```java
package proj3;  // Gradescope needs this.
import proj3.LinkedList;
/**
 * This is the sequence class it stores strings
 * @author Matthew Caulfield
 * @version 10/13/17
 *
 * 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
{
	// contents is a linked list that holds the strings in the sequence.
	//some of the nodes in the LL may be empty and empty nodes have null in the
data section
	//The linked list is filled starting with the first node and there can be no
gaps of nodes
	//nodes storing objects between nodes that are actively storing objects
	private LinkedList contents;
	//numItems is the number of objects actively being stored in the sequence
	private int numItems;
	//current index is the index of the current object in the sequence
	private int currentIndex;
	private final int DEFAULT_CAPACITY = 10;


    /**
     * Creates a new sequence with initial capacity 10.
     */
    public Sequence() {
       contents = new LinkedList();
       for(int i = 0; i<DEFAULT_CAPACITY; i++) {
              contents.insertAtHead(null);
       }
    }


    /**
     * Creates a new sequence.
     *
     * @param initialCapacity the initial capacity of the sequence.
     */
    public Sequence(int initialCapacity){
       contents = new LinkedList();
       for(int i = 0; i<initialCapacity; i++) {
              contents.insertAtHead(null);
       }
    }


    /**
     * 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)
{
   Sequence copy = this.clone();
   if (this.numItems + 1 > contents.getLength()) {
          this.doubleCapacity();
   }
   if(this.currentIndex == this.numItems) {
          this.currentIndex = 0;
   }
   for(int i = 0; i < currentIndex; i++){
          this.contents.changeAtIndex(copy.contents.returnByIndex(i), i);
   }
   this.contents.changeAtIndex(value, this.currentIndex);
   this.numItems++;
   for(int i = this.currentIndex + 1; i < this.numItems; i++) {
          this.contents.changeAtIndex(copy.contents.returnByIndex(i-1), i);
   }

}


/**
 * Adds a string to the sequence in the location after the current
 * element. If the sequence has no current element, the string is
 * added to the end of the sequence.
 *
 * The added element becomes the current element.
 *
 * If the sequences's capacity has been reached, the sequence will
 * expand to twice its current capacity plus 1.
 *
 * @param value the string to add.
 */
public void addAfter(String value)
{
   Sequence copy = this.clone();
   if (this.numItems + 1 > contents.getLength()) {
          this.doubleCapacity();
   }
   if(this.currentIndex == this.numItems) {
          this.currentIndex = this.numItems - 1;
   }
   for(int i = 0; i <= currentIndex; i++){
          this.contents.changeAtIndex(copy.contents.returnByIndex(i), i);
   }
   this.currentIndex++;
   this.contents.changeAtIndex(value, this.currentIndex);
```

```
   this.numItems++;
   for(int i = this.currentIndex + 1; i < this.numItems; i++) {
          this.contents.changeAtIndex(copy.contents.returnByIndex(i-1), i);
   }
}


/**
 * doubles and adds one to the capacity of a sequence
 *
 */
private void doubleCapacity() {
   int capacity = 2*this.contents.getLength()+1;
          contents  = new LinkedList();
          for(int i = 0; i < capacity; i++) {
                contents.insertAtHead(null);;
   }
}

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


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


/**
 * @return the element at the current location in the sequence, or
 * null if there is no current element.
 */
public String getCurrent()
{
   if(currentIndex != numItems) {
          return(this.contents.returnByIndex(currentIndex));
   }
   else {
          return null;
   }


}


/**
```

```
     * Increase the sequence's capacity to be
     * at least minCapacity.  Does nothing
     * if current capacity is already >= minCapacity.
     *
     * @param minCapacity the minimum capacity that the sequence
     * should now have.
