```java
package proj3;
/**
 *  Class modeling a Sequence Abstract Data Type (ADT). This is a refactored version of the Sequence class from proj2, this version is LinkedList based
 *
 * Invariant:
 * 1. The number of elements in the sequence is contents.size(), i.e the size of the LinkedList.
 * 2. Sequence should not know about ListNodes.
 * 3. For an empty sequence, the contents LinkedList is empty. For a non-empty sequence, the elements of the sequence are linked from the front to the end.
 * 4. If there is a current element, it lies in contents.getIthItem(currentIndex). If there is no current element, then currentIndex equals the size of contents.
 *
 *
 * @author Neil Daterao
 * @version 02/13/24
 * @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 LinkedList contents;
    private int capacity;
    private int currentIndex;

    private final int DEFAULTCAPACITY = 10;

   /**
     * Creates a new sequence with initial capacity 10.
     */
```

```java
    public Sequence() {
        try { contents = new LinkedList(); }
        catch (OutOfMemoryError memoryError){
            throw new OutOfMemoryError("Indicates insufficient memory for new Sequence");
        }
        capacity = DEFAULTCAPACITY;
        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 { contents = new LinkedList(); }
        catch (OutOfMemoryError memoryError){
            throw new OutOfMemoryError("Indicates insufficient memory for new Sequence");
        }
        capacity = initialCapacity;
        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()) { contents.insertAtIndex(currentIndex, value); }
        else {
            int startingIndex = 0;
            contents.insertAtHead(value);
            setCurrentIndex(startingIndex);
        }

    }


    /**
     * 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.
     */
```

```java
    public void addAfter(String value)
    {
        if (getCapacity() <= size()) { ensureCapacity(2*getCapacity() + 1); }
        if (isCurrent()) {
            int indexAfterCurrent = getCurrentIndex() + 1;
            contents.insertAtIndex(indexAfterCurrent, value);
            setCurrentIndex(indexAfterCurrent);
        }
        else {
            int endOfSequenceIndex = size();
            contents.insertAtTail(value);
            setCurrentIndex(endOfSequenceIndex); // this index is now equal to size()-1, so it is the last valid index.
        }

    }


    /**
     * @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 capacity;
    }


```

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


    /**
     * Increase the sequence's capacity to be
     * minCapacity.  Does nothing
     * if current capacity is already >= minCapacity.
     *
     * @param minCapacity the capacity that the sequence
     * should now have.
     */
    public void ensureCapacity(int minCapacity)
    {
        if (getCapacity() < minCapacity) { setCapacity(minCapacity); }
    }


    /**
     * 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 (just enough) 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()) {
            if (getCurrentIndex() < size()) { currentIndex++; }
        }
        else { noCurrentElement(); }
    }


    /**
     * 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());
            LinkedList clonedContents = contents.
```

```
clone();
            clonedSequence.contents = clonedContents;
            clonedSequence.setCurrentIndex(getCurrentIndex());

            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()) { contents.removeAtIndex(getCurrentIndex());}
    }


    /**
     * @return the number of elements stored in the sequence.
     */
```

```
    public int size()
    {
        return contents.size();
    }


    /**
     * 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()
    {
        setCapacity(size());
    }


    /**
     * 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(contents.getIthItem(index));
            } else {
                sb.append(contents.getIthItem(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; }
        if (other == null) { return false; }


        if (size() == other.size() && getCurrentIndex() == other.getCurrentIndex() ) {
            Sequence clonedOther = other.clone();

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

                if (!contents.getIthItem(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 contents.isEmpty();
    }


    /**
     *  empty the sequence.  There should be no current element.
     */
    public void clear()
    {
        contents.clear();
        noCurrentElement();
    }


