```java
package proj2;
/**
 * Class modeling a Sequence Abstract Data Type (ADT)
 *
 * Invariant:
 * 1. Number of elements in the sequence is stored in the instance variable manyItems
 * 2. For an empty sequence, we don't care what is stored in data. For a nonempty sequence, the elements of the sequence
 *    are stored from the front to the end in data[0] to data[manyItems-1], we don't care what's stored in the rest of
 *    data
 * 3. If there is a current element, then it lies in data[currentIndex]; if there is no current element, then
 *    currentIndex equals manyItems.
 *
 * @author Neil Daterao
 * @version 01/29/24
 * @Note I affirm that I have carried out the attached academic endeavors with full academic honesty, in
 * accordance with the Union College Honor Code and the course syllabus.
 *
 */


public class Sequence
{
    private String[] data;
    private int manyItems;
    private int currentIndex;



    //Constants to avoid magic numbers
    private final int DEFAULTCAPACITY = 10;
```

```java
    private final int INITIALAMOUNTOFITEMS = 0;

    /**
     * Creates a new sequence with initial capacity 10.
     */
    public Sequence() {
        try { data = new String[DEFAULTCAPACITY]; }
        catch (OutOfMemoryError memoryError){
            throw new OutOfMemoryError("Indicates insufficient memory for new Sequence");
        }
        manyItems = INITIALAMOUNTOFITEMS;
        noCurrentElement();
    }


    /**
     * Creates a new sequence.
     *
     * @param initialCapacity the initial capacity of the sequence.
     */
    public Sequence(int initialCapacity){
        if (initialCapacity < 0) {
            initialCapacity = DEFAULTCAPACITY; //in the event of a negative input
        }
        try { data = new String[initialCapacity]; }
        catch (OutOfMemoryError memoryError){
            throw new OutOfMemoryError("Indicates insufficient memory for new Sequence");
        }
        manyItems = INITIALAMOUNTOFITEMS;
        noCurrentElement();


    }


    /**
```

```
     * Adds a string to the sequence in the location before the
     * current element. If the sequence has no current element, the
     * string is added to the beginning of the sequence.
     *
     * The added element becomes the current element.
     *
     * If the sequences's capacity has been reached, the sequence will
     * expand to twice its current capacity plus 1.
     *
     * @param value the string to add.
     */
    public void addBefore(String value)
    {
        if (getCapacity() <= size()) { ensureCapacity(2*getCapacity() + 1); }
        if (isCurrent()) {
            shiftRightFromIndex(getCurrentIndex());
            data[getCurrentIndex()] = value;
            increaseItemCount();
        }
        else {
            int startingIndex = 0;
            shiftRightFromIndex(startingIndex);
            data[startingIndex] = value; //added to beginning of sequence
            increaseItemCount();
            setCurrentIndex(startingIndex); //added element becomes current element
        }


    }


    /**
     * Adds a string to the sequence in the
```

```
location after the current
     * element. If the sequence has no current element, the string is
     * added to the end of the sequence.
     *
     * The added element becomes the current element.
     *
     * If the sequences's capacity has been reached, the sequence will
     * expand to twice its current capacity plus 1.
     *
     * @param value the string to add.
     */
    public void addAfter(String value)
    {
        if (getCapacity() <= size()) { ensureCapacity(2*getCapacity() + 1); }
        if (isCurrent()) {
            int indexAfterCurrent = getCurrentIndex() + 1;
            shiftRightFromIndex(indexAfterCurrent);
            data[indexAfterCurrent] = value;
            increaseItemCount();
            advance(); //added element becomes the current element
        }
        else {
            int endOfSequenceIndex = size(); // Sequence contents is from data[0] to data[manyItems-1], next available is data[manyItems].
            data[endOfSequenceIndex] = value;
            increaseItemCount();
            setCurrentIndex(endOfSequenceIndex); // added element becomes current element.

        }

    }

    /**
```

```
     * Private helper method to shift elements in sequence right from a given index
     * @param indexToShiftFrom
     */
    private void shiftRightFromIndex(int indexToShiftFrom) {
        for (int index = size(); index > indexToShiftFrom; index--) {
            data[index] = data[index - 1];
        }
    }


    /**
     * @return true if and only if the sequence has a current element.
     */
    public boolean isCurrent()
    {
        return (getCurrentIndex()< size());
    }


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


    /**
     * @return the element at the current location in the sequence, or
     * null if there is no current element.
     */
    public String getCurrent()
    {
        if (isCurrent()) { return data[getCurrentIndex()]; }
```

```
        else { return null; }
    }


    /**
     * Increase the sequence's capacity 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 (getCapacity() < minCapacity) {
            String[] newArray = new String[minCapacity];
            duplicateElementsToNewArray(newArray);
            data = newArray;
        }

    }


    /**
     * Places the contents of another sequence at the end of this sequence.
     *
     * If adding all elements of the other sequence would exceed the
     * capacity of this sequence, the capacity is changed to make room for
     * all of the elements to be added.
     *
     * Postcondition: NO SIDE EFFECTS!  the other sequence should be left
     * unchanged.  The current element of both sequences should remain
     * where they are. (When this method ends, the
```

```
current element
     * should refer to the same element that it did at the time this method
     * started.)
     *
     * @param another the sequence whose contents should be added.
     */
    public void addAll(Sequence another)
    {
        ensureCapacity(size() + another.size());
        Sequence clonedAnother = another.clone();

        int currentIndexHolder = getCurrentIndex();
        boolean noCurrentBeforeAddAll = false;
        if (!isCurrent())  { noCurrentBeforeAddAll = true; }
        setCurrentIndex(size()-1);

        clonedAnother.start();