     */
    public void ensureCapacity(int minCapacity)
    {
      int capacity = this.getCapacity();
      if(capacity < minCapacity) {
            int spaceNeeded = minCapacity - capacity;
            capacity = minCapacity;
            for(int i = 0; i<spaceNeeded; i++) {
                  contents.insertAtTail(null);
            }

      }
    }


    /**
     * 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 totalItems = this.numItems + another.numItems;
      this.ensureCapacity(totalItems);
      Sequence anotherCopy =  another.clone();
      for(int i = this.numItems; i < totalItems; i++) {
            this.contents.changeAtIndex(anotherCopy.contents.returnByIndex(i-
this.numItems), i);
      }
      if (this.currentIndex == this.numItems) {
            this.currentIndex = totalItems;
      }
      this.numItems = totalItems;

    }


    /**
```

```java
 * 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++;
   }
}


/**
 * 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 duplicate = new Sequence(this.getCapacity());
   duplicate.numItems = this.numItems;
   duplicate.currentIndex = this.currentIndex;
   duplicate.contents = contents.clone();
   return(duplicate);
}


/**
 * 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()
{
   Sequence copy = this.clone();
   for(int i = 0; i<this.currentIndex; i++) {
         this.contents.changeAtIndex(copy.contents.returnByIndex(i), i);
   }
   if(this.currentIndex < this.numItems) {
         this.numItems = this.numItems-1;
   }
   for(int i = this.currentIndex; i<this.numItems;  i++) {
         this.contents.changeAtIndex(copy.contents.returnByIndex(i+1), i);
```

```
   }

}


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


/**
 * 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()
{
   Sequence copy = this.clone();
   this.contents = new LinkedList();
   for(int i =0; i < this.numItems; i++){
          this.contents.insertAtTail(copy.contents.returnByIndex(i));
   }
}


/**
 * 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 output = "{";
        for(int i = 0; i<numItems; i++) {
                if(i>0) {
                        output += ", ";
                }
                if(i == currentIndex) {
                        output += ">";
                }
                output += this.contents.returnByIndex(i);
        }
        output += "} (capacity = " + this.contents.getLength() +")";
        return output;
    }

    /**
     * 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.numItems != other.numItems) {
                return false;
        }
        else if(this.currentIndex != other.currentIndex) {
                return false;
        }
        else {
                for(int i = 0; i < this.numItems; i++) {
                        if
(this.contents.returnByIndex(i)!=other.contents.returnByIndex(i)){
                                return false;
                        }
                }
                return true;
        }

    }


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

}

import junit.framework.TestCase;
import proj3.Sequence;
/**
 * JUnit tests for sequence
 * @author Matt
 * @version 10/13/17
 * 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 JUnitSequenceTest extends TestCase {

        /**
         * makes a sequence with strings from a given array of strings
         * @param sList is an array of strings to be stored in the sequence
         * @param index is desired pointer position
         * @return the sequence
         */
        private Sequence makeSequence(String[] sList, int index) {
                Sequence s = new Sequence();
                for(int i = 0; i<sList.length; i++) {
                        s.addAfter(sList[i]);
                }
                s.start();
