        ensureCapacity(2 * capacity + 1);
    }
    data.addFirst(value);
    size = data.size();
}

/**
 * inserts a string on the right of the deque.
 *
 * If the deque's capacity has been reached, the deque will
 * expand to twice its current capacity plus 1.
 *
 * @param value the string to add.
 */
public void addRight(String value)
{
    if (value == null) {
        System.out.println("Invalid value");
```

```
            return;
        }
        if (size == capacity) {
            ensureCapacity(2 * capacity + 1);
        }
        data.addLast(value);
        size = data.size();
    }

    /**
     * Remove and return the left item from the deque.  Return
     * null if the deque is empty.
     */
    public String removeLeft()
    {
        if(data.isEmpty()) {
            return null;
        }
        String value = data.removeFirst();
        size = data.size();
        return value;
    }

    /**
     * Remove and return the right item from the deque.  Return
     * null if the deque is empty.
     */
    public String removeRight()
    {
        if(data.isEmpty()) {
            return null;
        }
        String value = data.removeLast();
        size = data.size();
        return value;
    }

    /**
     * Places the contents of another deque in order at the right
     * end of this deque.  For example, if this deque is {A,B} and
     * the other deque is {B,C,D}, addAll will change this deque
     * to be {A,B,B,C,D}.
```

```
     *
     * If adding all elements of the other deque would exceed the
     * capacity of this deque, the capacity is changed to make
     * exactly enough room for all the elements to be added.
     *
     * Postcondition: NO SIDE EFFECTS!  the other deque should be left
     * unchanged.
     *
     * @param otherDeque the deque whose contents should be added.
     */
    public void addAll(Deque otherDeque)
    {
        if (otherDeque == null) {
            System.out.println("Invalid deque");
            return;
        }
        if (otherDeque.size() == 0) {
            return;
        }
        if (size + otherDeque.size() > capacity) {
            ensureCapacity(size + otherDeque.size());
        }
        ListNode otherCurrent = otherDeque.data.getFirstNode();
        while (otherCurrent != null) {
            data.addLast(otherCurrent.data);
            otherCurrent = otherCurrent.next;
        }
        size = data.size();
    }

    /**
     * Make a copy of this deque.  Changes to the copy
     * do not affect the current deque, and vice versa.
     *
     * Postcondition: NO SIDE EFFECTS!  This deque's leftmost
     * and rightmost elements should remain unchanged.
     *
     * @return the copy of this deque.
     */
    public Deque clone()
    {
        Deque copy = new Deque(capacity);
```

```
        ListNode current = data.getFirstNode();
        while (current != null) {
            copy.addRight(current.data);
            current = current.next;
        }
        return copy;
    }

    /**
     * Change the current capacity of this sequence.  The sequence's
     * capacity will be changed to be at least minCapacity.  Does
     * nothing if current capacity is already >= minCapacity.
     *
     * @param minCapacity the minimum capacity that the sequence
     * should now have.
     */
    public void ensureCapacity(int minCapacity)
    {
        if (minCapacity > capacity) {
            capacity = minCapacity;
        }
    }

    /**
     * @return the capacity of the sequence.
     */
    public int getCapacity()
    {
        return capacity;
    }

    /**
     * @return the leftmost element in the deque, or
     * null if deque is empty
     */
    public String leftMost()
    {
        if(data.isEmpty()) {
            return null;
        }
        return data.getFirstNode().data;
    }
```

```java
/**
 * @return the rightmost element in the deque, or
 * null if deque is empty
 */
public String rightMost()
{
    if(data.isEmpty()) {
        return null;
    }
    return data.getLastNode().data;
}



/**
 * @return the number of elements stored in the sequence.
 */
public int size()
{
    return size;
}



/**
 * Reduce the current capacity to its actual size, so that it has
 * capacity to store only the elements currently stored.
 */
public void trimToSize()
{
    if (size < capacity) {
        capacity = size;
    }
}