        for (int count = 0; count < clonedAnother.size(); count++) {
            addAfter(clonedAnother.getCurrent());
            clonedAnother.advance();
        }
        if (noCurrentBeforeAddAll) { noCurrentElement(); } //reset current index to prior to method call
        else { setCurrentIndex(currentIndexHolder); } //reset current index to prior to method call

    }

    /**
     * Move forward in the sequence so that the current element is now
     * the next element in the sequence.
     *
     * If the current element was already the end of the sequence,
     * then advancing causes there to be no current
```

```
 element.
     *
     * If there is no current element to begin with, do nothing.
     */
    public void advance()
    {
        if (isCurrent()) {
            //we know based on invariant that last element is data[manyItems-1]
            //thus size()-2 ensures we don't pass the last element
            if (getCurrentIndex() < size() - 1 ) { currentIndex++; }
            else { noCurrentElement(); } //if no current element, currentIndex = manyItems, based on invariant.
        }
    }


    /**
     * Make a copy of this sequence.  Subsequence changes to the copy
     * do not affect the current sequence, and vice versa.
     *
     * Postcondition: NO SIDE EFFECTS!  This sequence's current
     * element should remain unchanged.  The clone's current
     * element will correspond to the same place as in the original.
     *
     * @return the copy of this sequence.
     */
    public Sequence clone()
    {
        try {
            Sequence clonedSequence = new Sequence(getCapacity());
```

```
            //add all elements from sequence to cloned sequence
            for (int index = 0; index < size(); index++) {
                clonedSequence.addAfter(data[index]);
            }

            //set currentIndex's equal to each other.
            clonedSequence.start();
            for (int count = 0; count < getCurrentIndex(); count++) { clonedSequence.advance(); }

            return clonedSequence;

        }
        catch (OutOfMemoryError memoryError){
            throw new OutOfMemoryError("Indicates insufficient memory for Cloned Sequence");
        }

    }


    /**
     * Remove the current element from this sequence.  The following
     * element, if there was one, becomes the current element.  If
     * there was no following element (current was at the end of the
     * sequence), the sequence now has no current element.
     *
     * If there is no current element, does nothing.
     */
    public void removeCurrent()
    {
```

```
        if (isCurrent()) {
            String[] newArray = new String[getCapacity()];
            int newArrayIndex = 0;
            for (int index = 0; index < size(); index++) {
                if (index != getCurrentIndex()) { newArray[newArrayIndex++] = data[index]; }
            }
            decreaseItemCount();
            data = newArray;

        }

    }


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


    /**
     * Sets the current element to the start of the sequence.  If the
     * sequence is empty, the sequence has no current element.
     */
    public void start()
    {
        if (!isEmpty()) { setCurrentIndex(0); }
        else { noCurrentElement(); }
    }


    /**
```

```
     * Reduce the current capacity to its actual size, so that it has
     * capacity to store only the elements currently stored.
     */
    public void trimToSize()
    {
        try {
            String[] newArray = new String[size()];
            duplicateElementsToNewArray(newArray);
            data = newArray;
        }
        catch (OutOfMemoryError memoryError){
            throw new OutOfMemoryError("Indicates insufficient memory for trim");
        }

    }


    /**
     * Produce a string representation of this sequence.  The current
     * location is indicated by a >.  For example, a sequence with "A"
     * followed by "B", where "B" is the current element, and the
     * capacity is 5, would print as:
     *
     *    {A, >B} (capacity = 5)
     *
     * The string you create should be formatted like the above example,
     * with a comma following each element, no comma following the
     * last element, and all on a single line.  An empty sequence
     * should give back "{}" followed by its capacity.
     *
     * @return a string representation of this
```

```
sequence.
     */
    public String toString()
    {
        StringBuilder sb = new StringBuilder();

        sb.append("{");

        for (int index = 0; index < size(); index ++) {
            // Add the current element with a ">" if it matches the current index
            if (index == getCurrentIndex()) {
                sb.append(">").append(data[index]);
            } else {
                sb.append(data[index]);
            }

            // Add a comma if it's not the last element
            if (index < size() - 1) {
                sb.append(", ");
            }
        }

        sb.append("} (capacity = ").append(getCapacity()).append(")");

        return sb.toString();

    }

    /**
     * Checks whether another sequence is equal to this one.  To be
     * considered equal, the other sequence must have the same size
     * as this sequence, have the same elements, in the same
     * order, and with the same element marked
     * current.  The capacity can differ.
```

```
     *
     * Postcondition: NO SIDE EFFECTS!  this sequence and the
     * other sequence should remain unchanged, including the
     * current element.
     *
     * @param other the other Sequence with which to compare
     * @return true iff the other sequence is equal to this one.
     */
    public boolean equals(Sequence other)
    {
        if (this == other) { return true; } //see if objects are the same
        if (other == null) { return false; }

        //see if size and currentIndex are equal before running through each sequence
        if (size() == other.size() && getCurrentIndex() == other.getCurrentIndex() ) {
            Sequence clonedOther = other.clone();
            clonedOther.start();

            for (int index = 0; index < size(); index++) {

                if (!data[index].equals(clonedOther.getCurrent())) { return false; }
                else { clonedOther.advance(); }
            }
            return true;

        }

        else { return false; }
    }


    /**
```