                for(int i=0; i<index; i++) {
                        s.advance();
                }
                return s;
        }

        /**
         * makes a sequence with strings from a given array of strings
         * with a non default capacity
         * @param sList is an array of strings to be stored in the sequence
         * @param index is desired pointer position
         * @param cap is the desired capacity of the sequence
         * @return the sequence
         */
        private Sequence makeSequenceWithCapacity(String[] sList, int index, int cap)
{
```

```
		Sequence s = new Sequence(cap);
		for(int i = 0; i<sList.length; i++) {
			s.addAfter(sList[i]);
		}
		s.start();
		for(int i=0; i<index; i++) {
			s.advance();
		}
		return s;
	}


	//@test tests construction of the sequence and the to string of sequence
	public void testConstruction() {
		String [] list = {"A", "B", "C"};
		Sequence seq = makeSequence( list , 0);
		assertEquals("{>A, B, C} (capacity = 10)", seq.toString());
		Sequence seq2 = makeSequenceWithCapacity( list, 0, 5);
		assertEquals("{>A, B, C} (capacity = 5)", seq2.toString());
		String [] list2 = {};
		Sequence seq3 = makeSequence( list2 , 0);
		assertEquals("{} (capacity = 10)", seq3.toString());
		Sequence seq4 = makeSequenceWithCapacity( list2, 0, 5);
		assertEquals("{} (capacity = 5)", seq4.toString());
		String [] list3 = {"A", "B", "C", "D", "E", "F", "G", "H", "I", "J"};
		Sequence seq5 = makeSequence( list3 , 0);
		assertEquals("{>A, B, C, D, E, F, G, H, I, J} (capacity = 10)",
seq5.toString());
	}

	//@test tests addBefore method
	public void testAddBefore() {
		String [] list = {};
		Sequence seq = makeSequenceWithCapacity( list , 0, 4);
		seq.addBefore("D");
		assertEquals("{>D} (capacity = 4)", seq.toString());
		seq.addBefore("C");
		assertEquals("{>C, D} (capacity = 4)", seq.toString());
		seq.addBefore("B");
		assertEquals("{>B, C, D} (capacity = 4)", seq.toString());
		seq.addBefore("A");
		assertEquals("{>A, B, C, D} (capacity = 4)", seq.toString());
		seq.addBefore("Z");
		assertEquals("{>Z, A, B, C, D} (capacity = 9)", seq.toString());
	}

	//@test tests addAfter method
	public void testAddAfter() {
		String [] list = {};
		Sequence seq = makeSequenceWithCapacity( list , 0, 4);
		seq.addAfter("A");
		assertEquals("{>A} (capacity = 4)", seq.toString());
		seq.addAfter("B");
		assertEquals("{A, >B} (capacity = 4)", seq.toString());
		seq.addAfter("C");
```

```
	assertEquals("{A, B, >C} (capacity = 4)", seq.toString());
	seq.addAfter("D");
	assertEquals("{A, B, C, >D} (capacity = 4)", seq.toString());
	seq.addAfter("E");
	assertEquals("{A, B, C, D, >E} (capacity = 9)", seq.toString());
}

//@test tests isCurrent
public void testIsCurrent() {
	String [] list = {"A", "B", "C"};
	Sequence seq = makeSequence( list , 0);
	assertEquals(true, seq.isCurrent());
	for(int i = 0; i<3; i++) {
		seq.advance();
	}
	assertEquals(false, seq.isCurrent());
}

//@test tests getCapacity
public void testGetCapacity() {
	String [] list = {"A", "B", "C"};
	Sequence seq = makeSequence( list , 0);
	assertEquals(10, seq.getCapacity());
}

//@test tests getCurrent
public void testGetCurrent() {
	String [] list = {"A", "B", "C"};
	Sequence seq = makeSequence( list , 0);
	assertEquals("A", seq.getCurrent());
	seq.advance();
	assertEquals("B", seq.getCurrent());
	seq.advance();
	assertEquals("C", seq.getCurrent());
	seq.advance();
	assertEquals(null, seq.getCurrent());
}

//@test tests ensureCapacity
public void testEnsuresCapacity() {
	String [] list = {"A", "B", "C"};
	Sequence seq = makeSequence( list , 0);
	seq.ensureCapacity(11);
	assertEquals(11, seq.getCapacity());
	seq.ensureCapacity(9);
	assertEquals(11, seq.getCapacity());
	assertEquals("{>A, B, C} (capacity = 11)", seq.toString());
}

//@test tests addAll
