```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.
 *
 * This is the only class where I'll let you use public instance variables.
 *
 */
public class ListNode
{
    public String data;
    public ListNode next;

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

    public String toString(){
        return data;
    }

}
```

```java
package proj3;  // Gradescope needs this.
/**
 * Emma Vu - Project 3
 * @version 5/11/2020
 *
 * CLASS INVARIANTS: From p150 and Java objects
 *
 *  Our suggested design for the sequence ADT has four private instance variables.
 * The first variable, contents, is a LinkedList that stores the elements of the sequence. A second instance variable,
 * called manyItems, keeps track of how much of the contents is currently
 * being used. The third instance variable, currentIndex, gives the
 * index of the current element in the LinkedList (if there is one). Sometimes a sequence
 * has no current element, in which case currentIndex will be set to the same
 * number as manyItems (since this is larger than any valid index).
 * The final instance variable is capacity, which stores the capacity of the sequence.
 * Capacity should not be changed (i.e, increments by 1) after adding elements that still not outnumber the allowed capacity.
 *
 * The complete invariant of our ADT is stated as these rules:
 * 1. The number of elements in the sequence is stored in the instance variable
 * manyItems.
 * 2. For an empty sequence (with no elements), the length of the LinkedList = 0,
 * for a nonempty sequence, the ith element of the sequence is stored correspondingly to the ith element in the LinkedList
 * 3. If there is a current element, then it lies in contents.getItemAt(currentIndex); if there
 * is no current element, then currentIndex < 0 or currentIndex >= manyItems
 * By default, we set currentIndex = manyItems for an empty sequence that doesn't have a current element

 */
public class Sequence
{
    //Instance variables
    private LinkedList contents;
    private int capacity;
    private int currentIndex;
    private int manyItems;

    private final int INITIAL_CAPACITY = 10;


    /**
     * Creates a new default sequence with initial capacity 10.
     * capacity is reflected in the length of the internal LinkedList
     */
    public Sequence() {
```

```
        capacity = INITIAL_CAPACITY;

        currentIndex = manyItems;
        contents = new LinkedList();


    }


    /**
     * Creates a new non-default sequence.
     * capacity is reflected in the length of the internal LinkedList
     *
     * @param initialCapacity the initial capacity of the sequence. Have to
be positive integer
     */
    public Sequence(int initialCapacity){
        if(initialCapacity < 0){

            initialCapacity = INITIAL_CAPACITY;
        }

        capacity = initialCapacity;

        currentIndex = manyItems;
        contents = new LinkedList();

    }


    /**
     * 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 sequence has current element, there's no need to change pointer
 position after adding.
     *
     * 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(size()+ 1 > getCapacity()) {
            ensureCapacity(size()*2 + 1);
        }

        if(isCurrent()){
            addAtIndex(value, currentIndex);
```

```
        }
        else{
            addAtIndex(value, 0);
            currentIndex = 0;
        }


    }


    /**
     * 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 sequence has a current element, then after adding, the
currentIndex will increment by 1
     * If the current element is at the end of the sequence then after adding
, the size of sequence will increment by 1,
     * so the currentIndex = the old size before adding value which is at the
 end of the sequence
     * 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(size() + 1 > getCapacity()) {
            ensureCapacity(size()*2 + 1);
        }

        if(isCurrent()) {
            addAtIndex(value, currentIndex + 1);

            currentIndex = currentIndex + 1;
        }
        else {
            addAtIndex(value, manyItems);

            currentIndex = manyItems - 1;
        }

    }


    /**
     * @return true if and only if the sequence has a current element.
     */
    public boolean isCurrent()
    {
```

```
        return manyItems != currentIndex;
    }


    /**
     * @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 null;
        }
        else{
            return this.contents.getItemAt(currentIndex);
        }
    }


    /**
     * 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) {
            capacity = 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)
    {
        int bothSize = this.size() + another.size();

        if(!isCurrent()){
            currentIndex = bothSize;
        }

        if(this.size()+ another.size() > getCapacity()) {
            ensureCapacity(bothSize);
        }


        for(int i = 0; i < another.size(); i++){

            this.contents.insertAtTail(another.contents.getItemAt(i));

        }
        this.manyItems += another.manyItems;
    }


    /**
     * 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()){

            currentIndex = currentIndex + 1;
        }
    }


    /**
     * 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()
{
    Sequence cloneSq = new Sequence();
    cloneSq.addAll(this);
    cloneSq.currentIndex = this.currentIndex;
    cloneSq.manyItems = this.manyItems;
    cloneSq.capacity = this.capacity;

    return cloneSq;




}


/**
 * 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()){
        this.contents.removeAtIndex(currentIndex);
        manyItems--;

    }
}


/**
 * @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()
    {
        currentIndex = 0;
    }


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

    /**
     * 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()
    {

        String printToString = "{} (capacity = " + getCapacity() + ")";

        if(!isEmpty()){
            printToString = toStringOfNonEmpty();
        }

        return printToString;
    }



    /**
     * Private helper method to print toString of a non empty sequence
     * 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.
     *
     * @return
     */

    private String toStringOfNonEmpty(){
        String nonEmptyString = "{";
        for(int i = 0; i < size(); i++){
            if(i == 0) {
                if (i == currentIndex) {
                    nonEmptyString = nonEmptyString + ">" + getCurrent();
                }
                else {
                    nonEmptyString = nonEmptyString + contents.getItemAt(i);
                }
            }
            else{
                if(i == currentIndex){
                    nonEmptyString = nonEmptyString + ", >" + getCurrent();

                }
                else{
                    nonEmptyString = nonEmptyString + ", " + contents.
getItemAt(i);
                }
            }

        }
        nonEmptyString = nonEmptyString + "} (capacity = " + getCapacity() +
")";
        return nonEmptyString;
    }


    /**
     * 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)
    {
```

```
        return this.size() == other.size() && this.currentIndex == other.
currentIndex
                && this.contents.toString().equals(other.contents.toString
());
    }


    /**
     *
     * @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()
    {
        manyItems = 0;
        currentIndex = manyItems;
        contents = new LinkedList();
    }

    /** Add value to a specified position of the LinkedList Data.
     * After adding the size will increment by 1
     * PRECONDITION: position has to be positive, position cannot be greater
than length
     * POST-CONDITION: data get one extra element, at the specified position.
     * @param value The value we want to add to data
     * @param index The position we want to add.
     */
    private void addAtIndex(String value, int index){
        if(index >= 0) {

            contents.insertAtIndex(value, index);

            manyItems += 1;
        }
    }

}
```

```java
package proj3;

/**
 *  Emma Vu - Project 3
 * @version 5/11/2020
 * LinkedList is a collection of data nodes. All methods here relate to how one can manipulate those nodes
 * The LinkedList class gives the access to the beginning of a LinkedList through instance variable called firstNode
 * The length of the LinkedList (number of the items) is stored in an instance variable called length
 * The last Node of the list has variable next = null
 * The LinkedList is defined by the firstNode.
 * The second node is referred by firstNode.next, the third node is referred by firstNode.next.next, etc.
 * When we reach the last node of the LinkedList, the last node.next will be null.
 * For an empty LinkedList to start with, the default is length = 0 and firstNode = null
 *
 */
public class LinkedList
{
    //Instance variables
    private int length;
    private ListNode firstNode;

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

    /**
     * get the length of the LinkedList
     * @return length
     */
    public int getLength()
    {
        return length;
    }

    /** insert new String at linked list's head
     *
     * @param data the String to be inserted
     *
     */

    public void insertAtHead(String data)
    {
        ListNode newNode = new ListNode(data);
        if (isEmpty())
```

```
        {
            firstNode=newNode;
        }
        else
        {
            newNode.next=firstNode;
            firstNode=newNode;
        }
        length++;
    }


    /**
     * return the String that denotes the elements in the list with correct
order
     * @return The list of element in correct order
     */
    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;
    }

    /**
     * Check if the List if empty or not
     * @return true if length is > 0, false otherwise
     */
    public boolean isEmpty(){
        return getLength() == 0;
    }

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

            ListNode runner = firstNode;
```

```
            while(runner.next!=null){
                runner = runner.next;
            }
            runner.next = insertNode;

        }
        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)
    {   int index = 0;

        ListNode runner = firstNode;
        while(runner!= null){
            if (runner.data.equals(value)){
                return index;
            }
            else{
                runner = runner.next;
                index ++;
            }
        }
        return -1;

    }

    /**
     * Remove element at specific index. If the linkedlist is empty, do
nothing
     * If the index is invalid (< 0 or > linkedlist.length), then do nothing
     * If the index is at the beginning of the linkedlist, remove the head of
 the linkedlist
     * If the index is at the end of the linkedlist, remove the tail of the
linkedlist
     * After removing, the length will decrement by 1
     * @param index the index of the element want to remove
     */

    public void removeAtIndex(int index){
        if(!isEmpty()){
            removeAtIndexNonEmpty(index);
        }

    }

    /**
```

```
     * remove the element at the end of the linkedlist
     * if the linkedlist is empty, do nothing.
