Info 3

Laboratory 8

02.07.2020

Lab 8: Test-driven development

Niklas Lengert s0563290
Pavel Tsvyatkov s0559632
Robin Jaspers s0568739
Nataliia Azarnykh s0568691

Your task is to develop a small package that implements a simple linked list. You need to develop methods for inserting into the list, deleting an element from the list, and reversing the list. You will be doing this in a test-driven manner! You can do this either in Java or in any other language. You find stubs in Java and Ruby on Prof. Kleinen's **github** if you are unsure what is meant by this.

Create a Class Node as head of a Linked List and develop a method for inserting Nodes into it in a test-driven manner, that is, develop the Unit Tests first.

Choose one of the two methods for dealing with assertions and expectations—by either creating a helper (e.g. assert.java) or a toString method as in toString.java. Implement the assertions first, and then implement the code until it conforms to the assertions. First implement the insertion, then the deletion from the list.

Then do the same for reversing the list, that is, develop the method reverse() **test-driven**. This is a bit tricky but doable with just one pass through the singly-linked list. Stick to the test cases, and give it a try—one node at a time! Start with the empty list, then a list with one node, then lists with more than one node.

Note that you need to make decisions on the actual interface of the methods and the whole list while writing the test, as well as on how the anchor for the first element is stored and changed if necessary. Write a short note (possibly as comment in the Test Code) why you decided to do it as you did.

For the bored: Refactor your code after the tests are running, and experiment with different implementations for deletion and reversion of a list.

We created a simple Node class as head of a LinkedList first.

Node.java | LinkedList.java

```java
public class Node {

    Object data;
    Node next;

    public Node(Object data, Node next) {
        this.data = data;
        this.next = next;
    }

    public Node(Object data) {
        this.data = data;
    }

}
```

LinkedList.java

```java
public class LinkedList {
    Node head;
}
```

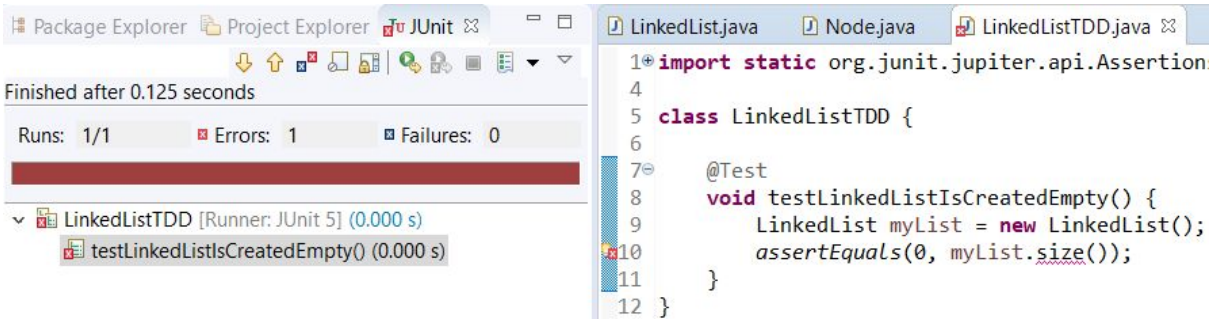
From here on we have three steps to follow, this is our mantra:

1. Red - Create a unit tests that fails
2. Green - Write production code that makes that test pass.
3. Refactor - Clean up the mess you just made.

retrieved from: https://blog.cleancoder.com/uncle-bob/2014/12/17/TheCyclesOfTDD.html

Before even creating a LinkedList class, we first wrote a test to make sure the size of the LinkedList is 0 when it is first created.

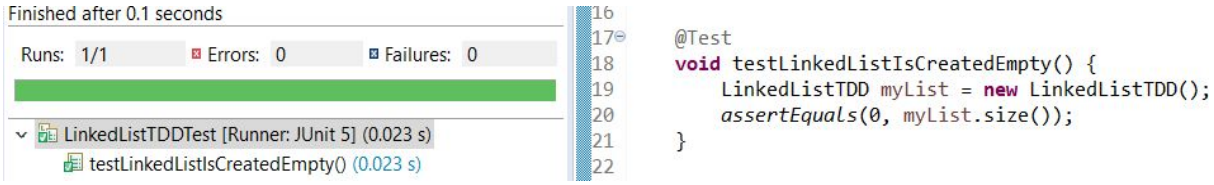
That's our red step.