public void testAddAll() {
	String [] listE = {};
	Sequence seqE1 = makeSequence( listE, 0);
	Sequence seqE2 = makeSequence( listE, 0);
	seqE1.addAll(seqE2);
```

```
        assertEquals("{} (capacity = 10)", seqE1.toString());
        assertEquals(0, seqE1.size());
        String [] list = {"A", "B", "C"};
        Sequence seq = makeSequence( list , 0);
        seq.addAll(seqE1);
        assertEquals("{>A, B, C} (capacity = 10)", seq.toString());
        assertEquals(3, seq.size());
        seqE1.addAll(seq);
        assertEquals("{>A, B, C} (capacity = 10)", seq.toString());
        String [] list2 = {"A", "B"};
        Sequence seq2 = makeSequence( list2 , 0);
        seq.addAll(seq2);
        assertEquals("{>A, B, C, A, B} (capacity = 10)", seq.toString());
        assertEquals(5, seq.size());
        Sequence seq3 = makeSequenceWithCapacity(list, 0, 3);
        seq3.addAll(seq2);
        assertEquals("{>A, B, C, A, B} (capacity = 5)", seq3.toString());
}

//@test test clone
public void testClone() {
        String [] list = {"A", "B", "C"};
        Sequence seq = makeSequence( list , 0);
        Sequence seqClone = seq.clone();
        assertEquals(seq.toString(), seqClone.toString());
        seqClone.removeCurrent();
        assertEquals("{>A, B, C} (capacity = 10)", seq.toString());
        seq.removeCurrent();
        seq.removeCurrent();
        assertEquals("{>B, C} (capacity = 10)", seqClone.toString());
}

//@test test removeCurrent
public void testRemoveCurrent() {
        String [] list = {};
        Sequence seq = makeSequence(list , 0);
        seq.removeCurrent();
        assertEquals("{} (capacity = 10)", seq.toString());
        String [] list2 = {"A", "B", "C"};
        Sequence seq2 = makeSequence(list2, 3);
        seq2.removeCurrent();
        assertEquals("{A, B, C} (capacity = 10)", seq2.toString());
        seq2.addAfter("D");
        seq2.start();
        seq2.removeCurrent();
        assertEquals("{>B, C, D} (capacity = 10)", seq2.toString());
        seq2.advance();
        seq2.removeCurrent();
        assertEquals("{B, >D} (capacity = 10)", seq2.toString());
        seq2.removeCurrent();
        assertEquals("{B} (capacity = 10)", seq2.toString());
}

//@test test trimToSize
public void testTrimToSize() {
```

```
	String [] list = {"A", "B", "C"};
	Sequence seq = makeSequence(list , 0);
	seq.trimToSize();
	assertEquals("{>A, B, C} (capacity = 3)", seq.toString());
	seq.trimToSize();
	assertEquals("{>A, B, C} (capacity = 3)", seq.toString());
	String [] listE = {};
	Sequence seqE = makeSequence(listE, 0);
	seqE.trimToSize();
	assertEquals("{} (capacity = 0)", seqE.toString());
}

//@test test equals
public void testEquals() {
	String [] list = {"A", "B", "C"};
	Sequence seq = makeSequence(list , 0);
	Sequence seq2 = makeSequenceWithCapacity(list , 0, 4);
	assertEquals(true, seq.getCapacity()!=seq2.getCapacity());
	assertEquals(true, seq.equals(seq2));
	assertEquals(true, seq2.equals(seq));
	String [] listE = {};
	Sequence seqE1 = makeSequence( listE, 0);
	Sequence seqE2 = makeSequence( listE, 0);
	assertEquals(true, seqE1.equals(seqE2));
	assertEquals(false, seq.equals(seqE2));
	assertEquals(true, seq.equals(seq));
	seq2.advance();
	assertEquals(false, seq.equals(seq2));
	seq.advance();
	seq.advance();
	seq.advance();
	seq2.advance();
	seq2.advance();
	assertEquals(true, seq.equals(seq2));
}

//@test test isEmpty
public void testIsEmpty() {
	String [] list = {"A", "B", "C"};
	Sequence seq = makeSequence(list , 0);
	assertEquals(false, seq.isEmpty());
	String [] listE = {};
	Sequence seqE = makeSequence(listE, 0);
	assertEquals(true, seqE.isEmpty());
}

//@test clear
public void testClear() {
	String [] list = {"A", "B", "C"};
	Sequence seq = makeSequence(list , 0);
	seq.clear();