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

    /**
```

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


     /**
     *
     * Private setter method to update capacity.
     */
    private void setCapacity(int newCapacityValue
) {
        capacity = newCapacityValue;
    }

    /**
     * Private setter method to set current index
     * @param newIndex
     */
    private void setCurrentIndex(int newIndex) {
        currentIndex = newIndex;
    }



}
```

```java
package proj3;

/**
 * Linked List is a collection of data nodes.  All methods here relate to
 * how one can manipulate those nodes.
 *
 * @author Neil Daterao
 * @version 2/13/2024
 */
public class LinkedList
{
    private int length;
    private ListNode firstNode;

    /**
     * default constructor
     */
    public LinkedList()
    {
        length=0;
        firstNode=null;
    }

    /**
     * return number of items in list
     * @return number of items (nodes) in the list
     */
    public int getLength()
    {
        return length;
    }

    /**
     * tells if this linked list is empty or not
     * @return true if this list has no nodes; else false
     */
    public boolean isEmpty() {
        return getLength()==0;
    }
```

```java

    /** insert new String at linked list's head
     *
     * @param newData the String to be inserted
     */
    public void insertAtHead(String newData)
    {
        ListNode newnode = new ListNode(newData);
        if (isEmpty()) { firstNode=newnode; }
        else
        {
            newnode.next=firstNode;
            firstNode=newnode;
        }
        length++;
    }

    /**
     * returns printable version of this list
     * @return string version of the list
     */
    public String toString(){
        String toReturn = "(";
        ListNode runner = firstNode;
        while(runner != null){
            toReturn = toReturn + runner;
            runner = runner.next;
            if(runner != null){
                toReturn = toReturn + ", ";
            }
        }
        toReturn = toReturn + ")";
        return toReturn;
    }


    /** remove and return data at the head of the list
     *
     *  @return the String the deleted node contains.  Returns null if list empty.
```

```java
     */
    public String removeHead()
    {
        if (!isEmpty())
        {
            ListNode secondNode = firstNode.next;
            String deletedNodeContents = firstNode.data;
            firstNode = secondNode;
            length--;
            return deletedNodeContents;
        }
        else { return null; }

    }


    /** insert data at end of list
     *
     * @param newData new String to be inserted
     */
    public void insertAtTail(String newData)
    {
        if (isEmpty()) { insertAtHead(newData); }

        else
        {
            ListNode newNode = new ListNode(newData);
            ListNode runner = firstNode;

            while (runner.next != null)
            {
                runner = runner.next;

            }
            runner.next = newNode;
            length++;
        }

    }
```

```


    /**
     * search for first occurrence of value and return index where found
     *
     * @param value string to search for
     * @return index where string occurs (first node is index 0).  Return -1 if value not found.
     */
    public int indexOf(String value)
    {
        if (isEmpty()) { return -1; }

        ListNode runner = firstNode;
        int index = 0;

        while (runner != null) {
            if (runner.data.equals(value)) { return index; }

            else
            {
                runner = runner.next;
                index++;
            }

        }
        return -1;
    }


    /**
     * returns the ith item in the list (where first item is index 0)
     * @return the data inside the node at given index. Return null if not there.
     */
    public String getIthItem(int index){
        ListNode runner = firstNode;
        while (runner!=null){
```

```
            if (index==0){
                return runner.data;
            }
            else {
                runner = runner.next;
                index--;
            }
        }
        return null;
    }

    /**
     * Alias for get length, maintain convention in Java Collections.
     * @return size (amount of items in Linked List)
     */
    public int size() { return getLength(); }



    /**
     * Method to insert data at a specified index.
     * @param index Index to be inserted at
     * @param newData
     */
    public void insertAtIndex(int index, String newData) {
        if (index < 0 || index > length) {
            throw new IndexOutOfBoundsException("Invalid index: " + index);
        }
        ListNode runner = firstNode;
        ListNode newNode = new ListNode(newData);

        if (index == 0) { insertAtHead(newData); }

        else {

            for (int i = 0; i < index - 1; i++) {
                runner = runner.next;
```

```
            }

            newNode.next = runner.next;
            runner.next = newNode;
            length++;
        }


    }

    /**
     * Return a deepcopy of a linkedlist.
     */
    public LinkedList clone() {
        LinkedList newList = new LinkedList();
        ListNode runner = firstNode;

        while (runner != null) {
            newList.insertAtTail(runner.data);
            runner = runner.next;
        }

        return newList;
    }

    /**
     * Returns true if two objects are equal, false if not.
     */
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (obj == null || getClass() != obj.getClass()) {
            return false;
        }
        LinkedList otherList = (LinkedList) obj;

        if (this.size() != otherList.size()) {
            return false;
```

```java
        }

        ListNode thisRunner = this.firstNode;
        ListNode otherRunner = otherList.firstNode;

        while (thisRunner != null) {
            if (!thisRunner.data.equals(otherRunner.data)) {
                return false;
            }
            thisRunner = thisRunner.next;
            otherRunner = otherRunner.next;
        }

        return true;
    }

    /**
     *
     * @param value
     * @return True if linked list contains value, false if not
     */
    public boolean contains(String value) {
        ListNode runner = firstNode;
        while (runner != null) {
            if (runner.data.equals(value)) {
                return true;
            }
            runner = runner.next;
        }
        return false;
    }


    /**
     * Method to remove data at a specified index.
     * @param index Index to be removed
     */
    public void removeAtIndex(int index) {
        if (index < 0 || index >= length) {
```

```
            throw new IndexOutOfBoundsException("Invalid index: " + index);
        }
        if (index == 0) { removeHead(); }
        else {

            ListNode previousNode = getNodeAtIndex(index - 1);
            ListNode currentNode = previousNode.next;
            previousNode.next = currentNode.next; //skip over currentNode.
            length--;
        }


    }

    /**
     * Private helper/getter to get the node at a specified index
     * @param index given index
     * @return ListNode at given index.
     */
    private ListNode getNodeAtIndex(int index) {
        ListNode current = firstNode;
        for (int i = 0; i < index; i++) {
            current = current.next;
        }
        return current;
    }


    /**
     * Empty the LinkedList.
     */
    public void clear() {
        firstNode = null;
        length = 0;
    }