     * If the linkedlist has one element, then after removing the linkedlist
becomes empty
     * after removing, the length will decrement by 1
     */

    public void removeTail(){
        if(isEmpty()){
            return;
        }
        if(firstNode.next == null){
            firstNode = null;
            length = 0;
            return;
        }
        ListNode runner = firstNode;
        while(runner.next.next!=null){
            runner = runner.next;
        }
        runner.next = null;
        length --;


    }

    /** remove and return data at the head of the list
     *
     *  @return the String the deleted node contains.  Returns null if list
empty.
     */
    public String removeHead()
    {
        if(isEmpty()){
            return null;
        }
        else{
            String remove = firstNode.data;
            firstNode = firstNode.next;
            length--;
            return remove;
        }

    }

    /**
     * Get the Data of the Node at a specified position
     * @param index the position to get the data
     * @return null for invalid index, and the data otherwise
     */


    public String getItemAt(int index){
```

```
        if(!isEmpty() && index >= 0 && index < getLength()){

            return getNodeAt(index).data;
        }
        else{
            return null;
        }
    }

    /**
     * Get the ith Node in the LinkedList
     * @param index the position of the node to get
     * @return the Node at position index
     */
    private ListNode getNodeAt(int index){
        ListNode nodeToGet = firstNode;
        int count = 0;
        while(nodeToGet != null && count < index){
            nodeToGet = nodeToGet.next;
            count++;
        }
        return nodeToGet;



    }

    /**
     * A private helper method to remove the element at specific index of a
non-empty linkedlist
     * Removing by running the pointer to the wanted index and then storing
pointer to the next of node to be deleted
     * After that, unlink the deleted node from linkedlist. After removing,
the length will decrement by 1
     *
     * If the index is invalid (< 0 or > linkedlist.length), then do nothing
     * If the index is at the beginning of the linkedlist, remove the head of
 the linkedlist
     * If the index is at the end of the linkedlist, remove the tail of the
linkedlist
     *
     * @param index the index of the element want to remove
     */


    private void removeAtIndexNonEmpty(int index){
        if (index == 0) {
            removeHead();
        }
        else if (index == getLength()) {
            removeTail();
        }
        else if (index > getLength() || index < 0){
```

```
            return;
        }
        else {
            ListNode runner = firstNode;
            int counter = 0;
            while(runner!=null && counter < index-1){
                runner = runner.next;
                counter++;

            }

            ListNode temp = runner.next.next;


            runner.next = temp;
            length--;
        }
    }

    /**
     * Insert a new Node at a specified position of the list, with the
specified Data
     * After inserting, the length will increment by 1
     * Inserting by having current node in the wanted index moving right to
one position
     * and having the node with the wanted value be inserted in the wanted
index
     *
     * If the index is invalid (< 0 or > linkedlist.length), then do nothing
     * If the index is at the beginning of the linkedlist, insert at the head
 of the linkedlist
     * If the index is at the end of the linkedlist, insert at the tail of
the linkedlist
     *
     * @param value The data of the new Node
     * @param index The position to insert that new Node
     */
     public void insertAtIndex(String value, int index) {
        ListNode nodeToInsert = new ListNode(value);
        if(isEmpty()){
            firstNode = nodeToInsert;
        }

         if (index == 0) {
             insertAtHead(value);
         }
         else if(index == getLength()) {
             insertAtTail(value);
         }
         else if(index < 0 || index > getLength()){
             return;
         }
         else{
```

```
            ListNode temp = getNodeAt(index - 1).next;
            getNodeAt(index - 1).next = new ListNode(value);
            getNodeAt(index - 1).next.next = temp;
            length ++;
        }
    }



}
```

```java
/**
 * JUnit test class.  Use these tests as models for your own.
 */
import org.junit.*;


import org.junit.rules.Timeout;

import static org.junit.Assert.*;

import proj3.LinkedList;

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
    //Remove empty LinkedList.
    //Return null, should have 0 nodes, the content doesn't change
    public void testRemoveHeadEmpty(){
        LinkedList ll = new LinkedList();
        assertNull(ll.removeHead());
        assertEquals(0,ll.getLength());
        assertEquals("()",ll.toString());
    }
    @Test
    //Remove LinkedList with one element. Return the only element after
removing
    //Should have 0 nodes, the content will change
    public void testRemoveHeadOne(){
        LinkedList ll = new LinkedList();
        ll.insertAtHead("A");

        String expected = ll.removeHead();
        assertEquals(0,ll.getLength());
        assertEquals("A",expected);
        assertEquals("()",ll.toString());

    }
    @Test
    //Remove LinkedList with more than one element
    //The length should decrement by 1. Return the removed element.
    // Content will change
    public void testRemoveHeadNonEmpty(){
        LinkedList ll = new LinkedList();
        ll.insertAtHead("A");
        ll.insertAtTail("B");
        ll.insertAtTail("C");
        ll.insertAtTail("D");
        String expected = ll.removeHead();
        assertEquals(3,ll.getLength());
        assertEquals("A",expected);
```

```
        assertEquals("(B, C, D)",ll.toString());




    }

    @Test
    //Remove the first element in a multiple element LinkedList even though
it is identical to the others.
    //Should remove the exact element in the exact position. Length will
decrement by 1.
    //Should not alter the values besides removing from the LinkedList
    public void removeHeadSameMultiple(){
        LinkedList ll = new LinkedList();
        ll.insertAtHead("A");
        ll.insertAtTail("A");
        ll.insertAtTail("A");
        ll.insertAtTail("A");
        ll.insertAtTail("A");
        assertEquals("A",ll.removeHead());
        assertEquals(4,ll.getLength());
        assertEquals("(A, A, A, A)",ll.toString());

    }

    @Test
    //Remove head of a LinkedList that has two identical elements and after
removing there will only be one left
    //The length should decrement by 1. Content will change
    public void testRemoveHeadSameTwice(){
        LinkedList ll = new LinkedList();
        ll.insertAtHead("A");
        ll.insertAtTail("A");
        assertEquals("A",ll.removeHead());
        assertEquals(1,ll.getLength());
        assertEquals("(A)",ll.toString());
    }
    @Test
    //Insert an empty LinkedList. Then it became the first element of
LinkedList
    //The length should increment by 1, content will change
    public void testInsertTailEmpty(){
        LinkedList ll = new LinkedList();
        ll.insertAtTail("A");
        assertEquals(1,ll.getLength());
        assertEquals("(A)",ll.toString());

    }
    @Test
    //Insert LinkedList with more than one elements
    //The length should increment by 1
    // content will change as the inserted element will be added at the end
```

```
    public void testInsertTailNonEmpty(){
        LinkedList ll = new LinkedList();
        ll.insertAtHead("C");
        ll.insertAtHead("B");
        ll.insertAtHead("A");
        ll.insertAtTail("Z");
        assertEquals(4,ll.getLength());
        assertEquals("(A, B, C, Z)",ll.toString());


    }
    @Test
    //Insert to one element LinkedList
    //The length should increment by 1
    //content will change as the inserted element will be added at the end
    public void testInsertTailOne(){
        LinkedList ll = new LinkedList();
        ll.insertAtHead("A");
        ll.insertAtTail("Z");
        assertEquals(2,ll.getLength());
        assertEquals("(A, Z)",ll.toString());

    }

    @Test
    //Insert an identical element to a one-element LinkedList.
    // The LinkedList will have identical elements instead of unable to
insert.