	assertEquals(true, seq.isEmpty());
	assertEquals(false, seq.isCurrent());
}
```

```
}
```

```
package proj3;

import proj3.ListNode;

/**
 *This is the Linked List class it acts like a list
 *but its data is stored in different nodes
 *@author Matthew Caulfield
 *@version 10/13/17
 *
 * 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 LinkedList
{
    private int length;          // number of nodes
    private ListNode firstNode;  // pointer to first node

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

    /** 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++;
    }

    /** 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(this.isEmpty()) {
          return null;
   }
   else {
          String firstNodeString = firstNode.data;
          firstNode = firstNode.next;
          length --;
          return firstNodeString;
   }

}

/** insert data at end of list
 *
 * @param newData new String to be inserted
 */
public void insertAtTail(String newData)
{
   ListNode newNode = new ListNode(newData);
   ListNode currentNode = firstNode;
   if(this.isEmpty() == true) {
          firstNode = newNode;
   }
   else {
          for(int i = 0; i < this.getLength()-1; i++) {
                currentNode = currentNode.next;
          }
          currentNode.next = newNode;
   }
   length++;
}



/**
 * Inserts a node into the LL at the index given the following nodes
 * get pushed back an index number. If the number of nodes is less than the index
 * than the data is added at the tail
 * @param index number of where the new string should be added
 */
public void insertAtIndex(String newData, int index) {
   ListNode newNode = new ListNode(newData);
   ListNode currentNode = firstNode;
   if(index == 0) {
          this.insertAtHead(newData);
   }
   else if(this.getLength() > index) {
          for(int i = 0; i < index - 1; i++) {
                currentNode = currentNode.next;
          }
          newNode.next = currentNode.next;
          currentNode.next = newNode;
          length++;
```

```
		}
		else {
			this.insertAtTail(newData);
		}
	}

	/**
	 * returns the data at a given index
	 * if the index does not exist it returns null
	 * @param the index that holds the returned data
	 * @return data at the given index
	 */
	public String returnByIndex(int index) {
		if(index < 0) {
			return null;
		}
		else if(index < this.getLength()) {
			ListNode currentNode = firstNode;
				for(int i = 0; i < index; i++) {
					currentNode = currentNode.next;
				}
				return currentNode.data;
		}
		else {
			return null;
		}
	}

	/**
	 * changes the data of the node at the given index
	 * @param the new data for the node
	 * @param the index at which the data is changed
	 */
	public void changeAtIndex(String newData, int index) {
		if(index >= 0 && index < this.getLength()) {
			ListNode currentNode = firstNode;
			for(int i = 0; i < index; i++) {
				currentNode = currentNode.next;
			}
			currentNode.data = newData;
		}
	}

	/**
	 * 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)
	{
		boolean found = false;
		int i = 0;
		ListNode currentNode = firstNode;
```

```
        while(found == false && i < this.getLength()) {
                if(currentNode.data.equals(value)) {
                        found = true;
                        return i;
                }
                i++;
                currentNode = currentNode.next;
        }
        return -1;
    }


    /**
     *  @return return linked list as printable string
     */
    public String toString()
    {
        String toReturn="(";
        ListNode runner=firstNode;
        while (runner!=null)
        {
                toReturn = toReturn + runner;  //call node's toString automatically
                runner=runner.next;
                if (runner!=null)