```

```
}
```

```java
package proj3;

/**
 * The ListNode class is more data-specific than the LinkedList class.  It
 * details what a single node looks like.  This node has one data field,
 * holding a pointer to a String object.
 *
 * @author Neil Daterao
 *
 */
public class ListNode
{
    public String data;
    public ListNode next;

    public ListNode(String newData) {
        data = newData;
        next = null;
    }

    public String toString(){
        return data;
    }

}

```

```java
package test;

import proj3.LinkedList;
import org.junit.*;
import org.junit.rules.Timeout;
import static org.junit.Assert.*;

/**
 * LinkedList testing class for all methods in class.
 * @author Neil Daterao
 */

public class LinkedListTester {

    @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 testRemoveHead() {
        LinkedList ll = new LinkedList();
        //Linked list with multiple elements
        ll.insertAtHead("A");
        ll.insertAtHead("B");
        ll.insertAtHead("C");
        ll.insertAtHead("D");
        ll.insertAtHead("E");
        ll.insertAtHead("F");
        ll.insertAtHead("G");

        ll.removeHead();

        //test remove head
        assertEquals("(F, E, D, C, B, A)", ll.toString());
        assertEquals(6, ll.getLength());

    }

    @Test
```

```java
    public void testRemoveHeadFromListWithOneElement() {
        // Create a linked list with one element
        LinkedList ll = new LinkedList();
        ll.insertAtHead("A");

        // Test removing from a list with one element
        assertEquals("A", ll.removeHead());
        assertTrue(ll.isEmpty());
    }

    @Test
    public void testRemoveHeadFromEmptyList() {
        // Create an empty linked list
        LinkedList ll = new LinkedList();

        // Test removing from an empty list
        assertNull(ll.removeHead());
    }

    @Test
    public void testInsertAtTailInEmptyList() {
        // Create an empty linked list
        LinkedList ll = new LinkedList();

        // Insert at the tail of an empty list
        ll.insertAtTail("A");

        // Verify the list after insertion
        assertEquals("(A)", ll.toString());
        assertEquals(1, ll.getLength());
    }

    @Test
    public void testInsertAtTailInListWithOneElement() {
        // Create a linked list with one element
        LinkedList ll = new LinkedList();
        ll.insertAtHead("A");