/**
 * Produce a left-to-right string representation of this deque.
 * For example, a deque of capacity 5 with "A" as its leftmost elmt,
 * "C" as its rightmost elmt, and "B" in the middle would print as:
 *
 *    {A, B, C} (capacity = 5)
 *
 * The string you create should be formatted like the above example,
```

```
     * with a comma and space following each element, no comma following the
     * last element, and all on a single line. An empty deque
     * should give back "{}" followed by its capacity.
     *
     * @return a string representation of this deque.
     */
    public String toString()
    {
        String result = "{";
        ListNode current = data.getFirstNode();
        while (current != null) {
            result += current.data;
            current = current.next;
            if (current != null) {
                result += ", ";
            }
        }
        result += "} (capacity = " + capacity + ")";
        return result;
    }

    /**
     * Checks whether another deque is equal to this one.  To be
     * considered equal, both deques must have exactly the same
     * elements in the same order.  The capacity can differ.
     *
     * NO SIDE EFFECTS!  Both this deque and the other deque
     * must be the same at the end of this method as they were
     * at the start of this method.
     *
     * @param other the other Deque with which to compare
     * @return true iff the other Deque is equal to this one.
     */
    public boolean equals(Deque other)
    {
        if (other == null) {
            return false;
        }
        if (size != other.size()) {
            return false;
        }
        ListNode current = data.getFirstNode();
```

```
        ListNode otherCurrent = other.data.getFirstNode();
        while (current != null && otherCurrent != null) {
            if (!current.data.equals(otherCurrent.data)) {
                return false;
            }
            current = current.next;
            otherCurrent = otherCurrent.next;
        }
        return true;
    }

    /**
     *
     * @return true if deque empty, else false
     */
    public boolean isEmpty()
    {
        return data.isEmpty();
    }

    /**
     *  empty the deque
     */
    public void clear()
    {
        data = new DoublyLinkedList();
        size = 0;
    }


}
```

==========
==========

DequeJavaTest.java

```
package proj3;
/**
* JUnit test class.  Use these tests as models for your own.
*/
```

```
import java.util.concurrent.TimeUnit;

import junit5.documentation.src.test.java.example.FastTest;

import java.beans.Transient;


public class DequeJavaTest {
   @Rule // a test will fail if it takes longer than 1/10 of a second to run
   public Timeout timeout = Timeout.millis(100);

   // Test the constructor
   // The constructor should create an empty deque with a default capacity of 10
   @Test
   public void testAddNull() {
       Deque d = new Deque();
       d.addLeft(null);
       assertEquals(0, d.size());
       assertEquals("{} (capacity = 10)", d.toString());
       d.addRight(null);
       assertEquals(0, d.size());
       assertEquals("{} (capacity = 10)", d.toString());
   }

   // Test removeLeft and removeRight
   // The removeLeft and removeRight methods should return null when the deque is
empty
   // and should return the correct element when the deque is not empty
   @Test
   public void testRemoveLeftOnEmptyRepeatedly() {
       Deque d = new Deque();
       assertNull(d.removeLeft());
       assertEquals(0, d.size());
       assertNull(d.removeLeft());
       assertEquals(0, d.size());
   }

   @Test
   public void testRemoveRightOnEmptyRepeatedly() {
       Deque d = new Deque();
       assertNull(d.removeRight());
```

```
        assertEquals(0, d.size());
        assertNull(d.removeRight());
        assertEquals(0, d.size());
    }

    // Test the ensureCapacity method
    // The ensureCapacity method should not change the size of the deque
    // and should not change the capacity of the deque if the new capacity is less than
the current capacity
    @Test
    public void testEnsureCapacity() {
        Deque d = new Deque(5);
        d.addRight("a");
        assertEquals(5, d.getCapacity());
        d.ensureCapacity(3);
        assertEquals(5, d.getCapacity());
    }