                {
                        toReturn = toReturn + ",";
                }
        }
        toReturn = toReturn + ")";
        return toReturn;
    }

    /**
     * @return returns a copy of the Linked List
     */
    public LinkedList clone(){
        LinkedList aClone = new LinkedList();
        for(int i = 0; i < this.getLength(); i++) {
                aClone.insertAtTail(this.returnByIndex(i));
        }
        return aClone;
    }

    /**
     *
     * @return length of LL
     */
    public int getLength() {return length;}

    /**
     *
     * @return true if LL empty or false if not
     */
    public boolean isEmpty() {return getLength()==0;}
}
```

```java
import junit.framework.TestCase;
import proj3.LinkedList;

/**
 *
 * @author Matt
 *JUnit test for Linked Lists
 *@version 10/13/2017
 */
public class LinkedListJUnitTests extends TestCase {
	/**
	 * tests removeHead linked list method
	 */
	public void testRemoveHead() {
		LinkedList aLinkedList = new LinkedList();
		aLinkedList.insertAtHead("D");
		aLinkedList.insertAtHead("C");
		aLinkedList.insertAtHead("B");
		aLinkedList.insertAtHead("A");
		assertEquals("(A,B,C,D)", aLinkedList.toString());
		assertEquals(aLinkedList.getLength(), 4);
		aLinkedList.removeHead();
		assertEquals(aLinkedList.getLength(), 3);
		assertEquals("(B,C,D)", aLinkedList.toString());
		aLinkedList.removeHead();
		assertEquals(aLinkedList.getLength(), 2);
		assertEquals("(C,D)", aLinkedList.toString());
		aLinkedList.removeHead();
		assertEquals(aLinkedList.getLength(), 1);
		assertEquals("(D)", aLinkedList.toString());
		aLinkedList.removeHead();
		assertEquals(aLinkedList.getLength(), 0);
		assertEquals("()", aLinkedList.toString());
		aLinkedList.removeHead();
		assertEquals(aLinkedList.getLength(), 0);
		assertEquals("()", aLinkedList.toString());
	}

	/**
	 * test insert at tail
	 */
	public void testInsertAtTail() {
		LinkedList aLinkedList = new LinkedList();
		aLinkedList.insertAtTail("A");
		assertEquals(aLinkedList.getLength(), 1);
		assertEquals("(A)", aLinkedList.toString());
		aLinkedList.insertAtTail("B");
		assertEquals("(A,B)", aLinkedList.toString());
		assertEquals(aLinkedList.getLength(), 2);
		aLinkedList.insertAtTail("C");
		assertEquals("(A,B,C)", aLinkedList.toString());
```

```
        assertEquals(aLinkedList.getLength(), 3);
        aLinkedList.insertAtTail("D");
        assertEquals("(A,B,C,D)", aLinkedList.toString());
        assertEquals(aLinkedList.getLength(), 4);
}

/**
 * test of indexOf
 */
public void testIndexOf() {
        LinkedList aLinkedList = new LinkedList();
        assertEquals(-1, aLinkedList.indexOf("A"));
        assertEquals("()", aLinkedList.toString());
        assertEquals(aLinkedList.getLength(), 0);
        aLinkedList.insertAtTail("A");
        assertEquals(aLinkedList.indexOf("A"), 0);
        assertEquals(-1, aLinkedList.indexOf("B"));
        assertEquals(aLinkedList.getLength(), 1);
        assertEquals("(A)", aLinkedList.toString());
        aLinkedList.insertAtTail("B");
        aLinkedList.insertAtTail("C");
        assertEquals(aLinkedList.indexOf("A"), 0);
        assertEquals(1, aLinkedList.indexOf("B"));
        assertEquals(2, aLinkedList.indexOf("C"));
        assertEquals(-1, aLinkedList.indexOf("D"));
        assertEquals(aLinkedList.getLength(), 3);
        assertEquals("(A,B,C)", aLinkedList.toString());

}

/**
 * tests the insertAtIndex function for an empty Linked List
 */
public void test_insertAtIndexEmpty() {
        LinkedList aLinkedList = new LinkedList();
        aLinkedList.insertAtIndex("A", 0);
        assertEquals("(A)", aLinkedList.toString());
        aLinkedList.removeHead();
        assertEquals("()", aLinkedList.toString());
        aLinkedList.insertAtIndex("A", 1);