    //Length will increment by 1

    public void testInsertTailSameOne(){
        LinkedList ll = new LinkedList();
        ll.insertAtHead("A");
        ll.insertAtTail("A");
        assertEquals(2,ll.getLength());
        assertEquals("(A, A)",ll.toString());


    }

    @Test
    //Insert tail of a multiple identical element LinkedList. Length and
content should change
    public void testInsertTailSameMultiple(){
        LinkedList ll = new LinkedList();
        ll.insertAtHead("F");
        ll.insertAtHead("F");
        ll.insertAtHead("F");
        ll.insertAtHead("F");
        ll.insertAtHead("F");
        ll.insertAtTail("F");
        assertEquals(6,ll.getLength());
        assertEquals("(F, F, F, F, F, F)",ll.toString());
```

```
    }
    @Test
    //IndexOf empty LinkedList. Return -1. Should not change LinkedList
content nor the length
    public void testIndexOfEmpty(){
        LinkedList ll = new LinkedList();
        assertEquals(-1,ll.indexOf("A"));
        assertEquals("()",ll.toString());
        assertEquals(0,ll.getLength());

    }
    @Test
    //IndexOf invalid data of one element LinkedList. Return -1
    //Should not change LinkedList content nor the length
    public void testIndexOfInvalidOne(){
        LinkedList ll = new LinkedList();
        ll.insertAtHead("A");
        assertEquals(-1,ll.indexOf("B"));
        assertEquals(1,ll.getLength());
        assertEquals("(A)",ll.toString());



    }

    @Test
    //IndexOf invalid data of more than one element LinkedList. Return -1
    //Should not change LinkedList content nor the length
    public void testIndexOfInvalidMultiple(){
        LinkedList ll = new LinkedList();
        ll.insertAtHead("A");
        ll.insertAtTail("B");
        ll.insertAtTail("C");
        ll.insertAtTail("D");
        assertEquals(-1,ll.indexOf("E"));
        assertEquals(4,ll.getLength());
        assertEquals("(A, B, C, D)",ll.toString());



    }
    @Test
    //IndexOf one element LinkedList. Return the index of that data (0)
    //Should not change LinkedList content nor the length
    public void testIndexOfOne(){
        LinkedList ll = new LinkedList();
        ll.insertAtHead("A");
        assertEquals(0,ll.indexOf("A"));
        assertEquals(1,ll.getLength());
        assertEquals("(A)",ll.toString());
```

```
    }
    @Test
    //IndexOf more than one element LinkedList
    //Should return the corresponding indexes when given valid data
    //Should not change LinkedList content nor the length
    public void testIndexOfMultiple(){
        LinkedList ll = new LinkedList();
        ll.insertAtHead("A");
        ll.insertAtTail("B");
        ll.insertAtTail("C");
        ll.insertAtTail("D");
        assertEquals(0,ll.indexOf("A"));
        assertEquals(1,ll.indexOf("B"));
        assertEquals(2,ll.indexOf("C"));
        assertEquals(3,ll.indexOf("D"));
        assertEquals(4,ll.getLength());
        assertEquals("(A, B, C, D)",ll.toString());
    }
    @Test
    //IndexOf identical data that appear in the LinkedList more than once.
    //Return the first occurrence of that data
    //Should not change LinkedList content nor the length
    public void testIndexOfSameMultiple(){
        LinkedList ll = new LinkedList();
        ll.insertAtHead("C");
        ll.insertAtTail("C");
        ll.insertAtTail("C");
        ll.insertAtTail("C");
        ll.insertAtTail("C");
        assertEquals(0,ll.indexOf("C"));
        assertEquals(5,ll.getLength());
        assertEquals("(C, C, C, C, C)",ll.toString());



    }

    @Test
    //Get index of a LinkedList that has two identical elements. Should get
the correct index (0).
    //Should not change content nor length
    public void testIndexOfSameTwice(){
        LinkedList ll = new LinkedList();
        ll.insertAtHead("A");
        ll.insertAtTail("A");
        assertEquals(0,ll.indexOf("A"));
        assertEquals(2,ll.getLength());
        assertEquals("(A, A)",ll.toString());
    }

    @Test
    //Test insertAtHead to an empty LinkedList. The inserted element will be
the first element in the LinkedList
```

```
    //Content will change, the length will increment by 1
    public void testInsertHeadEmpty(){
        LinkedList ll = new LinkedList();
        ll.insertAtHead("A");
        assertEquals("(A)",ll.toString());
        assertEquals(1,ll.getLength());

    }
    @Test
    //Test insertAtHead to a LinkedList with one element. The length will
increment by 1. Content will change
    public void testInsertHeadOne(){
        LinkedList ll = new LinkedList();
        ll.insertAtTail("A");
        ll.insertAtHead("D");
        assertEquals(2,ll.getLength());
        assertEquals("(D, A)",ll.toString());

    }

    @Test
    //Test insertAtHead to a LinkedList with more than one elements. The
content and length will change
    public void testInsertHeadMultiple(){
        LinkedList ll = new LinkedList();
        ll.insertAtTail("S");
        ll.insertAtTail("D");
        ll.insertAtTail("A");
        ll.insertAtHead("B");
        assertEquals(4,ll.getLength());
        assertEquals("(B, S, D, A)",ll.toString());

    }

    @Test
    //Test insertAtHead to a one element LinkedList by inserting the
identical element
    //The content and length will change.
    //Should insert at the beginning instead of unable to insert
    public void testInsertHeadSameOne(){
        LinkedList ll = new LinkedList();
        ll.insertAtTail("D");
        ll.insertAtHead("D");
        assertEquals(2,ll.getLength());
        assertEquals("(D, D)",ll.toString());

    }

    @Test
    //Test insertAtHead of a LinkedList with multiple identical elements.
Content and length should change
    public void testInsertHeadSameMultiple(){
        LinkedList ll = new LinkedList();
```

```
        ll.insertAtTail("D");
        ll.insertAtTail("D");
        ll.insertAtTail("D");
        ll.insertAtTail("D");
        ll.insertAtTail("D");
        ll.insertAtHead("D");
        assertEquals(6,ll.getLength());
        assertEquals("(D, D, D, D, D, D)",ll.toString());


    }

    @Test
    //Test isEmpty of an empty LinkedList. Return true
    public void testIsEmptyTrue(){
        LinkedList ll = new LinkedList();
        assertTrue(ll.isEmpty());

    }

    @Test
    //Test isEmpty of a non-empty LinkedList. Should return false
    public void testIsEmptyFalse(){
        LinkedList ll = new LinkedList();
        ll.insertAtHead("F");
        ll.insertAtTail("D");
        assertFalse(ll.isEmpty());

    }

    @Test
    //Test getLength of an empty LinkedList. Return 0
    public void testGetLengthEmpty(){
        LinkedList ll = new LinkedList();
        assertEquals(0,ll.getLength());
    }

    @Test
    //Test getLength of a non-empty LinkedList. Return the length
    public void testGetLengthNonEmpty(){
        LinkedList ll = new LinkedList();
        ll.insertAtHead("A");
        ll.insertAtTail("R");
        assertEquals(2,ll.getLength());
    }

    @Test
    //Test toString of empty LinkedList.