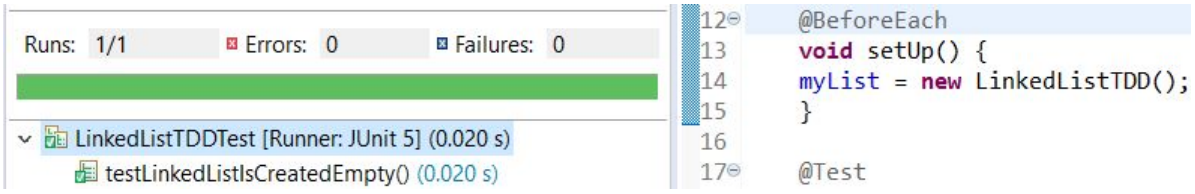
Now comes our green step.
We want to write enough code to pass the test, so we include the private size field and a public method to return it.

```
1 public class LinkedList {
2     Node head;
3     private int size = 0;
4
5     public int size() {
6         return size;
7     }
8 }
```

Yay, running the JUnit test now comes out green.



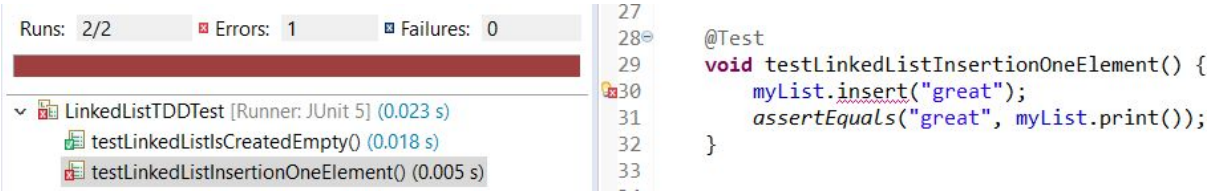
Since we will be using the myList instance multiple times, we decided to set up an @Before annotation.
It was interesting to see that we started getting a NullPointerException for the same test.
We noticed that in JUnit 5 it was changed from @Before to @BeforeEach and that fixed the NullPointerError, but still, we learned something about the changes from JUnit 4 to version 5.



We made sure to check the documentation for JUnit 5 to see if there are any other interesting things we could be using: https://junit.org/junit5/docs/current/user-guide/

We want to continue adding functionality in the same fashion, starting with the test cases first and then moving on to implement the methods in LinkedList so they conform to the test.

The next test case we wrote is the insertion one. We don't have the method implemented yet so it was still red at this point.

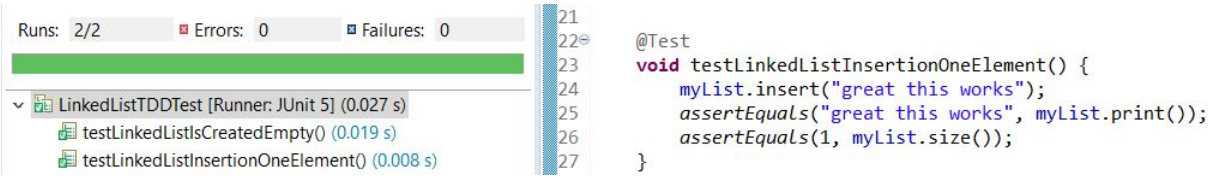


Since we already dealt with LinkedList implementations in Informatics 2, we added the method in the same way we did it before, but also included the required null checks.