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


    /**
     *  empty the sequence.  There should be no current element.
     */
    public void clear()
    {
        String[] emptyArray = new String[getCapacity()];
        data = emptyArray;
        clearItemCount();
        noCurrentElement(); //no current element.
    }

    /**
     * Private helper method to have no current element in sequence
     */
    private void noCurrentElement() {
        currentIndex = manyItems;
    }

    /**
     *
     * Private helper method to get current index.
     */
    private int getCurrentIndex() {
        return currentIndex;
    }

    /**
     * Private helper method to set current index
     * @param newIndex
```

```
     */
    private void setCurrentIndex(int newIndex) {
        currentIndex = newIndex;
    }

    /**
     * Private helper method to count of elements.
     */
    private void increaseItemCount(){
        manyItems++;
    }

    /**
     * Private helper method to count of elements.
     */
    private void decreaseItemCount(){
        manyItems--;
    }

    /**
     * Private helper method to duplicate elements to a new given array
     */
    private void duplicateElementsToNewArray(String[] array) {
        for (int index = 0; index < size(); index++) {
            array[index] = data[index];
        }
    }

    /**
     * Private helper method to clear many item count.
     */
    private void clearItemCount() {
        manyItems = 0;
    }
```

```
}
```

```java
package proj2;

import org.junit.*;
import org.junit.rules.Timeout;
import static org.junit.Assert.*;


/**
 * Testing class for Sequence ADT. Includes 66 JUnit tests, multiple tests for each method to ensure functionality
 * despite various edge cases.
 *
 * @author Neil Daterao
 */



public class SequenceTest {


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

    @Test
    public void testDefaultConstructor() {
        // Test the constructor
        Sequence sequence = new Sequence();

        // Verify that the sequence is empty
        assertTrue(sequence.isEmpty());

        // Verify that the initial capacity is 10
        assertEquals(10, sequence.getCapacity());
    }

    @Test
    public void testSecondConstructorValidInitialCapacity() {
        // Test the second constructor with a valid
```

```
initial capacity
        int validInitialCapacity = 15;
        Sequence sequence = new Sequence(validInitialCapacity);

        // Verify that the sequence is empty
        assertTrue(sequence.isEmpty());

        // Verify that the initial capacity is set correctly
        assertEquals(validInitialCapacity, sequence.getCapacity());
    }

    @Test
    public void testSecondConstructorNegativeInitialCapacity() {
        // Test the second constructor with a negative initial capacity, should revert to default.
        int negativeInitialCapacity = -5;
        int defaultCapacity = 10;
        Sequence sequence = new Sequence(negativeInitialCapacity);

        // Verify that the sequence is empty
        assertTrue(sequence.isEmpty());
        // Verify that the initial capacity is set correctly
        assertEquals(defaultCapacity, sequence.getCapacity());