    public void testToStringEmpty(){
        LinkedList ll = new LinkedList();
        assertEquals("()",ll.toString());

    }
```

```
    @Test
    //Test toString of multiple elements LinkedList
    public void testToStringMultiple(){
        LinkedList ll = new LinkedList();
        ll.insertAtTail("A");
        ll.insertAtTail("D");
        ll.insertAtHead("F");
        assertEquals("(F, A, D)",ll.toString());
    }

    @Test
    //Test toString of one element LinkedList
    public void testToStringOne(){
        LinkedList ll = new LinkedList();

        ll.insertAtHead("F");
        assertEquals("(F)",ll.toString());
    }


    @Test
    //test getItemAt valid position. Return the data corresponding to the
wanted index
    public void testGetItemAtNonEmptyValid(){
        LinkedList ll = new LinkedList();
        ll.insertAtHead("A");
        ll.insertAtTail("B");
        ll.insertAtTail("C");
        ll.insertAtTail("D");
        ll.insertAtTail("E");
        ll.insertAtTail("F");
        assertEquals("F",ll.getItemAt(5));
        assertEquals("B",ll.getItemAt(1));
        assertEquals("A",ll.getItemAt(0));
        assertEquals("C",ll.getItemAt(2));
        assertEquals("D",ll.getItemAt(3));
        assertEquals("E",ll.getItemAt(4));

    }

    @Test
    //test getItemAt negative position. Return null
    public void testGetItemAtNegative(){
        LinkedList ll = new LinkedList();
        ll.insertAtHead("A");
        ll.insertAtTail("B");
        ll.insertAtTail("C");
        ll.insertAtTail("D");
        ll.insertAtTail("E");
        ll.insertAtTail("F");
        assertNull(ll.getItemAt(-1));
```

```
    }


    @Test
    //test getItemAt position > the length of LinkedList. Return null
    public void testGetItemAtGreaterThanLength(){
        LinkedList ll = new LinkedList();
        ll.insertAtHead("A");
        ll.insertAtTail("B");
        ll.insertAtTail("C");
        ll.insertAtTail("D");
        ll.insertAtTail("E");
        ll.insertAtTail("F");
        assertNull(ll.getItemAt(10));
    }

    @Test
    //test SearchItemAt an empty LinkedList. Return null no matter the wanted
 position
    public void testGetItemAtEmpty(){
        LinkedList ll = new LinkedList();
        assertNull(ll.getItemAt(0));

    }
    @Test
    //Test removeTail of a non empty linkedlist. After removing the length
will decrement by 1
    public void testRemoveTailNonEmpty(){
        LinkedList ll = new LinkedList();
        ll.insertAtHead("A");
        ll.insertAtTail("B");
        ll.insertAtTail("C");
        ll.insertAtTail("D");
        ll.removeTail();
        assertEquals("(A, B, C)",ll.toString());
        assertEquals(3,ll.getLength());

    }

    @Test
    //Test removeTail of an empty linkedlist. Should remain the same after
removing
    public void testRemoveTailEmpty(){
        LinkedList ll = new LinkedList();
        ll.removeTail();
        assertEquals("()",ll.toString());
        assertEquals(0,ll.getLength());
    }

    @Test
    //Test removeTail One Element. After removing the linkedlist becomes
empty
    public void testRemoveTailOne(){
```

```
        LinkedList ll = new LinkedList();
        ll.insertAtHead("A");
        ll.removeTail();
        assertEquals("()",ll.toString());
        assertEquals(0,ll.getLength());
    }

    @Test
    //Test remove at index of a non-empty linkedlist with valid index. After
removing, the length will decrement by 1
    public void testRemoveAtIndexValidNonEmpty(){
        LinkedList ll = new LinkedList();
        ll.insertAtHead("A");
        ll.insertAtTail("B");
        ll.insertAtTail("C");
        ll.insertAtTail("D");
        ll.removeAtIndex(1);
        assertEquals(3,ll.getLength());
        assertEquals("C",ll.getItemAt(1));
        assertEquals("(A, C, D)",ll.toString());
        ll.removeAtIndex(2);
        assertEquals("(A, C)",ll.toString());
        ll.removeAtIndex(0);
        assertEquals("(C)",ll.toString());
        ll.removeAtIndex(0);
        assertTrue(ll.isEmpty());
    }

    @Test
    //Test remove at index of an empty linkedlist no matter the index is.
Should remain the same

    public void testRemoveAtIndexEmpty(){
        LinkedList ll = new LinkedList();
        ll.removeAtIndex(10);
        assertEquals("()",ll.toString());
        assertEquals(0,ll.getLength());
    }

    @Test

    //Remove at index of one-element linkedlist. After removing, the
linkedlist becomes empty
    public void testRemoveAtIndexOne(){
        LinkedList ll = new LinkedList();
        ll.insertAtHead("A");
        ll.removeAtIndex(0);
        assertTrue(ll.isEmpty());
    }

    @Test
    //Remove at index with invalid index (negative integer). Should remain
the same
```

```
    public void testRemoveAtIndexNegative(){
        LinkedList ll = new LinkedList();
        ll.insertAtHead("A");
        ll.insertAtTail("B");
        ll.insertAtTail("C");
        ll.insertAtTail("D");
        ll.removeAtIndex(-1);
        assertEquals("(A, B, C, D)",ll.toString());
    }

    @Test
    //Remove at index with index > the linkedlist.length. Should remain the
same

    public void testRemoveAtIndexGreaterThan(){
        LinkedList ll = new LinkedList();
        ll.insertAtHead("A");
        ll.insertAtTail("B");
        ll.insertAtTail("C");
        ll.insertAtTail("D");
        ll.removeAtIndex(10);
        assertEquals("(A, B, C, D)",ll.toString());
    }

    @Test
    //Remove at index with index == 0. Remove the head of the linkedlist
    public void testRemoveAtIndexHead(){
        LinkedList ll = new LinkedList();
        ll.insertAtHead("A");
        ll.insertAtTail("B");
        ll.removeAtIndex(0);
        assertEquals("(B)",ll.toString());
    }

    @Test
    //Remove at index with index at the end of the linkedlist. Remove the
tail of the linkedlist
    public void testRemoveAtIndexTail(){
        LinkedList ll = new LinkedList();
        ll.insertAtHead("A");
        ll.insertAtTail("B");
        ll.removeAtIndex(1);
        assertEquals("(A)",ll.toString());
    }

    @Test
    //Remove at index with index in the middle of the linkedlist. Remove the
middle element of the linkedlist
    public void testRemoveAtIndexMiddle(){
        LinkedList ll = new LinkedList();
        ll.insertAtHead("A");
        ll.insertAtTail("B");
```

```
        ll.insertAtTail("C");
        ll.insertAtTail("D");
        ll.insertAtTail("E");
        ll.removeAtIndex(2);
        assertEquals("(A, B, D, E)",ll.toString());
    }




    @Test
    //Insert at index of a non empty linkedlist with valid index. The element
 will be inserted at the wanted index
    public void insertAtIndexValidNonEmpty(){
        LinkedList ll = new LinkedList();
        ll.insertAtHead("A");
        ll.insertAtTail("B");
        ll.insertAtTail("C");
        ll.insertAtTail("D");
        ll.insertAtIndex("E",1);
        assertEquals("(A, E, B, C, D)",ll.toString());
    }

    @Test
    //Insert at index of an empty linkedlist at no matter the index is. The
linkedlist will have one element
    public void insertAtIndexEmpty(){
        LinkedList ll = new LinkedList();
        ll.insertAtIndex("A",-1);
        assertEquals("(A)",ll.toString());
    }

    @Test
    //Insert at index with negative index of a nonempty linkedlist. Should
remain the same

    public void insertAtIndexNegative(){
        LinkedList ll = new LinkedList();
        ll.insertAtHead("Hello");
        ll.insertAtTail("Goodbye");
        ll.insertAtIndex("Hi",-1);
        assertEquals("(Hello, Goodbye)",ll.toString());
    }

    @Test
    //insert at index greater than linkedlist.length. Should remain the same

    public void insertAtIndexGreater(){
        LinkedList ll = new LinkedList();
        ll.insertAtHead("Hello");
        ll.insertAtTail("Goodbye");
```

```
        ll.insertAtIndex("Hi",10);
        assertEquals("(Hello, Goodbye)",ll.toString());
    }



    @Test
    //Insert at index with index == 0. Insert at the head of the linkedlist
    public void testInsertAtIndexHead(){
        LinkedList ll = new LinkedList();
        ll.insertAtHead("A");
        ll.insertAtTail("B");
        ll.insertAtIndex("C",0);
        assertEquals("(C, A, B)",ll.toString());
    }

    @Test
    //Insert at index with index at the end of the linkedlist. Insert at the
tail of the linkedlist
    public void testInsertAtIndexTail(){
        LinkedList ll = new LinkedList();
        ll.insertAtHead("A");
        ll.insertAtTail("B");
        ll.insertAtIndex("C",2);
        assertEquals("(A, B, C)",ll.toString());
    }

    @Test
    //Remove at index with index in the middle of the linkedlist. Remove the
middle element of the linkedlist
    public void testInsertAtIndexMiddle(){
        LinkedList ll = new LinkedList();
        ll.insertAtHead("A");
        ll.insertAtTail("B");
        ll.insertAtTail("C");
        ll.insertAtTail("D");
        ll.insertAtTail("E");
        ll.insertAtIndex("F",2);
        assertEquals("(A, B, F, C, D, E)",ll.toString());
    }

    @Test
    //remove tail of a non-empty linkedlist with same elements multiple times
. Length and contents will change
    public void testRemoveTailSameMultiple(){
        LinkedList ll = new LinkedList();
        ll.insertAtHead("B");
        ll.insertAtTail("B");
        ll.insertAtTail("B");
        ll.insertAtTail("B");
        ll.insertAtTail("B");
        ll.insertAtTail("B");
        ll.insertAtTail("B");
        ll.insertAtTail("B");
```

```
        ll.removeTail();
        ll.removeTail();
        ll.removeTail();
        assertEquals("(B, B, B, B)",ll.toString());
        assertEquals(4,ll.getLength());
    }

    @Test
    //remove tail of 2 same element linkedlist. Length and contents will
change
    public void testRemoveTailSameTwice(){
        LinkedList ll = new LinkedList();
        ll.insertAtHead("A");
        ll.insertAtTail("A");
        ll.removeTail();

        assertEquals("(A)",ll.toString());
        assertEquals(1,ll.getLength());
    }

    @Test
    //remove tail multiple times of a non-empty linkedlist. Contents and
length will change
    public void testRemoveTailMultiple(){
        LinkedList ll = new LinkedList();
        ll.insertAtHead("B");
        ll.insertAtTail("D");
        ll.insertAtTail("E");
        ll.insertAtTail("F");
        ll.insertAtTail("G");
        ll.insertAtTail("H");
        ll.insertAtTail("I");
        ll.removeTail();
        ll.removeTail();
        ll.removeTail();
        assertEquals("(B, D, E, F)",ll.toString());
        assertEquals(4,ll.getLength());

    }









}
```

```java
/**
 * JUnit test class.  Use these tests as models for your own.
