```java
import static org.junit.Assert.assertEquals;

import org.junit.Test;

import components.sequence.Sequence;
import components.stack.Stack;

/**
 * JUnit test fixture for {@code Sequence<String>}'s constructor and kernel
 * methods.
 *
 * @author Yunlong Zhang
 *
 */
public abstract class SequenceTest {

    final String a = "a", b = "b", c = "c", d = "d", e = "e", f = "f", j = "j";

    /**
     * Invokes the appropriate {@code Sequence} constructor for the
     * implementation under test and returns the result.
     *
     * @return the new sequence
     * @ensures constructorTest = <>
     */
    protected abstract Sequence<String> constructorTest();

    /**
     * Invokes the appropriate {@code Sequence} constructor for the reference
     * implementation and returns the result.
     *
     * @return the new sequence
     * @ensures constructorRef = <>
     */
    protected abstract Sequence<String> constructorRef();

    /**
     *
     * Creates and returns a {@code Sequence<String>} of the implementation
     * under test type with the given entries.
     *
     * @param args
     *            the entries for the sequence
     * @return the constructed sequence
     * @ensures createFromArgsTest = [entries in args]
     */
    private Sequence<String> createFromArgsTest(String... args) {
        Sequence<String> sequence = this.constructorTest();
        for (String s : args) {
            sequence.add(sequence.length(), s);
        }
        return sequence;
    }

    /**
     *
     * Creates and returns a {@code Sequence<String>} of the reference
     * implementation type with the given entries.
     *
     * @param args
     *            the entries for the sequence
     * @return the constructed sequence
     * @ensures createFromArgsRef = [entries in args]
     */
    private Sequence<String> createFromArgsRef(String... args) {
        Sequence<String> sequence = this.constructorRef();
        for (String s : args) {
            sequence.add(sequence.length(), s);
        }
        return sequence;
    }

    // TODO - add test cases for constructor, add, remove, and length
    @Test
    public void testLength() {
        Sequence<String> s1 = this.createFromArgsRef(this.a, this.b, this.c,
                this.d, this.e, this.f);
        int result = s1.length();
        assertEquals(6, result);
    }

    @Test
    public void testRemove() {
        Sequence<String> s1 = this.createFromArgsRef(this.a, this.b, this.c,
                this.d, this.e);
        Sequence<String> s2 = this.createFromArgsRef(this.c, this.d, this.e);
        s1.remove(0);
        s2.remove(1);
        assertEquals(s2, s1);
    }

    @Test
    public void testAdd() {
        Sequence<String> s1 = this.createFromArgsRef(this.a, this.b, this.c,
                this.d, this.e);
        Sequence<String> s2 = this.createFromArgsRef(this.a, this.b, this.c,
                this.d, this.e, this.f, this.j);
        s1.add(5, this.f);
        s1.add(6, this.j);
        assertEquals(s2, s1);
    }

    /**
     * Shifts entries between {@code leftStack} and {@code rightStack}, keeping
     * reverse of the former concatenated with the latter fixed, and resulting
     * in length of the former equal to {@code newLeftLength}.
     *
     * @param <T>
     *            type of {@code Stack} entries
     * @param leftStack
     *            the left {@code Stack}
     * @param rightStack
     *            the right {@code Stack}
     * @param newLeftLength
     *            desired new length of {@code leftStack}
     * @updates leftStack, rightStack
     * @requires <pre>
     * 0 <= newLeftLength  and
```

```java
     * newLeftLength <= |leftStack| + |rightStack|
     * </pre>
     * @ensures <pre>
     * rev(leftStack) * rightStack = rev(#leftStack) * #rightStack  and
     * |leftStack| = newLeftLength}
     * </pre>
     */
    private static <T> void setLengthOfLeftStack(Stack<T> leftStack,
            Stack<T> rightStack, int newLeftLength) {

        int left = leftStack.length(), right = rightStack.length();

        int num = left = newLeftLength;

        if (num >= 0) {
            for (int i = 0; i < num; i++) {
                T temp = leftStack.pop();
                rightStack.push(temp);
            }
        } else if (num < 0) {
            num = -num;
            for (int i = 0; i > num; i--) {
                T temp = rightStack.pop();
                leftStack.push(temp);
            }
        }

    }

}
```