    }

    @Test
    public void testAddBeforeEmptySequence() {
        // Test adding to an empty sequence
        Sequence sequence = new Sequence();
        sequence.addBefore("A");

        // Verify that the sequence is not empty
        assertFalse(sequence.isEmpty());
```

```java

        // Verify that the current element is "A"
        assertEquals("A", sequence.getCurrent());

        // Verify that the capacity has not changed
        assertEquals(10, sequence.getCapacity());
    }

    @Test
    public void testAddBeforeWithCurrentElement() {
        // Test adding with a current element present
        Sequence sequence = new Sequence();
        sequence.addAfter("A");
        sequence.addBefore("B");

        // Verify that the sequence is not empty
        assertFalse(sequence.isEmpty());

        // Verify that the current element is "B"
        assertEquals("B", sequence.getCurrent());

        // Verify that the capacity has not changed
        assertEquals(10, sequence.getCapacity());
        assertEquals("{>B, A} (capacity = 10)", sequence.toString());
    }

    @Test
    public void testAddBeforeWithCapacityExpansion() {
        // Test adding with capacity expansion
        Sequence sequence = new Sequence();
        for (int i = 0; i < 10; i++) {
            sequence.addAfter(String.valueOf(i));
        }
        sequence.addBefore("X");  // This should trigger capacity expansion

        // Verify that the sequence is not empty
        assertFalse(sequence.isEmpty());
```

```java

        // Verify that the current element is "X"
        assertEquals("X", sequence.getCurrent());

        // Verify that the capacity has been expanded
        assertEquals(21, sequence.getCapacity()); // Twice the current capacity plus 1

        assertEquals("{0, 1, 2, 3, 4, 5, 6, 7, 8, >X, 9} (capacity = 21)", sequence.toString());
    }

    @Test
    public void testAddAfterEmptySequence() {
        // Test adding to an empty sequence
        Sequence sequence = new Sequence();
        sequence.addAfter("A");

        // Verify that the sequence is not empty
        assertFalse(sequence.isEmpty());

        // Verify that the current element is "A"
        assertEquals("A", sequence.getCurrent());

        // Verify that the capacity has not changed
        assertEquals(10, sequence.getCapacity());
    }

    @Test
    public void testAddAfterWithCurrentElement() {
        // Test adding with a current element present
        Sequence sequence = new Sequence();
        sequence.addBefore("A");
        sequence.addAfter("B");

        // Verify that the sequence is not empty
        assertFalse(sequence.isEmpty());

        // Verify that the current element is "B"
```

```java
        assertEquals("B", sequence.getCurrent());

        // Verify that the capacity has not changed
        assertEquals(10, sequence.getCapacity());
    }

    @Test
    public void testAddAfterWithCapacityExpansion() {
        // Test adding with capacity expansion
        Sequence sequence = new Sequence();
        for (int i = 0; i < 10; i++) {
            sequence.addBefore(String.valueOf(i));
        }
        sequence.addAfter("X");  // This should trigger capacity expansion

        // Verify that the sequence is not empty
        assertFalse(sequence.isEmpty());

        // Verify that the current element is "X"
        assertEquals("X", sequence.getCurrent());

        // Verify that the capacity has been expanded
        assertEquals(21, sequence.getCapacity()); // Twice the current capacity plus 1
    }

    @Test
    public void testIsCurrentEmptySequence() {
        // Test an empty sequence
        Sequence sequence = new Sequence();
        assertFalse(sequence.isCurrent());
    }

    @Test
    public void testIsCurrentWithCurrentElement() {
        // Test with a current element present
        Sequence sequence = new Sequence();
        sequence.addBefore("A");
```

```
        assertTrue(sequence.isCurrent());
    }

    @Test
    public void testIsCurrentWithoutCurrentElement() {
        // Test without a current element
        Sequence sequence = new Sequence();
        sequence.addBefore("A");
        sequence.advance(); // Move to the end
        assertFalse(sequence.isCurrent());
        sequence.addAfter("B");
        sequence.advance();
        assertFalse(sequence.isCurrent());
    }

    @Test
    public void testGetCapacityEmptySequence() {
        // Test the capacity of an empty sequence
        Sequence sequence = new Sequence();
        assertEquals(10, sequence.getCapacity()); // Default capacity is 10
    }
    @Test
    public void testGetCapacityEmptySequenceSecondConstructor() {
        // Test the capacity of an empty sequence
        Sequence sequence = new Sequence(48);
        assertEquals(48, sequence.getCapacity()); // Default capacity is 10
    }