    // Test trimToSize
    // The trimToSize method should not change the size of the deque
    // and cuts the capacity of the deque to the current size
    @Test
    public void testTrimToSize() {
        Deque d = new Deque(8);
        d.trimToSize();
        assertEquals(8, d.getCapacity());
        Deque d2 = new Deque(10);
        d2.addLeft("a");
        assertEquals(8, d2.getCapacity());
    }

    // Test the toString method
    // The toString method should return a string representation of the deque
    @Test
    public void testToStringAddLeft() {
        Deque d = new Deque();
        d.addLeft("a");
        assertEquals("{a} (capacity = 10)", d.toString());
        d.addLeft("b");
        assertEquals("{b, a} (capacity = 10)", d.toString());
    }
```

```java
@Test
public void testToStringAddRight() {
    Deque d = new Deque();
    d.addRight("a");
    assertEquals("{a} (capacity = 10)", d.toString());
    d.addRight("b");
    assertEquals("{a, b} (capacity = 10)", d.toString());
}

// Test clone
// The clone method should return a copy of the deque
// and the copy should be independent of the original deque
@Test
public void testCloneIndependence() {
    Deque orin = new Deque();
    orin.addLeft("a");
    Deque clone = (Deque)orin.clone();
    assertEquals(orin.size(), clone.size());
    clone.addLeft("b");
    assertNotEquals(orin.size(), clone.size());
    assertEquals(orin.size(), 1);
    assertEquals(clone.size(), 2);
}

// Test the addAll method
// It should add all elements from the given collection to the deque
// and should not change the size of the deque if the collection is empty
@Test
public void testAddAllNullArgument() {
    Deque d = new Deque();
    d.addAll(null);
    assertEquals(0, d.size());
    assertEquals("{} (capacity = 10)", d.toString());
}

// Test clear
// The clear method should remove all elements from the deque
@Test
public void testClear() {
    Deque d = new Deque(5);
    d.addLeft("a");
    assertEquals(1, d.size());
```

```
        d.clear();
        assertEquals(0, d.size());
        assertEquals("{} (capacity = 5)", d.toString());
    }

    // Test equals
    // The equals method should return true if the two deques are equal
    @Test
    public void testEquals() {
        Deque d1 = new Deque();
        d1.addLeft("a");
        d1.addRight("b");
        Deque d2 = new Deque();
        d2.addLeft("a");
        d2.addRight("b");
        assertTrue(d1.equals(d2));
        assertFalse(d1.equals(null));
        assertFalse(d1.equals(new Object()));
    }

    // Test the size method
    // The size method should return the number of elements in the deque
    @Test
    public void testSize() {
        Deque d = new Deque();
        assertEquals(0, d.size());
        d.addLeft("a");
        assertEquals(1, d.size());
        d.addRight("b");
        assertEquals(2, d.size());
        d.removeLeft();
        assertEquals(1, d.size());
        d.removeRight();
        assertEquals(0, d.size());
    }

    // Test the isEmpty method
    // The isEmpty method should return true if the deque is empty
    // and false otherwise
    @Test
    public void testIsEmpty() {
        Deque d = new Deque();
```

```
        assertTrue(d.isEmpty());
        d.addLeft("a");
        assertFalse(d.isEmpty());
        d.removeLeft();
        assertTrue(d.isEmpty());
    }

    // Test the addLeft method
    // The addLeft method should add an element to the left end of the deque
    @Test
    public void testAddLeft() {
        Deque d = new Deque();
        d.addLeft("a");
        assertEquals(1, d.size());
        assertEquals("{a} (capacity = 10)", d.toString());
        d.addLeft("b");
        assertEquals(2, d.size());
        assertEquals("{b, a} (capacity = 10)", d.toString());
    }

    // Test the addRight method
    // The addRight method should add an element to the right end of the deque
    @Test
    public void testAddRight() {
        Deque d = new Deque();
        d.addRight("a");
        assertEquals(1, d.size());
        assertEquals("{a} (capacity = 10)", d.toString());
        d.addRight("b");
        assertEquals(2, d.size());
        assertEquals("{a, b} (capacity = 10)", d.toString());
    }