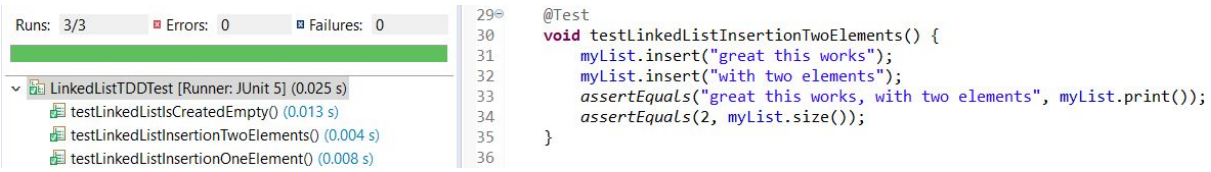
```java
public void insert(Object data) {

        if (head == null) {
            head = new Node(data, null);
        }
        else {
            Node temp = head;
            while (temp.next != null) {
                temp = temp.next;
            }
            temp.next = new Node(data);
        }
        size++;
}
```

We also asked Prof. Weber-Wulff if we can use her print method implementation for the LinkedList that she showed to us last semester and she said it's okay. (We really liked it and wanted to use that, because it deals with the commas separating the elements of the list).

After having those methods ready, we ran the test and also made sure that we still keep track of the size correctly.



We tried it with more than one element just to be sure.



And then another test for inserting something else other than Strings.

Runs: 4/4 Errors: 0 Failures: 0

LinkedListTDDTest [Runner: JUnit 5] (0.033 s)
- testLinkedListIsCreatedEmpty() (0.018 s)
- testLinkedListInsertionTwoElements() (0.003 s)
- testLinkedListInsertionOneElement() (0.002 s)
- testLinkedListInsertionTwoIntegers() (0.010 s)

```
@Test
void testLinkedListInsertionTwoIntegers() {
    myList.insert(1);
    myList.insert(2);
    assertEquals("1, 2", myList.print());
    assertEquals(2, myList.size());
}
```

We also found the idea to have a method addFirst, that adds elements in the beginning of the list instead of just always putting them at the end.

We wrote a test case that assumes the elements are always added at the front. So in this case the second pass should come as the first element in the list.

Finished after 0.125 seconds

Runs: 5/5 Errors: 1 Failures: 0

LinkedListTDDTest [Runner: JUnit 5] (0.039 s)
- testLinkedListInsertionWithAddFirst() (0.018 s)

```
@Test
void testLinkedListInsertionWithAddFirst() {
    myList.addFirst("pass one");
    myList.addFirst("pass two");
    assertEquals("pass two, pass one", myList.print());
}
```

We used Prof. Weber-Wulff's implementation of the addFirst method as well. Tests passed!

Runs: 5/5 Errors: 0 Failures: 0

LinkedListTDDTest [Runner: JUnit 5] (0.027 s)
- testLinkedListInsertionWithAddFirst() (0.017 s)

```
@Test
void testLinkedListInsertionWithAddFirst() {
    myList.addFirst("pass one");
    myList.addFirst("pass two");
    assertEquals("pass two, pass one", myList.print());
}
```

Next, we wrote a tests for deleting from the list.

Runs: 6/6 Errors: 1 Failures: 0

LinkedListTDDTest [Runner: JUnit 5] (0.038 s)
- testDeletingFromList() (0.017 s)

```
@Test
void testDeletingFromList() {
    myList.insert(1);
    myList.remove();
    assertEquals("", myList.print());
}
```

We used the same remove method from the Informatics 2 classes that removes from the front of the list.
It was interesting to see that using the insert() method and then remove() did not work correctly, but using the addFirst() method and then remove() worked properly. We went back to look at our insert() method and figure out what was wrong. We noticed we were not keeping track of the current Node and managed to fix it quickly.

Tests passed when adding and deleting one element.