    @Test
    public void testGetCapacityAfterAddingElements() {
        // Test the capacity after adding elements
        Sequence sequence = new Sequence();
        sequence.addBefore("A");
        sequence.addAfter("B");
        assertEquals(10, sequence.getCapacity()); // Capacity should not change with small number of
```

```
elements
    }

    @Test
    public void testGetCapacityAfterCapacityExpansion() {
        // Test the capacity after triggering capacity expansion
        Sequence sequence = new Sequence();
        for (int i = 0; i < 15; i++) {
            sequence.addBefore(String.valueOf(i));
        }
        assertEquals(21, sequence.getCapacity()); // Assuming the capacity expands to 2 * current capacity + 1
    }

    @Test
    public void testGetCurrentEmptySequence() {
        // Test getting the current element from an empty sequence
        Sequence sequence = new Sequence();
        assertNull(sequence.getCurrent());
    }

    @Test
    public void testGetCurrentWithCurrentElement() {
        // Test getting the current element with a current element present
        Sequence sequence = new Sequence();
        sequence.addBefore("A");
        assertEquals("A", sequence.getCurrent());
    }

    @Test
    public void testGetCurrentWithoutCurrentElement() {
        // Test getting the current element without a current element
        Sequence sequence = new Sequence();
```

```java
        sequence.addBefore("A");
        sequence.advance(); // Move to the end
        assertNull(sequence.getCurrent());
    }

    @Test
    public void testEnsureCapacityNoExpansion() {
        // Test ensureCapacity with no expansion needed
        Sequence sequence = new Sequence();
        sequence.ensureCapacity(9);
        assertEquals(10, sequence.getCapacity()); // Capacity should remain unchanged
    }

    @Test
    public void testEnsureCapacityWithExpansion() {
        // Test ensureCapacity with capacity expansion needed
        Sequence sequence = new Sequence();
        sequence.ensureCapacity(150);
        assertEquals(150, sequence.getCapacity());

    }

    @Test
    public void testEnsureCapacityEqualCapacity() {
        // Test ensureCapacity with equal capacity
        Sequence sequence = new Sequence();
        sequence.ensureCapacity(10);
        assertEquals(10, sequence.getCapacity()); // Capacity should remain unchanged
    }

    @Test
    public void testAddAllEmptySequences() {
        // Test adding two empty sequences
        Sequence sequence1 = new Sequence();
        Sequence sequence2 = new Sequence();
        sequence1.addAll(sequence2);

```

```
        // Verify using toString
        assertEquals("{} (capacity = 10)", sequence1.toString());
    }

    @Test
    public void testAddAllEmptyToNonEmpty() {
        // Test adding an empty sequence to a non-empty sequence
        Sequence sequence1 = new Sequence();
        sequence1.addBefore("A");
        Sequence sequence2 = new Sequence();
        sequence1.addAll(sequence2);

        // Verify using toString
        assertEquals("{>A} (capacity = 10)", sequence1.toString());
    }

    @Test
    public void testAddAllNonEmptyToEmpty() {
        // Test adding a non-empty sequence to an empty sequence
        Sequence sequence1 = new Sequence();
        Sequence sequence2 = new Sequence();
        sequence2.addBefore("B");
        sequence2.addAfter("C");
        sequence1.addAll(sequence2);

        // Verify using toString
        assertEquals("{B, C} (capacity = 10)", sequence1.toString());
    }

    @Test
    public void testAddAllCurrentElementPreservation() {
        // Test current element preservation
        Sequence sequence1 = new Sequence();
        sequence1.addBefore("A");
        sequence1.addAfter("B");
```

```
        Sequence sequence2 = new Sequence();
        sequence2.addBefore("X");
        sequence2.addAfter("Y");
        sequence1.addAll(sequence2);

        // Verify using toString
        assertEquals("{A, >B, X, Y} (capacity = 10)", sequence1.toString());
    }

    @Test
    public void testAddAllCapacityExpansion() {
        // Test capacity expansion
        Sequence sequence1 = new Sequence();
        sequence1.addBefore("A");
        sequence1.addAfter("B");
        Sequence sequence2 = new Sequence();
        for (int i = 0; i < 15; i++) {
            sequence2.addBefore(String.valueOf(i));
        }
        sequence1.addAll(sequence2);

        // Verify using toString
        assertEquals("{A, >B, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0} (capacity = 17)", sequence1.toString());
    }