```

```
        // Insert at the tail of a list with one element
        ll.insertAtTail("B");

        // Verify the list after insertion
        assertEquals("(A, B)", ll.toString());
        assertEquals(2, ll.getLength());
    }

    @Test
    public void testInsertAtTailInListWithMultipleElements() {
        // Create a linked list with multiple elements
        LinkedList ll = new LinkedList();
        ll.insertAtHead("A");
        ll.insertAtTail("B");
        ll.insertAtTail("C");
        ll.insertAtTail("D");
        ll.insertAtTail("E");
        ll.insertAtTail("F");
        ll.insertAtTail("G");

        // Verify the list after insertion
        assertEquals("(A, B, C, D, E, F, G)", ll.toString());
        assertEquals(7,ll.getLength());
    }


    @Test
    public void testIndexOfInEmptyList() {
        // Create an empty linked list
        LinkedList list = new LinkedList();

        // Test indexOf in an empty list
        assertEquals(-1, list.indexOf("A"));
        assertEquals(-1, list.indexOf("B"));
        assertEquals(-1, list.indexOf("C"));
    }

```

```java
    @Test
    public void testIndexOfInListWithOneElement() {
        // Create a linked list with one element
        LinkedList list = new LinkedList();
        list.insertAtHead("A");

        // Test indexOf in a list with one element
        assertEquals(0, list.indexOf("A"));
        assertEquals(-1, list.indexOf("B"));
        assertEquals(-1, list.indexOf("C"));
    }

    @Test
    public void testIndexOfInListWithMultipleElements() {
        // Create a linked list with multiple elements
        LinkedList list = new LinkedList();
        list.insertAtHead("A");
        list.insertAtTail("B");
        list.insertAtTail("C");
        list.insertAtTail("D");
        list.insertAtTail("E");
        list.insertAtTail("F");
        list.insertAtTail("G");

        // Test indexOf in a list with multiple elements
        assertEquals(0, list.indexOf("A"));
        assertEquals(1, list.indexOf("B"));
        assertEquals(2, list.indexOf("C"));
        assertEquals(3, list.indexOf("D"));
        assertEquals(4, list.indexOf("E"));
        assertEquals(6, list.indexOf("G"));
        assertEquals(-1, list.indexOf("Z"));


    }

    @Test
    public void testIndexOfWithDuplicateValues() {
```

```java
        // Create a linked list with duplicate values
        LinkedList list = new LinkedList();
        list.insertAtHead("A");
        list.insertAtHead("B");
        list.insertAtHead("A");

        // Test indexOf with duplicate values
        assertEquals(0, list.indexOf("A"));
        assertEquals(1, list.indexOf("B"));
        assertEquals(-1, list.indexOf("C"));
    }

    @Test
    public void testInsertAtIndex() {
        LinkedList list = new LinkedList();
        list.insertAtHead("A");
        list.insertAtHead("B");
        list.insertAtHead("C");

        // Test inserting at index 0
        list.insertAtIndex(0, "X");
        assertEquals("(X, C, B, A)", list.toString());

        // Test inserting in the middle
        list.insertAtIndex(2, "Y");
        assertEquals("(X, C, Y, B, A)", list.toString());

        // Test inserting at the end
        list.insertAtIndex(5, "Z");
        assertEquals("(X, C, Y, B, A, Z)", list.toString());

        // Test inserting at index equal to length (append)
        list.insertAtIndex(6, "W");
        assertEquals("(X, C, Y, B, A, Z, W)", list.toString());
    }
```

```java

    @Test
    public void testInsertAtIndexEmptyList() {
        LinkedList list = new LinkedList();

        // Test inserting into an empty list
        list.insertAtIndex(0, "X");
        assertEquals("(X)", list.toString());
    }

    @Test
    public void testInsertAtIndexInvalidIndex() {
        LinkedList list = new LinkedList();
        list.insertAtHead("A");
        list.insertAtHead("B");

        // Test inserting at index out of bounds
        assertThrows(IndexOutOfBoundsException.class, () -> {
            list.insertAtIndex(3, "X");
        });

        assertThrows(IndexOutOfBoundsException.class, () -> {
            list.insertAtIndex(-1, "X");
        });


    }

    @Test
    public void testGetIthItem() {
        LinkedList list = new LinkedList();
        list.insertAtHead("A");
        list.insertAtHead("B");
        list.insertAtHead("C");

        // Test getting the first item (index 0)
        assertEquals("C", list.getIthItem(0));

