Deque.java:

```java
package proj2;
/**
*  Class description, name, and honor code go here.
*  I'd fill this in if I were you.
*  @author Kaito Nguyen
*  @version Apr 22 2025, CSC 151, Spring 2025
*  I confirm that I follow honor code
*/

public class Deque {

   /**
    * invariant goes here
    */
   private String[] data; // the array that holds the deque's elements
   private int left; // the index of the leftmost element
   private int right; // the index of the rightmost element
   private int size; // the number of elements in the deque
   private int capacity = 10; // the capacity of the deque

   /**
    * Creates a new deque with initial capacity 10.
    */
   public Deque()
   {
       data = new String[capacity]; // initialize the array
       left = 0; // initialize the left index
       right = 0; // initialize the right index
       size = 0; // initialize the size of the deque
   }

   /**
    * 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; // invalid capacity, do not create the deque
        }
        data = new String[initialCapacity];
        left = 0;
        right = 0;
        size = 0;
        capacity = initialCapacity;
    }

    private Deque makeAcloneWithNewCapacity(int newCapacity) {
        Deque newDeque = new Deque(newCapacity); // create a new deque with
the new capacity
        for (int i = 0; i < size; i++) {
            newDeque.data[i] = data[Math.floorMod(left + i, capacity)];
        }
        newDeque.left = 0;
        if (size > 0) {
            newDeque.right = size - 1; // set the right index to the last
element
        } else {
            newDeque.right = 0; // set the right index to 0 if the deque is
empty
        }
        newDeque.capacity = newCapacity; // set the capacity of the new
deque
        newDeque.size = size;
        return newDeque; // return the new deque
    }

    /**
     * inserts a string on the left of the deque.
     *
     * If this cannot be done because the deque has
     * reached its capacity, the deque will
     * expand to twice its current capacity plus 1
     * to accommodate the new entry.
     *
     * @param value the string to add.
     */
```

```
  private void growCapacity() {
      if (size == capacity) {
          Deque newDeque = makeAcloneWithNewCapacity(capacity * 2 + 1);
// create a new deque with the new capacity
          if (newDeque == null) {
              return; // invalid capacity, do not create the deque
          }
          data = newDeque.data;
          left = newDeque.left;
          right = newDeque.right;
          capacity = newDeque.capacity;
          size = newDeque.size;
      }
  }

  public void addLeft(String value)
  {
      if (value == null) {  // check if value is null, then do nothing
          return;
      }
      growCapacity();
      left = Math.floorMod(left - 1, capacity);
      data[left] = value;
      if (size == 0) right = left;
      size++;
  }

  /**
   * inserts a string on the right of the deque.
   *
   * If this cannot be done because the deque has
   * reached its capacity, the deque will
   * expand to twice its current capacity plus 1
   * to accommodate the new entry.
   *
   * @param value the string to add.
   */
  public void addRight(String value)
  {
```

```
    if (value == null) {   // check if value is null, then do nothing
        return;
    }
    growCapacity();
    right = Math.floorMod(right + 1, capacity);
    data[right] = value;
    if (size == 0) left = right;
    size++;
}

/**
 * Remove and return the left item from the deque.  Return
 * null if the deque is empty.
 */
public String removeLeft()
{
    if (isEmpty()) {
        return null;
    }
    String value = data[left];
    data[left] = null;
    if (size == 1) {
        left = 0;
        right = 0;
    } else {
        left = Math.floorMod(left + 1, capacity);
    }
    size--;
    return value;
}

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

```
        String value = data[right];
        data[right] = null;
        if (size == 1) {
            left = 0;
            right = 0;
        } else {
            right = Math.floorMod(right - 1, capacity);
        }
        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 || otherDeque.isEmpty()) {
            return;
        }
        int otherSize = otherDeque.size(); // get the size of the other
deque
        if (otherSize == 0) {
            return; // do nothing if the other deque is empty
        }
        ensureCapacity(size + otherSize);
        for (int i = 0; i < otherSize; i++) {
```