    @Test(expected = NullPointerException.class)
    public void testAddAllNullParameter() {
        // Test adding a null sequence, expecting NullPointerException
        Sequence sequence1 = new Sequence();
        sequence1.addAll(null);
    }

    @Test
    public void testAddAllAddCurrentElementToNonEmpty() {
        // Test adding a sequence with a current element to a non-empty sequence
```

```java
        Sequence sequence1 = new Sequence(3);
        sequence1.addBefore("C");
        sequence1.addAfter("D");
        sequence1.addAfter("E");

        Sequence sequence2 = new Sequence(2);
        sequence2.addBefore("A");
        sequence2.addAfter("B");

        sequence2.addAll(sequence1);

        // Verify using toString
        assertEquals("{A, >B, C, D, E} (capacity = 5)", sequence2.toString());
        assertEquals(5, sequence2.size());
        assertEquals(5, sequence2.getCapacity());
    }

    @Test
    public void testAdvanceEmptySequence() {
        // Test advancing in an empty sequence
        Sequence sequence = new Sequence();
        sequence.advance();
        assertNull(sequence.getCurrent());
    }

    @Test
    public void testAdvanceWithCurrentElement() {
        // Test advancing with a current element
        Sequence sequence = new Sequence();
        sequence.addBefore("A");
        sequence.addAfter("B");
        sequence.start();
        sequence.advance();
        assertEquals("B", sequence.getCurrent());
    }

    @Test
    public void testAdvanceAtEndNoCurrentElement() {
        // Test advancing at the end, resulting in
```

```
no current element
        Sequence sequence = new Sequence();
        sequence.addBefore("A");
        sequence.advance();
        sequence.advance();
        assertNull(sequence.getCurrent());
    }

    @Test
    public void testAdvanceNoCurrentElement() {
        // Test advancing with no current element to begin with
        Sequence sequence = new Sequence();
        sequence.advance();
        assertNull(sequence.getCurrent());
    }

    @Test
    public void testCloneEmptySequence() {
        // Test cloning an empty sequence
        Sequence original = new Sequence();
        Sequence cloned = original.clone();

        // Verify that the clone is not the same object
        assertNotSame(original, cloned);

        // Verify that modifications to one sequence do not affect the other
        original.addBefore("A");
        assertNull(cloned.getCurrent());
    }

    @Test
    public void testCloneNonEmptySequence() {
        // Test cloning a non-empty sequence
        Sequence original = new Sequence();
        original.addBefore("A");
        original.addAfter("B");
        Sequence cloned = original.clone();

```

```
        // Verify that the clone is not the same object
        assertNotSame(original, cloned);

        // Verify that modifications to one sequence do not affect the other
        original.advance();
        assertNull(original.getCurrent());
        assertEquals("B", cloned.getCurrent());
    }

    @Test
    public void testCloneWithNullElements() {
        // Test cloning a sequence with null elements
        Sequence original = new Sequence();
        original.addBefore(null);
        original.addAfter(null);
        Sequence cloned = original.clone();

        // Verify that the clone is not the same object
        assertNotSame(original, cloned);

        // Verify that modifications to one sequence do not affect the other
        original.advance();
        assertNull(cloned.getCurrent());
    }

    @Test
    public void testRemoveCurrentWithCurrentElement() {
        // Test removing the current element
        Sequence sequence = new Sequence();
        sequence.addBefore("A");
        sequence.addAfter("B");
        sequence.removeCurrent();

        // Verify using toString
        assertEquals("{A} (capacity = 10)",
```

```
sequence.toString());

    }

    @Test
    public void testRemoveCurrentAtEndNoCurrentElement() {
        // Test removing at the end, resulting in no current element
        Sequence sequence = new Sequence();
        sequence.addBefore("A");
        sequence.advance();  // Set current to "A"
        sequence.removeCurrent();
        assertNull(sequence.getCurrent());
    }

    @Test
    public void testRemoveCurrentEmptySequence() {
        // Test removing with no current element to begin with
        Sequence sequence = new Sequence();
        sequence.removeCurrent();
        assertNull(sequence.getCurrent());
    }

    @Test
    public void testRemoveCurrentWithNoCurrentElementNonEmptySequence() {
        // Test removing a current element that is null
        Sequence sequence = new Sequence();
        sequence.addBefore("C");
        sequence.addAfter("B");
        sequence.advance();  // Set current to "B"
        sequence.removeCurrent();

        // Verify using toString
        assertEquals("{C, B} (capacity = 10)", sequence.toString());
        assertNull(sequence.getCurrent());
```

```
    }

    @Test
    public void testSizeEmptySequence() {
        // Test size of an empty sequence
        Sequence sequence = new Sequence();
        assertEquals(0, sequence.size());
    }