 */
import org.junit.*;


import org.junit.rules.Timeout;

import static org.junit.Assert.*;

import proj3.Sequence;




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);

    /**
     * customize the sequence from the string array and be able to set the
capacity and current index
     * @param newArray the string array given so that convert to sequence
     * @param capacity the capacity wanted in the sequence
     * @param currentIndex the current index for the element in the sequence
     * @return the customized sequence
     */

    private Sequence createSequence(String[] newArray, int capacity, int
currentIndex){
        Sequence sq = new Sequence(capacity);
        for(String element:newArray){
            sq.addAfter(element);
        }
        sq.start();
        for(int i = 0; i < currentIndex; i++){
            sq.advance();
        }
        return sq;

    }


    @Test
    //Test  default constructor
    public void test_DefaultConstructor_EmptySequence(){
        Sequence sq = new Sequence();
        String toStringExpected = "{} (capacity = 10)";
        assertEquals(toStringExpected,sq.toString());
        assertEquals(10,sq.getCapacity());
        assertNull(sq.getCurrent());
```

```
        assertEquals(0,sq.size());

    }


    @Test
    //Test non-default constructor with positive initial capacity
    public void test_NonDefaultConstructor_EmptySequence_PositiveCap(){
        Sequence sq = new Sequence(22);
        String toStringExpected = "{} (capacity = 22)";
        assertEquals(toStringExpected,sq.toString());
        assertEquals(22,sq.getCapacity());
        assertNull(sq.getCurrent());
        assertEquals(0,sq.size());

    }

    @Test
    //Test non-default constructor with negative initial capacity. Then the
capacity set to 10
    public void test_NonDefaultConstructor_EmptySequence_NegativeCap(){
        Sequence sq = new Sequence(-3);
        String toStringExpected = "{} (capacity = 10)";
        assertEquals(toStringExpected,sq.toString());
        assertEquals(10,sq.getCapacity());
        assertNull(sq.getCurrent());
        assertEquals(0,sq.size());

    }


    @Test
    //Test addBefore to empty sequence.
    // Then the element added is the first element in the sequence and also
the current element
    public void test_AddBefore_EmptySequence(){
        Sequence sq = new Sequence();


        sq.addBefore("A");

        String toStringExpected = "{>A} (capacity = 10)";
        assertEquals(toStringExpected,sq.toString());
        assertEquals("A",sq.getCurrent());
        assertEquals(10,sq.getCapacity());
        assertEquals(1,sq.size());

    }

    @Test
    //Test addBefore to a full sequence.
    //Then the sequence's current capacity will expand twice plus 1
    //The size should +1, the added element become the current element
```

```
    public void test_AddBefore_FullSequence(){
        Sequence sq = createSequence(new String[]{"A","B","C"},3,3);
        sq.addBefore("D");
        assertEquals(4,sq.size());
        assertEquals(7,sq.getCapacity());
        assertEquals("D",sq.getCurrent());
        String toStringExpected = "{>D, A, B, C} (capacity = 7)";
        assertEquals(toStringExpected,sq.toString());
    }

    @Test
    //Test addBefore to a sequence that has room so capacity not change. The
added element becomes the current
    public void test_AddBefore_NonEmptySequence_ThatHasRoom(){
        Sequence sq = createSequence(new String[]{"A","B","C"},8,1);
        sq.addBefore("D");
        assertEquals(4,sq.size());
        assertEquals(8,sq.getCapacity());
        assertEquals("D",sq.getCurrent());
        String toStringExpected = "{A, >D, B, C} (capacity = 8)";
        assertEquals(toStringExpected,sq.toString());

    }

    @Test

    //add an element to an almost full sequence. The size should increase by
1, capacity stay the same

    public void test_AddBefore_AlmostFull(){
        Sequence sq = createSequence(new String[]{"A", "B", "C","D","E"},6,4
);
        sq.addBefore("F");
        assertEquals(6,sq.size());
        assertEquals(6,sq.getCapacity());
        assertEquals("F",sq.getCurrent());
        String toStringExpected = "{A, B, C, D, >F, E} (capacity = 6)";
        assertEquals(toStringExpected,sq.toString());

    }

    @Test
    //test add before to a sequence with no current. Then the added element
become the current at the first
    public void test_AddBefore_NoCurrent(){

        Sequence sq = createSequence(new String[]{"A","B","C"},5,3);
        sq.addBefore("D");
        assertEquals(4,sq.size());
        assertEquals(5,sq.getCapacity());
        assertEquals("D",sq.getCurrent());
        String toStringExpected = "{>D, A, B, C} (capacity = 5)";
        assertEquals(toStringExpected,sq.toString());
```

```
    }


    @Test
    //Test addAfter to an empty sequence
    //Then the element added is the end of the sequence and also the current
element
    public void test_AddAfter_EmptySequence(){
        Sequence sq = new Sequence();
        sq.addAfter("A");
        String toStringExpected = "{>A} (capacity = 10)";
        assertEquals(toStringExpected,sq.toString());
        assertEquals("A",sq.getCurrent());
        assertEquals(10,sq.getCapacity());
        assertEquals(1,sq.size());
    }

    @Test
    //Test addAfter to a full sequence
    //Then the sequence's current capacity is expanded twice plus 1
    public void test_AddAfter_FullSequence(){
        Sequence sq = createSequence(new String[]{"A", "B", "C", "D"},4,4);
        sq.addAfter("F");

        assertEquals("F",sq.getCurrent());
        assertEquals(9,sq.getCapacity());
        assertEquals(5,sq.size());

        String toStringExpected = "{A, B, C, D, >F} (capacity = 9)";
        assertEquals(toStringExpected, sq.toString());
    }

    @Test

    //Test add after to non empty sequence that has room. Capacity should not
 change, size should + 1

    public void test_AddAfter_NonEmptySequence_ThatHasRoom(){

        Sequence sq = createSequence(new String[]{"A", "B", "C", "D"},6,1);
        sq.addAfter("F");

        assertEquals("F",sq.getCurrent());
        assertEquals(6,sq.getCapacity());
        assertEquals(5,sq.size());

        String toStringExpected = "{A, B, >F, C, D} (capacity = 6)";
        assertEquals(toStringExpected, sq.toString());

    }

    @Test
```

```
    //Should be added at the end. The added is the current
    public void test_AddAfter_NoCurrent(){

        Sequence sq = createSequence(new String[]{"A", "B", "C", "D"},6,4);
        sq.addAfter("F");

        assertEquals("F",sq.getCurrent());
        assertEquals(6,sq.getCapacity());
        assertEquals(5,sq.size());

        String toStringExpected = "{A, B, C, D, >F} (capacity = 6)";
        assertEquals(toStringExpected, sq.toString());


    }

    @Test

    //The capacity should not change. Size + 1

    public void test_AddAfter_AlmostFull(){

        Sequence sq = createSequence(new String[]{"A", "B", "C", "D"},5,4);
        sq.addAfter("F");

        assertEquals("F",sq.getCurrent());
        assertEquals(5,sq.getCapacity());
        assertEquals(5,sq.size());

        String toStringExpected = "{A, B, C, D, >F} (capacity = 5)";
        assertEquals(toStringExpected, sq.toString());

    }

    @Test
    //return false to an empty default sequence. Return true after adding
    public void test_IsCurrent_EmptyDefault(){
        Sequence sq = new Sequence();
        assertFalse(sq.isCurrent());
        sq.addAfter("A");
        assertTrue(sq.isCurrent());


    }


    @Test
    //Test isCurrent to a sequence after advancing the last current to no
current.Return false
    public void test_IsCurrent_Advancing_LastCurrent(){
        Sequence sq = createSequence(new String[]{"A","B","C"},5,2);
        assertTrue(sq.isCurrent());
        sq.advance();
```

```
        assertFalse(sq.isCurrent());

    }



    @Test
    //Test getCapacity to a default sequence. Return 10
    public void test_GetCapacity_DefaultSequence(){
        Sequence sq = new Sequence();
        assertEquals(10,sq.getCapacity());

    }

    @Test
    //Test getCapacity to a non-default sequence. Return the capacity
    public void test_GetCapacity_NonDefaultSequence(){
        Sequence sq = new Sequence(12);
        assertEquals(12,sq.getCapacity());
    }

    @Test
    //return null if sequence is empty
    public void test_GetCurrent_Empty(){
        Sequence sq = new Sequence();
        assertNull(sq.getCurrent());

    }

    @Test
    //return null if the non empty sequence has no current
    public void test_GetCurrent_NonEmpty_NoCurrent(){
        Sequence sq = createSequence(new String[]{"A","B","C"},4,3);
        assertNull(sq.getCurrent());


    }

    @Test
    //return the element of the non empty sequence
    public void test_GetCurrent_NonEmpty_Current(){
        Sequence sq = createSequence(new String[]{"A","B","C"},4,1);
        assertEquals("B",sq.getCurrent());
    }

    @Test
    //Test ensureCapacity of a sequence that has capacity smaller than
minCapacity.