Runs: 6/6 Errors: 0 Failures: 0

LinkedListTDDTest [Runner: JUnit 5] (0.029 s)
- testLinkedListInsertionWithAddFirst() (0.017 s)
- testLinkedListIsCreatedEmpty() (0.001 s)

```
@Test
void testDeletingOneElementFromList() {
    myList.insert("delete this");
    myList.remove();
    assertEquals("", myList.print());
}
```

We tried adding and deleting multiple elements to make sure everything is working fine still. (This time using the addFirst method to add elements)

Runs: 7/7 Errors: 0 Failures: 0

- LinkedListTDDTest [Runner: JUnit 5] (0.034 s)
  - testDeletingMoreElementsFromList() (0.018 s)
  - testLinkedListInsertionWithAddFirst() (0.001 s)
  - testLinkedListIsCreatedEmpty() (0.001 s)
  - testLinkedListInsertionTwoElements() (0.002 s)
  - testDeletingOneElementFromList() (0.001 s)
  - testLinkedListInsertionOneElement() (0.001 s)
  - testLinkedListInsertionTwoIntegers() (0.010 s)

```java
    @Test
    void testDeletingMoreElementsFromList() {
        myList.addFirst("first");
        myList.remove();
        myList.addFirst("second");
        myList.addFirst("third");
        myList.addFirst("fourth");
        myList.remove();
        myList.remove();
        assertEquals("second", myList.print());
        assertEquals(1, myList.size());
    }
```

The next test we wrote was to reverse the linked list. We first started with an empty list and then make our way to reverse a list with more elements.

Runs: 8/8 Errors: 1 Failures: 0

- LinkedListTDDTest [Runner: JUnit 5] (0.041 s)
  - testDeletingMoreElementsFromList() (0.020 s)
  - testReverseEmptyList() (0.009 s)

```java
    }

    @Test
    void testReverseEmptyList() {
        myList.reverse()
        assertEquals("", myList.print());
    }
```

First we started with a very simple step of the reverse method, the null check.

```java
    public void reverse() {
        if (head == null) {
            return;
        }
    }
```

No real functionality in the method yet, but the unit tests passed.

Runs: 8/8 Errors: 0 Failures: 0

- LinkedListTDDTest [Runner: JUnit 5] (0.035 s)
  - testDeletingMoreElementsFromList() (0.020 s)
  - testReverseEmptyList() (0.001 s)

```java
    @Test
    void testReverseEmptyList() {
        myList.reverse();
        assertEquals("", myList.print());
    }
```

We wrote another JUnit test to reverse a one element list, which also passed at this point.

Runs: 9/9 Errors: 0 Failures: 0

- LinkedListTDDTest [Runner: JUnit 5] (0.044 s)
  - testDeletingMoreElementsFromList() (0.024 s)
  - testReverseEmptyList() (0.001 s)
  - testLinkedListInsertionWithAddFirst() (0.002 s)
  - testLinkedListIsCreatedEmpty() (0.001 s)
  - testLinkedListInsertionTwoElements() (0.001 s)
  - testDeletingOneElementFromList() (0.001 s)
  - testLinkedListInsertionOneElement() (0.001 s)
  - testLinkedListInsertionTwoIntegers() (0.001 s)
  - testReverseListWithOneElement() (0.012 s)

```java
    @Test
    void testReverseEmptyList() {
        myList.reverse();
        assertEquals("", myList.print());
    }

    @Test
    void testReverseListWithOneElement() {
        myList.insert("first");
        myList.reverse();
        assertEquals("first", myList.print());
    }
```

We wrote the following test with just two elements.

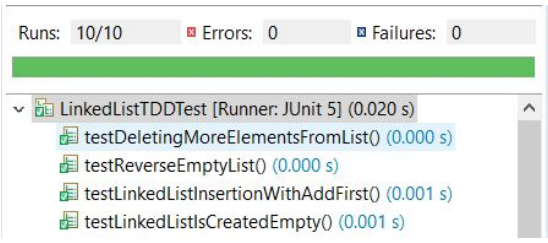
```java
@Test
void testReverseListWithMoreElements() {
    myList.insert("first");
    myList.insert("second");
    myList.reverse();
    assertEquals("second, first", myList.print());
}
```

Since some of us didn't deal with reversing a linked list yet, we thought about the implementation of the reverse method for a while. We made some sketches on paper to get the general idea and it became much more clear, but also going one step at a time with the tests helped a lot.