    @Test
    public void testSizeNonEmptySequence() {
        // Test size of a non-empty sequence
        Sequence sequence = new Sequence();
        sequence.addBefore("A");
        sequence.addAfter("B");
        sequence.addBefore("C");
        assertEquals(3, sequence.size());
    }

    @Test
    public void testSizeAfterRemoveCurrent() {
        // Test size after removing the current element
        Sequence sequence = new Sequence();
        sequence.addBefore("A");
        sequence.addAfter("B");
        sequence.removeCurrent();
        assertEquals(1, sequence.size());
    }

    @Test
    public void testSizeAfterAddAll() {
        // Test size after adding another sequence
        Sequence sequence1 = new Sequence();
        sequence1.addBefore("A");
        sequence1.addAfter("B");

        Sequence sequence2 = new Sequence();
        sequence2.addBefore("C");
        sequence2.addAfter("D");

```

```java
        sequence1.addAll(sequence2);
        assertEquals(4, sequence1.size());
    }


    @Test
    public void testStartWithNonEmptySequence() {
        // Test starting with a non-empty sequence
        Sequence sequence = new Sequence();
        sequence.addBefore("A");
        sequence.addAfter("B");
        sequence.start();

        // Verify that the current element is now at the start
        assertEquals("A", sequence.getCurrent());
    }

    @Test
    public void testStartWithEmptySequence() {
        // Test starting with an empty sequence
        Sequence sequence = new Sequence();
        sequence.start();

        // Verify that there is no current element
        assertNull(sequence.getCurrent());
    }

    @Test
    public void testStartAfterRemoveCurrent() {
        // Test starting after removing the current element
        Sequence sequence = new Sequence();
        sequence.addBefore("A");
        sequence.addAfter("B");
        sequence.addAfter("C");
        sequence.removeCurrent();
        sequence.start();

        // Verify that the current element is now at the start
```

```
        assertEquals("A", sequence.getCurrent());
    }

    @Test
    public void testStartAfterAddAll() {
        // Test starting after adding another sequence
        Sequence sequence1 = new Sequence();
        sequence1.addBefore("A");
        sequence1.addAfter("B");

        Sequence sequence2 = new Sequence();
        sequence2.addBefore("C");
        sequence2.addAfter("D");

        sequence1.addAll(sequence2);
        sequence1.start();

        // Verify that the current element is now at the start
        assertEquals("A", sequence1.getCurrent());
    }


    @Test
    public void testTrimToSizeWithEmptySequence() {
        // Test trimming the size of an empty sequence
        Sequence sequence = new Sequence();
        sequence.trimToSize();

        // Verify that the capacity is now zero
        assertEquals(0, sequence.getCapacity());
    }

    @Test
    public void testTrimToSizeWithNonEmptySequence() {
        // Test trimming the size of a non-empty sequence
        Sequence sequence = new Sequence();
```

```
        sequence.addBefore("A");
        sequence.addAfter("B");
        sequence.trimToSize();

        // Verify that the capacity is now the size of the sequence
        assertEquals(2, sequence.getCapacity());
    }

    @Test
    public void testTrimToSizeAfterRemoveCurrent() {
        // Test trimming the size after removing the current element
        Sequence sequence = new Sequence();
        sequence.addBefore("A");
        sequence.addAfter("B");
        sequence.addAfter("C");
        sequence.addAfter("D");
        sequence.removeCurrent();
        sequence.trimToSize();

        // Verify that the capacity is now the size of the sequence
        assertEquals(3, sequence.getCapacity());
    }

    @Test
    public void testTrimToSizeAfterAddAll() {
        // Test trimming the size after adding another sequence
        Sequence sequence1 = new Sequence();
        sequence1.addBefore("A");
        sequence1.addAfter("B");

        Sequence sequence2 = new Sequence();
        sequence2.addBefore("C");
        sequence2.addAfter("D");

        sequence1.addAll(sequence2);
        sequence1.trimToSize();
```

```java

        // Verify that the capacity is now the size of the combined sequences
        assertEquals(4, sequence1.getCapacity());
    }

    @Test
    public void testToStringWithEmptySequence() {
        // Test string representation of an empty sequence
        Sequence sequence = new Sequence();
        assertEquals("{} (capacity = 10)", sequence.toString());
    }

    @Test
    public void testToStringWithNonEmptySequence() {
        // Test string representation of a non-empty sequence
        Sequence sequence = new Sequence();
        sequence.addBefore("A");
        sequence.addAfter("B");
        assertEquals("{A, >B} (capacity = 10)", sequence.toString());
    }