    //Increase the capacity equal to minCapacity
    //Contents don't change
    public void test_EnsureCapacity_Smaller(){
        Sequence sq = createSequence(new String[]{"A","B","C"},10,1);
        sq.ensureCapacity(1000);
```

```
        assertEquals(1000,sq.getCapacity());
        String toStringExpected = "{A, >B, C} (capacity = 1000)";
        assertEquals(toStringExpected,sq.toString());
        assertEquals(3,sq.size());
        assertEquals("B",sq.getCurrent());

    }

    @Test
    //Test ensureCapacity of a sequence that has capacity greater than
minCapacity
    //Do nothing to the capacity
    //Contents don't change
    public void test_EnsureCapacity_Greater(){

        Sequence sq = createSequence(new String[]{"A","B","C"},1000,1);
        sq.ensureCapacity(10);
        assertEquals(1000,sq.getCapacity());
        String toStringExpected = "{A, >B, C} (capacity = 1000)";
        assertEquals(toStringExpected,sq.toString());
        assertEquals(3,sq.size());
        assertEquals("B",sq.getCurrent());

    }

    @Test
    //Test ensureCapacity of a sequence that has capacity equal to
minCapacity
    //Do nothing to the capacity
    //Contents don't change
    public void test_EnsureCapacity_Equal(){

        Sequence sq = createSequence(new String[]{"A","B","C"},1000,1);
        sq.ensureCapacity(1000);
        assertEquals(1000,sq.getCapacity());
        String toStringExpected = "{A, >B, C} (capacity = 1000)";
        assertEquals(toStringExpected,sq.toString());
        assertEquals(3,sq.size());
        assertEquals("B",sq.getCurrent());

    }

    @Test
    //Test addAll of an empty sequence to non empty sequence.
    //If this.sequence is empty and the other is not, then return this.
sequence with contents of the other sequence
    //The size of this.sequence must include the size of the other sequence.
Capacity not change
    //BUT when call isCurrent of this.sequence it's still false, getCurrent
other sequence it's still the same element


    public void test_AddAll_Empty_To_NonEmpty_Current(){
```

```
        Sequence sq = new Sequence();
        Sequence toAddAllSq = createSequence(new String[]{"A","B"},3,0);
        sq.addAll(toAddAllSq);
        assertEquals(2,sq.size());

        assertEquals("A",toAddAllSq.getCurrent());
        assertFalse(sq.isCurrent());
        String toStringExpected = "{A, B} (capacity = 10)";
        assertEquals(toStringExpected, sq.toString());

    }

    @Test

    //Test addAll of an empty sequence to non empty sequence without current.
    //If this.sequence is empty and the other is not, then return this.
sequence with contents of the other sequence
    //The size of this.sequence must include the size of the other sequence.
Capacity not change
    //BUT when call isCurrent of this.sequence it's still false, of the other
 sequence it's still false


    public void test_AddAll_Empty_To_NonEmpty_NoCurrent(){

        Sequence sq = new Sequence();
        Sequence toAddAllSq = createSequence(new String[]{"A","B"},3,2);
        sq.addAll(toAddAllSq);
        assertEquals(2,sq.size());

        assertFalse(toAddAllSq.isCurrent());
        assertFalse(sq.isCurrent());
        String toStringExpected = "{A, B} (capacity = 10)";
        assertEquals(toStringExpected, sq.toString());


    }

    @Test
    //Test addAll of non empty sequence to empty sequence.
    //Return this.sequence with contents of the other sequence. In this case
, contents don't change
    //The size of this.sequence must include the size of the other sequence.
Capacity not change
    //BUT when call isCurrent of other.sequence it's still false, getCurrent
this.sequence it's still the same element



    public void test_AddAll_NonEmpty_Current_To_Empty(){
```

```
        Sequence toAddAllSq = new Sequence();
        Sequence sq = createSequence(new String[]{"A","B"},3,1);
        sq.addAll(toAddAllSq);
        assertEquals(2,sq.size());


        assertEquals("B",sq.getCurrent());

        String toStringExpected = "{A, >B} (capacity = 3)";
        assertEquals(toStringExpected, sq.toString());

    }

    @Test
    //Test addAll of non empty sequence to empty sequence without current.
    //Return this.sequence with contents of the other sequence. In this case
, contents don't change
    //The size of this.sequence must include the size of the other sequence.
Capacity not change
    //BUT when call isCurrent of other.sequence it's still false, of this.
sequence it's still false


    public void test_AddAll_NonEmpty_NoCurrent_To_Empty(){

        Sequence toAddAllSq = new Sequence();
        Sequence sq = createSequence(new String[]{"A","B"},3,2);
        sq.addAll(toAddAllSq);
        assertEquals(2,sq.size());


        assertFalse(sq.isCurrent());

        String toStringExpected = "{A, B} (capacity = 3)";
        assertEquals(toStringExpected, sq.toString());
    }

    @Test
    //Test addAll of both empty sequences.
    // Then print out empty contents and both sequences when call isCurrent
still false

    public void test_AddAll_BothEmpty(){
        Sequence sq = new Sequence(3);
        Sequence toAddAllSq = new Sequence();
        sq.addAll(toAddAllSq);
        assertFalse(sq.isCurrent());

        assertEquals(0,sq.size());

        assertEquals(3,sq.getCapacity());
        String expected = "{} (capacity = 3)";
        String expectedToAdd = "{} (capacity = 10)";
```

```
        assertEquals(expected,sq.toString());
        assertEquals(expectedToAdd,toAddAllSq.toString());

    }

    @Test
    //Test addAll of two filled sequences but adding all the elements of
other.sequence will result in this.sequence
    // expanding just enough room (the capacity is changed). Then the
contents are both the elements from this and
    // other sequence and the current elements are the same

    public void test_AddAll_NotEnoughCapacity(){
        Sequence sq = createSequence(new String[]{"A","B"},2,1);
        Sequence toAddAllSq = createSequence(new String[]{"C","D","E"},3,0);
        sq.addAll(toAddAllSq);
        String expected = "{A, >B, C, D, E} (capacity = 5)";
        assertEquals(expected, sq.toString());
        assertEquals(5,sq.size());
        assertEquals(5,sq.getCapacity());
        assertEquals("C",toAddAllSq.getCurrent());
        assertEquals(3,toAddAllSq.size());
        assertEquals(3,toAddAllSq.getCapacity());

    }


    @Test
    //other.sequence has repeated contents to this.sequence. But still add
all to this.sequence anyway.