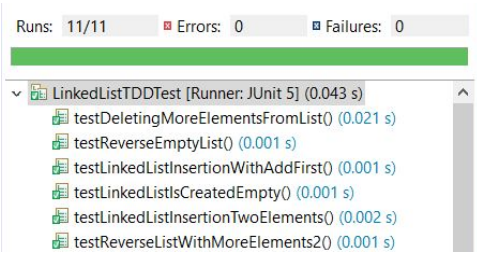
```java
public void reverse() {
    if (head == null) {
        return;
    }

    Node curr = head;
    Node temp;
    while (curr.next != null) {
        temp = curr.next;
        curr.next = temp.next ;
        temp.next = head;
        head = temp;
    }
}
```

We also found a reference about reversing a linked list here in this preview of a book.
Source: https://books.google.de/books?id=D-liDwAAQBAJ&printsec=frontcover#v=onepage&q&f=false

The test with two elements passed and we decided to test further with more elements.




```java
@Test
void testReverseListWithMoreElements() {
    myList.insert("first");
    myList.insert("second");
    //List looks like: first, second
    myList.reverse();
    assertEquals("second, first", myList.print());
}
```




```java
@Test
void testReverseListWithMoreElements2() {
    myList.insert("first");
    myList.insert("second");
    myList.insert("third");
    myList.insert("fourth");
    //List looks like: first, second, third, fourth
    myList.reverse();
    assertEquals("fourth, third, second, first", myList.print());
}
```

Then we also used the addFirst method to see if it also works correctly after we add elements and reverse them. We decided to add 5 elements, check the size of the list, remove one element, make sure the size is one less now and then reverse the list.

Runs: 12/12 Errors: 0 Failures: 0

- LinkedListTDDTest [Runner: JUnit 5] (0.041 s)
  - testDeletingMoreElementsFromList() (0.019 s)
  - testReverseManyElementsWithAddFirst() (0.001 s)
  - testReverseEmptyList() (0.001 s)
  - testLinkedListInsertionWithAddFirst() (0.002 s)
  - testLinkedListIsCreatedEmpty() (0.001 s)
  - testLinkedListInsertionTwoElements() (0.001 s)
  - testReverseListWithMoreElements2() (0.001 s)
  - testReverseListWithMoreElements() (0.001 s)
  - testDeletingOneElementFromList() (0.001 s)

Failure Trace

```java
@Test
void testReverseManyElementsWithAddFirst() {
    myList.addFirst(1);
    myList.addFirst(2);
    myList.addFirst(3);
    myList.addFirst(4);
    myList.addFirst(5);
    assertEquals(5, myList.size());
    // List looks like 5,4,3,2,1

    myList.remove();
    assertEquals(4, myList.size());
    // List looks like 4,3,2,1

    myList.reverse();

    assertEquals("1, 2, 3, 4", myList.print());
    assertEquals(4, myList.size());
}
```

Additional tests are always nice to have, so we tried to think of different test cases we would like to test.

Runs: 15/15 Errors: 0 Failures: 0

- LinkedListTDDTest [Runner: JUnit 5] (0.029 s)
  - testDeletingMoreElementsFromList() (0.005 s)
  - testReverseManyElementsWithAddFirst() (0.001 s)
  - testReverseEmptyList() (0.001 s)
  - TestRemovingWell() (0.001 s)
  - testLinkedListInsertionWithAddFirst() (0.001 s)
  - testLinkedListIsCreatedEmpty() (0.001 s)
  - testLinkedListInsertionTwoElements() (0.001 s)
  - testReverseListWithMoreElements2() (0.001 s)
  - testReverseListWithMoreElements() (0.001 s)

Failure Trace

```java
@Test
void testReverseManyElementsWithInsertAndRemove() {
    myList.insert(1);
    myList.insert(2);
    myList.insert(3);
    myList.insert(4);
    myList.insert(5);
    myList.insert(6);
    assertEquals(6, myList.size()); //List looks like 1, 2, 3, 4, 5, 6

    myList.remove(); // removing from the front of the list
    assertEquals(5, myList.size()); //List looks like 2, 3, 4, 5, 6

    myList.reverse();
    assertEquals("6, 5, 4, 3, 2", myList.print());
    assertEquals(5, myList.size());
}
```