    @Test
    public void testToStringAfterRemoveCurrent() {
        // Test string representation after removing the current element
        Sequence sequence = new Sequence();
        sequence.addBefore("A");
        sequence.addAfter("B");
        sequence.removeCurrent();
        assertEquals("{A} (capacity = 10)", sequence.toString());
    }

    @Test
    public void testToStringAfterAddAll() {
```

```java
        // Test string representation after adding another sequence
        Sequence sequence1 = new Sequence(15);
        sequence1.addBefore("A");
        sequence1.addAfter("B");

        Sequence sequence2 = new Sequence(15);
        sequence2.addBefore("C");
        sequence2.addAfter("D");

        sequence1.addAll(sequence2);
        assertEquals("{A, >B, C, D} (capacity = 15)", sequence1.toString());
    }

    @Test
    public void testEqualsWithIdenticalSequences() {
        // Test equals with identical sequences
        Sequence sequence1 = new Sequence();
        sequence1.addBefore("A");
        sequence1.addAfter("B");

        Sequence sequence2 = new Sequence();
        sequence2.addBefore("A");
        sequence2.addAfter("B");

        assertTrue(sequence1.equals(sequence2));
    }

    @Test
    public void testEqualsWithDifferentCurrentElements() {
        // Test equals with identical sequences but different current elements
        Sequence sequence1 = new Sequence();
        sequence1.addBefore("A");
        sequence1.addAfter("B");

        Sequence sequence2 = new Sequence();
        sequence2.addBefore("A");
```

```java
        sequence2.addAfter("B");
        sequence2.start();

        assertFalse(sequence1.equals(sequence2));
    }

    @Test
    public void testEqualsWithDifferentSizes() {
        // Test equals with sequences of different sizes
        Sequence sequence1 = new Sequence();
        sequence1.addBefore("A");
        sequence1.addAfter("B");

        Sequence sequence2 = new Sequence();
        sequence2.addBefore("A");

        assertFalse(sequence1.equals(sequence2));
    }

    @Test
    public void testEqualsWithDifferentCapacities() {
        // Test equals with sequences of different capacities
        Sequence sequence1 = new Sequence();
        sequence1.addBefore("A");
        sequence1.addAfter("B");

        Sequence sequence2 = new Sequence();
        sequence2.addBefore("A");
        sequence2.addAfter("B");

        // Set different capacity for sequence2
        sequence2.ensureCapacity(30);

        assertTrue(sequence1.equals(sequence2));
    }

    @Test
    public void testIsEmptyWithEmptySequence() {
```

```
        // Test isEmpty with an empty sequence
        Sequence sequence = new Sequence();
        assertTrue(sequence.isEmpty());
    }

    @Test
    public void testIsEmptyWithNonEmptySequence() {
        // Test isEmpty with a non-empty sequence
        Sequence sequence = new Sequence();
        sequence.addBefore("A");
        assertFalse(sequence.isEmpty());
    }

    @Test
    public void testIsEmptyAfterAddAllEmptySequence() {
        // Test isEmpty after adding elements from an empty sequence
        Sequence sequence1 = new Sequence();
        Sequence sequence2 = new Sequence();

        sequence1.addAll(sequence2);
        assertTrue(sequence1.isEmpty());
    }
    @Test
    public void testClearWithNonEmptySequence() {
        // Test clear with a non-empty sequence
        Sequence sequence = new Sequence();
        sequence.addBefore("A");
        sequence.addAfter("B");
        sequence.clear();

        // Verify that the sequence is empty
        assertTrue(sequence.isEmpty());

        // Verify that there is no current element
        assertNull(sequence.getCurrent());
    }

    @Test
    public void testClearWithEmptySequence() {
```

```
        // Test clear with an empty sequence
        Sequence sequence = new Sequence();
        sequence.clear();

        // Verify that the sequence is still empty
        assertTrue(sequence.isEmpty());

        // Verify that there is no current element
        assertNull(sequence.getCurrent());
    }

    @Test
    public void testClearWithAddAll() {
        Sequence sequence1 = new Sequence();
        sequence1.addBefore("A");
        sequence1.addAfter("B");

        Sequence sequence2 = new Sequence();
        sequence2.addBefore("C");
        sequence2.addAfter("D");

        sequence1.addAll(sequence2);
        sequence1.clear();

        // Verify that the sequence is still empty
        assertTrue(sequence1.isEmpty());

        // Verify that there is no current element
        assertNull(sequence1.getCurrent());
    }

}
```