    // Test the removeLeft method
    // The removeLeft method should remove and return the leftmost element of the deque
    @Test
    public void testRemoveLeft() {
        Deque d = new Deque();
        d.addLeft("a");
        d.addLeft("b");
        assertEquals("b", d.removeLeft());
        assertEquals(1, d.size());
```

```
        assertEquals("{a} (capacity = 10)", d.toString());
        assertEquals("a", d.removeLeft());
        assertEquals(0, d.size());
        assertEquals("{} (capacity = 10)", d.toString());
    }
}
```

==========
==========

DoublyLinkedList.java

```
package proj3;  // Gradescope needs this.
/**
* The doubly linked list class gives you access to the beginning and end of a linked
* list through instance variables firstNode and lastNode.  This class
* should contain all the methods for general manipulation of linked lists:
* traversal, insertion, deletion, searching, etc.
*
* @author
* @version
*/
public class DoublyLinkedList
{
   private int length;          // number of nodes
   private ListNode firstNode;  // pointer to first node
   private ListNode lastNode;   // pointer to last node

   /**
    * Default constructor
    */
   public DoublyLinkedList() {
       length = 0;
       firstNode = null;
       lastNode = null;
   }

   /**
    * Returns the number of nodes
    * @return the number of nodes
    */
```

```java
public int size() {
    return length;
}

/**
 * Returns true if the list is empty, false otherwise.
 * @return true if the list is empty, false otherwise
 */
public boolean isEmpty() {
    return length == 0;
}

/**
 * Returns the first node in the list.
 * @return the first node in the list
 */
public ListNode getFirstNode() {
    return firstNode;
}

/**
 * Returns the last node in the list.
 * @return the last node in the list
 */
public ListNode getLastNode() {
    return lastNode;
}

/**
 * Adds a new node to the front of the list.
 * @param newData the data to be added
 */
public void addFirst(String newData) {
    ListNode newNode = new ListNode(newData);
    if (isEmpty()) {
        firstNode = newNode;
        lastNode = newNode;
    } else {
        newNode.next = firstNode;
        firstNode.prev = newNode;
        firstNode = newNode;
    }
```

```
        length++;
    }

    /**
     * Adds a new node to the end of the list.
     * @param newData the data to be added
     */
    public void addLast(String newData) {
        ListNode newNode = new ListNode(newData);
        if (isEmpty()) {
            firstNode = newNode;
            lastNode = newNode;
        } else {
            lastNode.next = newNode;
            newNode.prev = lastNode;
            lastNode = newNode;
        }
        length++;
    }

    /**
     * Removes the first node from the list.
     * @return the data of the removed node
     */
    public String removeFirst() {
        if (isEmpty()) {
            return null;
        }
        String data = firstNode.data;
        if (firstNode == lastNode) {
            firstNode = null;
            lastNode = null;
        } else {
            firstNode = firstNode.next;
            firstNode.prev = null;
        }
        length--;
        return data;
    }

    /**
     * Removes the last node from the list.
```

```java
    * @return the data of the removed node
    */
   public String removeLast() {
       if (isEmpty()) {
           return null;
       }
       String data = lastNode.data;
       if (firstNode == lastNode) {
           firstNode = null;
           lastNode = null;
       } else {
           lastNode = lastNode.prev;
           lastNode.next = null;
       }
       length--;
       return data;
   }

   /**
    * Returns a string of the list.
    * @return a string of the list
    */
   public String toString() {
       String result = "[";
       ListNode currentNode = firstNode;
       while (currentNode != null) {
           result += currentNode.data;
           currentNode = currentNode.next;
           if (currentNode != null) {
               result += ", ";
           }
       }
       result += "]";
       return result;
   }
}
```