        // Test getting an item in the middle
```

```
        assertEquals("B", list.getIthItem(1));

        // Test getting the last item
        assertEquals("A", list.getIthItem(2));
    }

    @Test
    public void testGetIthItemIndexOutOfBounds() {
        LinkedList list = new LinkedList();
        list.insertAtHead("A");

        // Test getting item at index out of bounds
        assertNull(list.getIthItem(1));
    }

    @Test
    public void testGetIthItemEmptyList() {
        LinkedList list = new LinkedList();

        // Test getting item from an empty list
        assertNull(list.getIthItem(0));
    }

    @Test
    public void testClone() {
        LinkedList originalList = new LinkedList();
        originalList.insertAtHead("A");
        originalList.insertAtHead("B");
        originalList.insertAtHead("C");

        // Clone the original list
        LinkedList clonedList = originalList.clone();

        // Test that the cloned list is not the same object reference as the original list
        assertNotSame(originalList, clonedList);

        // Test that the data of each node in the cloned list matches the data of the corresponding node in the original list
```

```
        assertEquals(originalList.toString(), clonedList.toString());

        LinkedList emptyList = new LinkedList();
        LinkedList clonedEmptyList = emptyList.clone();

        assertEquals(emptyList.toString(), clonedEmptyList.toString());
    }

    @Test
    public void testEqualsSameObject() {
        LinkedList list = new LinkedList();
        list.insertAtHead("A");
        list.insertAtHead("B");
        list.insertAtHead("C");

        assertTrue(list.equals(list)); // Same object should be equal
    }

    @Test
    public void testEqualsNullObject() {
        LinkedList list = new LinkedList();
        list.insertAtHead("A");
        list.insertAtHead("B");
        list.insertAtHead("C");

        assertFalse(list.equals(null)); // Null object should not be equal
    }

    @Test
    public void testEqualsDifferentClass() {
        LinkedList list = new LinkedList();
        list.insertAtHead("A");
        list.insertAtHead("B");
        list.insertAtHead("C");

        assertFalse(list.equals("hello")); //
```

```
Different class should not be equal
    }

    @Test
    public void testEqualsDifferentLists() {
        LinkedList list1 = new LinkedList();
        list1.insertAtHead("A");
        list1.insertAtHead("B");
        list1.insertAtHead("C");

        LinkedList list2 = new LinkedList();
        list2.insertAtHead("X");
        list2.insertAtHead("Y");
        list2.insertAtHead("Z");

        assertFalse(list1.equals(list2)); // Different lists should not be equal
    }

    @Test
    public void testEqualsEqualLists() {
        LinkedList list1 = new LinkedList();
        list1.insertAtHead("A");
        list1.insertAtHead("B");
        list1.insertAtHead("C");

        LinkedList list2 = new LinkedList();
        list2.insertAtHead("A");
        list2.insertAtHead("B");
        list2.insertAtHead("C");

        assertTrue(list1.equals(list2)); // Equal lists should be equal
    }

    @Test
    public void testEqualsUnequalLengthLists() {
        LinkedList list1 = new LinkedList();
        list1.insertAtHead("A");
        list1.insertAtHead("B");
        list1.insertAtHead("C");
```

```java

        LinkedList list2 = new LinkedList();
        list2.insertAtHead("A");
        list2.insertAtHead("B");

        assertFalse(list1.equals(list2)); // Lists of unequal length should not be equal
    }

    @Test
    public void testEqualsEmptyLists() {
        LinkedList list1 = new LinkedList();
        LinkedList list2 = new LinkedList();

        assertTrue(list1.equals(list2)); // Empty lists should be equal
    }

    @Test
    public void testContainsEmptyList() {
        LinkedList list = new LinkedList();
        assertFalse(list.contains("A")); // Empty list should not contain any value
    }

    @Test
    public void testContainsSingleElement() {
        LinkedList list = new LinkedList();
        list.insertAtHead("A");
        assertTrue(list.contains("A")); // List with a single element should contain that value
        assertFalse(list.contains("B")); // List with a single element should not contain other values
    }

    @Test
    public void testContainsMultipleElements() {
        LinkedList list = new LinkedList();
        list.insertAtHead("A");