    // Current elements don't change
    public void test_AddAll_Repeat_One(){
        Sequence sq = createSequence(new String[]{"A"},5,1);
        Sequence toAddAllSq = createSequence(new String[]{"A","B","C","D"},4,
3);
        sq.addAll(toAddAllSq);
        assertEquals(5,sq.size());
        assertEquals("D",toAddAllSq.getCurrent());
        String expected = "{A, A, B, C, D} (capacity = 5)";
        assertEquals(expected,sq.toString());


    }


    @Test
    //other.sequence has repeated contents to this.sequence. But still add
all to this.sequence anyway.
    // Current elements don't change
    public void test_AddAll_Repeat_Two(){
        Sequence sq = createSequence(new String[]{"A","B","C"},5,1);
        Sequence toAddAllSq = createSequence(new String[]{"A","B"},3,2);
        sq.addAll(toAddAllSq);
```

```
        assertEquals(5,sq.size());
        String expected = "{A, >B, C, A, B} (capacity = 5)";
        assertEquals(expected,sq.toString());
        assertFalse(toAddAllSq.isCurrent());

    }


    @Test
    //other.sequence has same contents to this.sequence. But still add all to
 this.sequence anyway.
    // Current elements don't change. They are different objects
    public void test_AddAll_Repeat_All(){
        Sequence sq = createSequence(new String[]{"A","B"},5,1);
        Sequence toAddAllSq = createSequence(new String[]{"A","B"},5,1);
        sq.addAll(toAddAllSq);
        assertEquals(4,sq.size());
        String expected = "{A, >B, A, B} (capacity = 5)";
        assertEquals(expected,sq.toString());
        assertEquals("B",toAddAllSq.getCurrent());
        toAddAllSq.start();
        assertNotEquals(sq.getCurrent(),toAddAllSq.getCurrent());
        assertNotSame(sq,toAddAllSq);

    }

    @Test
    //Add all two non empty sequences without current element. The size
should be the sum of two sequences
    //After add two sequences still have no current elements
    public void test_AddAll_NonEmpty_NoCurrent(){

        Sequence sq = createSequence(new String[]{"A","B"},5,2);
        Sequence toAddAllSq = createSequence(new String[]{"C","D"},3,2);
        sq.addAll(toAddAllSq);
        assertEquals(4,sq.size());
        String expected = "{A, B, C, D} (capacity = 5)";
        assertEquals(expected,sq.toString());
        assertFalse(toAddAllSq.isCurrent());
        assertFalse(sq.isCurrent());


    }

    @Test

    //Add all two non empty sequences without current element. The size
should be the sum of two sequences
    //After add two sequences still have their original current elements

    public void test_AddAll_NonEmpty_Current(){
        Sequence sq = createSequence(new String[]{"A","B"},5,0);
        Sequence toAddAllSq = createSequence(new String[]{"C","D"},3,1);
```

```
        sq.addAll(toAddAllSq);
        assertEquals(4,sq.size());
        String expected = "{>A, B, C, D} (capacity = 5)";
        assertEquals(expected,sq.toString());
        assertEquals("D",toAddAllSq.getCurrent());
        assertEquals("A",sq.getCurrent());

    }


    @Test
    //test Advance empty sequence. Nothing change
    public void test_Advance_Empty(){
        Sequence sq = new Sequence();
        sq.advance();
        assertEquals(10,sq.getCapacity());
        assertEquals(0,sq.size());
        assertFalse(sq.isCurrent());
        String toStringExpected = "{} (capacity = 10)";
        assertEquals(toStringExpected,sq.toString());

    }

    @Test
    //Test advance a non empty sequence with current. After advancing the
current element change to the next one
    public void test_Advance_Current(){
        Sequence sq = createSequence(new String[]{"A","B","C","D"},4,0);
        sq.advance();
        assertEquals("B",sq.getCurrent());
        String toStringExpected = "{A, >B, C, D} (capacity = 4)";

        assertEquals(toStringExpected,sq.toString());

        sq.advance();
        assertEquals("C",sq.getCurrent());
        String expected = "{A, B, >C, D} (capacity = 4)";
        assertEquals(expected,sq.toString());

        sq.advance();
        assertEquals("D",sq.getCurrent());
        String answer = "{A, B, C, >D} (capacity = 4)";
        assertEquals(answer,sq.toString());


    }

    @Test
    //Test advance with a non empty sequence without current element. Call
isCurrent return false
    public void test_Advance_NoCurrent(){
        Sequence sq = createSequence(new String[]{"A","B","C","D"},4,4);
        sq.advance();
```

```
        assertFalse(sq.isCurrent());
        String toStringExpected = "{A, B, C, D} (capacity = 4)";
        assertEquals(toStringExpected,sq.toString());

    }



    @Test
    //Test advance with a sequence whose current element is at the end. Then
advancing causes no current element (null)
    public void test_Advance_Last(){
        Sequence sq = createSequence(new String[]{"A","B","C","D"},4,3);
        sq.advance();
        assertFalse(sq.isCurrent());
        String toStringExpected = "{A, B, C, D} (capacity = 4)";
        assertEquals(toStringExpected,sq.toString());


    }



    @Test
    //Test clone with an empty sequence. Then the clone has the same contents
 as the original
    public void test_Clone_EmptySequence(){
        Sequence sq = new Sequence();
        Sequence toCloneSq = sq.clone();
        assertEquals(10,toCloneSq.getCapacity());
        assertEquals(0,toCloneSq.size());
        assertFalse(toCloneSq.isCurrent());
        String expected = "{} (capacity = 10)";
        assertEquals(expected,toCloneSq.toString());

    }

    @Test
    //Test clone of a non empty sequence. Then the current elements of both
sequences are the same position
    //When make changes to clone or original, the other one doesn't change

    public void test_Clone_Sequence_DifferentObjects(){
        Sequence sq = createSequence(new String[]{"A","B","C","D","E"},10,3);
        Sequence toCloneSq = sq.clone();
        assertNotSame(sq,toCloneSq);
        sq.addAfter("F");
        //assertNotEquals(sq.toString(),toCloneSq.toString());

        toCloneSq = sq.clone();
        toCloneSq.addAfter("F");
        assertNotSame(sq,toCloneSq);
```

```
    }

    @Test
    //Cloning should produce identical contents: same size, capacity, current
 element
    public void test_Clone_Identical(){
        Sequence sq = createSequence(new String[]{"A","B","C","D","E"},10,3);
        Sequence toCloneSq = sq.clone();
        assertTrue(sq.equals(toCloneSq));

    }

    @Test
    //Test removeCurrent of non empty sequence with no current element.
    //When getCurrent it is null
    public void test_Remove_NoCurrent(){
        Sequence sq = createSequence(new String[]{"A","B"},3,2);
        sq.removeCurrent();
        assertNull(sq.getCurrent());
    }

    @Test
    //Test removeCurrent of an empty sequence which means there is no current
 element.
    //When getCurrent it is null
    public void test_Remove_EmptySequence(){
        Sequence sq = new Sequence();
        sq.removeCurrent();
        assertNull(sq.getCurrent());

    }

    @Test
    //Test removeCurrent of a non empty sequence whose current element is at
the end.
    //Then after remove call getCurrent it is null
    public void test_Remove_Last(){
        Sequence sq = createSequence(new String[]{"A","B","C"},4,2);
        sq.removeCurrent();
        assertNull(sq.getCurrent());

    }

    @Test
    //Test removeCurrent of a non empty sequence whose current element is NOT
 at the end.
    //Then the next element is the current element
    public void test_Remove_First(){
        Sequence sq = createSequence(new String[]{"A","B","C"},4,0);
        sq.removeCurrent();
        assertEquals("B",sq.getCurrent());

    }
```

```
    @Test
    //Test removeCurrent of a non empty sequence whose current element is NOT
 at the end.
    //Then the next (last) element is the current element
    public void test_Remove_Middle(){
        Sequence sq = createSequence(new String[]{"A","B","C"},4,1);
        sq.removeCurrent();
        assertEquals("C",sq.getCurrent());
    }

    @Test
    //Test the number of element in an empty default sequence. Return 0

    public void test_Size_Default(){
        Sequence sq = new Sequence();
        assertEquals(0,sq.size());

    }

    @Test
    //Test the number of element in an empty non default sequence. Return 0

    public void test_Size_NonDefault(){
        Sequence sq = new Sequence();
        assertEquals(0,sq.size());

    }

    @Test
    //Test the number of element in a non empty sequence. Return that number

    public void test_Size_NonEmpty(){
        Sequence sq = createSequence(new String[]{"A","B"},3,1);
        assertEquals(2,sq.size());

    }

    @Test
    //Test start of an empty sequence. Then do nothing because it has no
current element.