```
            addRight(otherDeque.data[Math.floorMod(otherDeque.left + i,
otherDeque.capacity)]);
        }
    }

    /**
     * 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 newDeque = makeAcloneWithNewCapacity(capacity); // create a
new deque with the same capacity
        if (newDeque == null) {
            return null; // invalid capacity, do not create the deque
        }
        return newDeque; // return the new deque
    }

    /**
     * Change the capacity of this deque to minCapacity
     * if it doesn't already have that much capacity. Does
     * nothing if current capacity is already >= minCapacity.
     *
     * @param minCapacity the minimum capacity that the deque
     * should now have.
     */
    public void ensureCapacity(int minCapacity)
    {
        int newCapacity = Math.max(capacity, minCapacity);
        if (newCapacity > capacity) {
            Deque newDeque = makeAcloneWithNewCapacity(newCapacity); //
create a new deque with the new capacity
            if (newDeque == null) {
                return; // invalid capacity, do not create the deque
```

```
            }
            data = newDeque.data;
            left = newDeque.left;
            right = newDeque.right;
            capacity = newDeque.capacity;
            size = newDeque.size;
        }
    }

    /**
     * Getter for the amount of data this deque can potentially hold.
     *
     * @return the capacity of the deque.
     */
    public int getCapacity()
    {
        return capacity;
    }

    /**
     * Returns the leftmost element in the deque without
     * altering the deque itself.
     *
     * @return the leftmost element in the deque, or
     * null if deque is empty
     */
    public String leftMost()
    {
        if (isEmpty()) {
            return null;
        }
        return data[left];
    }

    /**
     * Returns the rightmost element in the deque without
     * altering the deque itself.
     *
     * @return the rightmost element in the deque, or
     * null if deque is empty
```

```
     */
    public String rightMost()
    {
        if (isEmpty()) {
            return null;
        }
        return data[right];
    }


    /**
     * Getter for the amount of data this deque currently holds.
     *
     * @return the number of elements stored in the deque.
     */
    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) {
            Deque newDeque = makeAcloneWithNewCapacity(size); // create a
new deque with the new capacity
            if (newDeque == null) {
                return; // invalid capacity, do not create the deque
            }
            data = newDeque.data;
            left = newDeque.left;
            right = newDeque.right;
            capacity = newDeque.capacity;
            size = newDeque.size;
        }
    }
```

```java
   /**
    * Produce a left-to-right string representation of this deque.
    * For example, a deque of three elmts with capacity 5 with "A" as its
leftmost elmt,
    * "C" as its rightmost elmt, and "B" in the middle would produce this
string:
    *
    *    {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 = "{";
       for (int i = 0; i < size; i++) {
           result += data[Math.floorMod(left + i, capacity)];
           if (i < size - 1) {
               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. Else false.
     */
    public boolean equals(Deque other)
    {
        if (other == null) {
            return false;
        }
        if (size != other.size()) {
            return false;
        }
        for (int i = 0; i < size; i++) {
            if (!data[Math.floorMod(left + i,
capacity)].equals(other.data[Math.floorMod(other.left + i,
other.capacity)])) {
                return false;
            }
        }
        return true;
    }

    /**
     *
     * @return true if deque empty, else false
     */
    public boolean isEmpty()
    {
        return size == 0;
    }

    /**
     *  empty the deque
     */
    public void clear()
    {
        for (int i = 0; i < capacity; i++) {
            data[i] = null; // set all elements to null
        }
        // reset the left and right indices and size
        left = 0;
```

```
        right = 0;
        size = 0;
    }
}
```

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

DequeJavaTest.java

```
package proj2;
/**
* 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.size());
        d.ensureCapacity(3);
        assertEquals(5, d.size());
    }

    // 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(0, d.size());
        Deque d2 = new Deque(10);
        d2.addLeft("a");
        assertEquals(1, d2.size());
    }

    // 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());
    }

    @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());
    }
}
```