        list.insertAtHead("B");
```

```
        list.insertAtHead("C");

        assertTrue(list.contains("A")); // List should contain "A"
        assertTrue(list.contains("B")); // List should contain "B"
        assertTrue(list.contains("C")); // List should contain "C"
        assertFalse(list.contains("X")); // List should not contain "X"
    }

    @Test
    public void testContainsDuplicateElements() {
        LinkedList list = new LinkedList();
        list.insertAtHead("A");
        list.insertAtHead("A");

        assertTrue(list.contains("A")); // List with duplicate elements should contain "A"
    }


    @Test
    public void testRemoveAtIndexFromEmptyList() {
        LinkedList list = new LinkedList();
        assertThrows(IndexOutOfBoundsException.class, () -> {
            list.removeAtIndex(0);
        });
    }

    @Test
    public void testRemoveAtIndexOutOfBounds() {
        LinkedList list = new LinkedList();
        list.insertAtHead("A");
        list.insertAtHead("B");
        list.insertAtHead("C");
        assertThrows(IndexOutOfBoundsException.class, () -> {
            list.removeAtIndex(3); // Index out of
```

```
bounds
        });
    }

    @Test
    public void testRemoveAtIndexFirstNode() {
        LinkedList list = new LinkedList();
        list.insertAtHead("A");
        list.insertAtHead("B");
        list.insertAtHead("C");
        list.removeAtIndex(0);
        assertEquals("(B, A)", list.toString());
    }

    @Test
    public void testRemoveAtIndexLastNode() {
        LinkedList list = new LinkedList();
        list.insertAtHead("A");
        list.insertAtHead("B");
        list.insertAtHead("C");
        list.removeAtIndex(2);
        assertEquals("(C, B)", list.toString());
    }

    @Test
    public void testRemoveAtIndexMiddleNode() {
        LinkedList list = new LinkedList();
        list.insertAtHead("A");
        list.insertAtHead("B");
        list.insertAtHead("C");
        list.removeAtIndex(1);
        assertEquals("(C, A)", list.toString());
    }


    @Test
    public void testClearEmptyList() {
        LinkedList list = new LinkedList();
        list.clear();
        assertTrue(list.isEmpty());
        assertEquals(0, list.getLength());
```

```
    }

    @Test
    public void testClearNonEmptyList() {
        LinkedList list = new LinkedList();
        list.insertAtHead("A");
        list.insertAtHead("B");
        list.insertAtHead("C");
        list.clear();
        assertTrue(list.isEmpty());
        assertEquals(0, list.getLength());
    }

    @Test
    public void testClearMultipleTimes() {
        LinkedList list = new LinkedList();
        list.insertAtHead("A");
        list.insertAtHead("B");
        list.clear();
        list.clear();
        assertTrue(list.isEmpty());
        assertEquals(0, list.getLength());
    }



}
```

```java
package test;

import org.junit.*;
import org.junit.rules.Timeout;
import static org.junit.Assert.*;
import proj3.*;


/**
 * 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 SequenceTester {


    @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
```

```java
        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 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
```

```java
        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

        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());

```

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

```java
        Sequence sequence = new Sequence();
        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);
```

```java

        // 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");
```

```java
        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
        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();
```

```java

        // 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
```

```java
        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
```

```java
        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();

        // 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
```

```java
    public void testToStringAfterAddAll() {
        // 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");
        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
```

```java
    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
```

```java
    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());
    }

}
```