    //When call isCurrent return false
    public void test_Start_EmptySequence(){

        Sequence sq = new Sequence();
        sq.start();
        assertFalse(sq.isCurrent());


    }

    @Test
    //Test start of a filled sequence. Then the first element of the sequence
```

```
is the current element
    public void test_Start_NonEmpty_Current(){

        Sequence sq = createSequence(new String[]{"A","B","C"},5,1);
        sq.start();
        assertEquals("A",sq.getCurrent());
        String toStringExpected = "{>A, B, C} (capacity = 5)";
        assertEquals(toStringExpected,sq.toString());
    }

    @Test
    //Test start of a filled sequence. Then the first element of the sequence
is the current element

    public void test_Start_NonEmpty_NoCurrent(){

        Sequence sq = createSequence(new String[]{"A","B","C"},5,3);
        sq.start();
        assertEquals("A",sq.getCurrent());
        String toStringExpected = "{>A, B, C} (capacity = 5)";
        assertEquals(toStringExpected,sq.toString());

    }

    @Test
    //Test trimToSize of an empty sequence. Then capacity = 0
    public void test_TrimToSize_EmptySequence(){
        Sequence sq = new Sequence();
        sq.trimToSize();
        assertEquals(0,sq.getCapacity());

    }

    @Test
    //Test trimToSize of a filled sequence. Reduce the capacity to its actual
size

    public void test_TrimToSize_NonEmpty(){

        Sequence sq = createSequence(new String[]{"A","B"},10,1);
        sq.trimToSize();
        assertEquals(2,sq.getCapacity());

    }

    @Test
    //Test toString of an empty sequence. Return {} followed by its capacity
    public void test_ToString_Empty(){
        Sequence sq = new Sequence();
        String expected = "{} (capacity = 10)";
        assertEquals(expected,sq.toString());

    }
```

```
    @Test
    //Test toString of a sequence with no current element
    public void test_ToString_NoCurrent(){
        Sequence sq = createSequence(new String[]{"A","D"},3,2);
        String expected = "{A, D} (capacity = 3)";
        assertEquals(expected,sq.toString());

    }

    @Test
    //Test toString of a sequence with current element.

    public void test_ToString_Current(){
        Sequence sq = createSequence(new String[]{"A","D"},3,1);
        String expected = "{A, >D} (capacity = 3)";
        assertEquals(expected,sq.toString());

    }

    @Test
    //should be equal if they both have no current, same cap, same size same
order
    public void test_Equals_NoCurrent(){
        Sequence sq = createSequence(new String[]{"A","B","D"},3,3);
        Sequence anotherSq = createSequence(new String[]{"A","B","D"},3,3);
        assertTrue(sq.equals(anotherSq));


    }

    @Test
    //Test equals of two empty sequences. Return true
    public void test_Equals_TwoEmpty_Sequences(){
        Sequence sq = new Sequence();
        Sequence anotherSq = new Sequence(12);
        assertTrue(sq.equals(anotherSq));


    }

    @Test
    //empty sequence != non-empty sequence
    public void test_Equals_EmptyVsNonEmpty(){
        Sequence sq = new Sequence();
        Sequence anotherSq = createSequence(new String[]{"A","B"},3,1);
        assertFalse(sq.equals(anotherSq));

    }

    @Test
    //non-empty sequence != empty sequence
```

```
    public void test_Equals_NonEmptyVsEmpty(){
        Sequence anotherSq = new Sequence();
        Sequence sq = createSequence(new String[]{"A","B"},3,1);
        assertFalse(sq.equals(anotherSq));

    }

    @Test
    //Test equals of two non-empty sequences
    //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.
    public void test_Equals_TwoDifferentCapacity(){
        Sequence sq = createSequence(new String[]{"A"},2,0);
        Sequence anotherSq = createSequence(new String[]{"A"},3,0);
        assertTrue(sq.equals(anotherSq));

    }

    @Test
    //not equal when two equals  have different size even though they have
same capacity and same current element
    public void test_Equals_DifferentSize(){
        Sequence sq = createSequence(new String[]{"A","B","C"},3,1);
        Sequence anotherSq = createSequence(new String[]{"A","B"},3,1);
        assertFalse(sq.equals(anotherSq));

    }

    @Test
    //Two sequences with different elements should not be equal even if they
have same capacity, size, current element

    public void test_Equals_DifferentElements(){
        Sequence sq = createSequence(new String[]{"A","B","D"},3,1);
        Sequence anotherSq = createSequence(new String[]{"A","B","G"},3,1);
        assertFalse(sq.equals(anotherSq));

    }

    @Test
    //Should not be equal even though they are identical in capacity, size,
elements, order

    public void test_EqualsIdentical_ButOneHasCurrentOneNot(){
        Sequence sq = createSequence(new String[]{"A","B","D"},3,1);
        Sequence anotherSq = createSequence(new String[]{"A","B","D"},3,3);
        assertFalse(sq.equals(anotherSq));
```

```
    }

    @Test
    //Should not be equal even though they are identical in element, capacity
, size, current element
    public void test_Equals_DifferentOrder(){
        Sequence sq = createSequence(new String[]{"A","B","D"},3,1);
        Sequence anotherSq = createSequence(new String[]{"D","B","A"},3,1);
        assertFalse(sq.equals(anotherSq));



    }

    @Test
    //Should not be equal even though they are identical in element, capacity
, size, order
     public void test_Equals_DifferentCurrent(){
        Sequence sq = createSequence(new String[]{"A","B","D"},3,1);
        Sequence anotherSq = createSequence(new String[]{"A","B","D"},3,0);
        assertFalse(sq.equals(anotherSq));

    }

    @Test
    //equal to itself
    public void test_Equals_Reflexivity(){
        Sequence sq = createSequence(new String[]{"A","B","C"},3,1);
        assertTrue(sq.equals(sq));

    }

    @Test
    //clone equal to original
    public void test_Equals_Clone(){
        Sequence sq = createSequence(new String[]{"A","B","D"},3,1);
        Sequence anotherSq = sq.clone();
        assertTrue(sq.equals(anotherSq));


    }

    @Test
    //if A = B then B = A

    public void test_Equals_Symmetry(){
        Sequence sq = createSequence(new String[]{"A","B","C"},3,1);
        Sequence anotherSq = createSequence(new String[]{"A","B","C"},7,1);
        assertTrue(sq.equals(anotherSq));
        assertTrue(anotherSq.equals(sq));


    }
```

```
    @Test
    //Test isEmpty of an empty sequence. Return true

    public void test_IsEmpty_EmptySequence(){
        Sequence sq = new Sequence();
        assertTrue(sq.isEmpty());

    }

    @Test
    //Test isEmpty of a non empty sequence with current. Return false
    public void test_IsEmpty_NonEmptySequence_Current(){
        Sequence sq = createSequence(new String[]{"A","B"},3,0);
        assertFalse(sq.isEmpty());
    }

    @Test
    //Test isEmpty of a non empty sequence without current. Return false
    public void test_IsEmpty_NonEmptySequence_NoCurrent(){
        Sequence sq = createSequence(new String[]{"A","B"},3,2);
        assertFalse(sq.isEmpty());
    }

    @Test
    //Test clear an empty sequence. Capacity not change. Still the same
contents
    public void test_Clear_EmptySequence(){
        Sequence sq = new Sequence();
        sq.clear();
        assertEquals(10,sq.getCapacity());
        assertFalse(sq.isCurrent());
        assertEquals(0,sq.size());
        String toStringExpected = "{} (capacity = 10)";
        assertEquals(toStringExpected,sq.toString());


    }

    @Test
    //Test clear a non empty sequence. After clearing the sequence should be
empty and there is no current element
    //Capacity don't change. Size after clearing = 0
    public void test_Clear_NonEmpty(){
        Sequence sq = createSequence(new String[]{"A","B"},3,0);
        sq.clear();
        assertEquals(0,sq.size());
        assertEquals(3,sq.getCapacity());
        assertFalse(sq.isCurrent());
        String toStringExpected = "{} (capacity = 3)";
        assertEquals(toStringExpected,sq.toString());
```

```
        }



}
```