        assertEquals("(A)", aLinkedList.toString());
}

/**
 * tests the insertAtIndex function for a LL with strings in it
 */
public void test_insertAtIndexFull() {
        LinkedList aLinkedList = new LinkedList();
        aLinkedList.insertAtTail("A");
        aLinkedList.insertAtTail("B");
        aLinkedList.insertAtTail("C");
        assertEquals("(A,B,C)", aLinkedList.toString());
        aLinkedList.insertAtIndex("E", 1);
        assertEquals("(A,E,B,C)", aLinkedList.toString());
        assertEquals(4, aLinkedList.getLength());
```

```
        aLinkedList.insertAtIndex("F", 0);
        assertEquals("(F,A,E,B,C)", aLinkedList.toString());
        assertEquals(5, aLinkedList.getLength());
        aLinkedList.insertAtIndex("G", 4);
        assertEquals("(F,A,E,B,G,C)", aLinkedList.toString());
        assertEquals(6, aLinkedList.getLength());
        aLinkedList.insertAtIndex("H", 6);
        assertEquals("(F,A,E,B,G,C,H)", aLinkedList.toString());
        assertEquals(7, aLinkedList.getLength());
}

/**
 * test the return by index function for an empty LL
 */
public void test_returnByIndexEmpty(){
        LinkedList aLinkedList = new LinkedList();
        assertEquals(null,aLinkedList.returnByIndex(0));
        assertEquals(null,aLinkedList.returnByIndex(1));
        assertEquals(null, aLinkedList.returnByIndex(-1));
}

/**
 * test the return by index function for a full LL
 */
public void test_returnByIndexFull() {
        LinkedList aLinkedList = new LinkedList();
        aLinkedList.insertAtTail("A");
        aLinkedList.insertAtTail("B");
        aLinkedList.insertAtTail("C");
        assertEquals("(A,B,C)", aLinkedList.toString());
        assertEquals("A", aLinkedList.returnByIndex(0));
        assertEquals("B", aLinkedList.returnByIndex(1));
        assertEquals("C", aLinkedList.returnByIndex(2));
        assertEquals(null, aLinkedList.returnByIndex(3));
        assertEquals(null, aLinkedList.returnByIndex(-1));
        assertEquals("(A,B,C)", aLinkedList.toString());
        assertEquals(3, aLinkedList.getLength());
}

/***
 * test the changeAtIndex function for an empty LL
 */
public void test_changeAtIndexEmpty() {
        LinkedList aLinkedList = new LinkedList();
        aLinkedList.changeAtIndex("A", 0);
        assertEquals("()", aLinkedList.toString());
        aLinkedList.changeAtIndex("A", 1);
        assertEquals("()", aLinkedList.toString());
        aLinkedList.changeAtIndex("A", -1);
        assertEquals("()", aLinkedList.toString());
}

/**
 * test the changeAtIndex function for a full LL
 */
```

```
public void test_changeAtIndexFull() {
	LinkedList aLinkedList = new LinkedList();
	aLinkedList.insertAtTail("A");
	aLinkedList.insertAtTail("B");
	aLinkedList.insertAtTail("C");
	assertEquals("(A,B,C)", aLinkedList.toString());
	aLinkedList.changeAtIndex("D", -1);
	assertEquals("(A,B,C)", aLinkedList.toString());
	aLinkedList.changeAtIndex("D", 0);
	assertEquals("(D,B,C)", aLinkedList.toString());
	aLinkedList.changeAtIndex("D", 1);
	assertEquals("(D,D,C)", aLinkedList.toString());
	aLinkedList.changeAtIndex("D", 2);
	assertEquals("(D,D,D)", aLinkedList.toString());
	aLinkedList.changeAtIndex("D", 3);
	assertEquals("(D,D,D)", aLinkedList.toString());
}

/**
 * test the clone function for an empty LL
 */
public void test_cloneEmpty() {
	LinkedList aLinkedList = new LinkedList();
	assertEquals("()", aLinkedList.clone().toString());
}

/**
 * test the clone function for a full LL
 */
public void test_cloneFull() {
	LinkedList aLinkedList = new LinkedList();
	aLinkedList.insertAtTail("A");
	aLinkedList.insertAtTail("B");
	aLinkedList.insertAtTail("C");
	assertEquals("(A,B,C)", aLinkedList.toString());
	LinkedList aClone = aLinkedList.clone();
	assertEquals("(A,B,C)", aClone.toString());
	assertEquals(3, aClone.getLength());
}
```