Runs: 16/16 Errors: 0 Failures: 0

- LinkedListTDDTest [Runner: JUnit 5] (0.061 s)
  - testDeletingMoreElementsFromList() (0.036 s)
  - testReverseManyElementsWithAddFirst() (0.002 s)
  - testRemoveSomeMore() (0.001 s)
  - testReverseEmptyList() (0.002 s)
  - testLinkedListInsertionWithAddFirst() (0.001 s)
  - testLinkedListIsCreatedEmpty() (0.001 s)
  - testLinkedListInsertionTwoElements() (0.001 s)
  - testReverseListWithMoreElements2() (0.001 s)
  - testReverseListWithMoreElements() (0.001 s)

```java
}

@Test
void testRemoveSomeMore2() {
    myList.addFirst("a");
    myList.addFirst("b");
    myList.addFirst("c");
    myList.addFirst("c");
    myList.remove();
    myList.addFirst("c");
    myList.remove();
    myList.remove();
    // after removing all the C's
    assertEquals("b, a", myList.print());
}
```

We are only operating with a couple elements in this exercise, but if the list had thousands of elements, we would prefer to use the addFirst method since it always puts the new elements at the beginning.

There is another way to test the reverse. But this method needs an index (get method). Basically we create a new list, reverse one of the two lists and then iterate over both indexes. For one of those we start at the last element and iterate to the first and on the other list we start with the first index and iterate to the last.

```
@Test
public void testReverse_bigList() {
    givenAListContaining( ...elements: "a", "b", "c", "d", "e");

    list.reverse();

    for (int i = 0; i < list.size(); i++) {
        Assert.assertEquals(nonReversedList.get(list.size()-1-i), list.get(i));
    }
}
```

```
private void givenAListContaining(String... elements) {
    for (String s : elements) {
        list.add(s);
        nonReversedList.add(s);
    }
}
```

The way of getting to the index is the get() method. These two methods are from the video:https://www.youtube.com/watch?v=q6R2ZhQ9VwQ.

```
public T get(int index) {
    Node node = first;
    while (index > 0) {
        index--;
        node = node.next;
    }
    return node.value;
}
```

```
public int size() {
    int count = 0;
    Node node = first;
    while (node != null) {
        count++;
        node = node.next;
    }
    return count;
}
```

Some simpler tests for 2 elements:

```java
@Test
public void testReverse_twoElementList_secondElementIsFirst() {
    givenAListContaining( ...elements: "a", "b");
    list.reverse();
    Assert.assertEquals( expected: "b", list.get(0));
}

@Test
public void testReverse_twoElementList_firstElementIsSecond() {
    givenAListContaining( ...elements: "a", "b");
    list.reverse();
    Assert.assertEquals( expected: "a", list.get(1));
}
```

Reflection

Niklas: There was a lot of reference material here and that made our work much easier. The test first driven development is definitely an interesting concept and I do see the advantages that it has. You have to think a lot more about what you actually want to code and what your code has to do since you test it before actually writing it. It was fun to revisit the linked list concept and look at it from a different perspective.

Pavel: I think this was a really nice exercise to learn more about test-driven development. I really enjoyed it, because it showed me a different approach to implementing different methods. Since I was using the new version of JUnit to write the test cases, I learned more about the changes in the annotations that they made when updating from JUnit 4 to JUnit 5. There were also some newer assertion calls, but I didn't get to use those this time.

Robin: For getting into the test driven development i started with the video https://www.youtube.com/watch?v=q6R2ZhQ9VwQ. It gave a good introduction into the test-driven workflow plus it made another way of testing a reverse list possible. It is really impressive how much time can be saved by testing in the correct environment. This still requires proper test programming but those are probably way more simple than some giant algorithms. Also writing those tests are more about the result than the implementation so you actually have to think about what you really want first.

Nataliia:

TDD is a very interesting and proper way of writing a clean and nice code. A person should think about the good structure of the code, about all methods that will be implemented, before writing the code. But as far as I understood, not many developers apply this approach in real life for many reasons. Probably if all people use TDD approach, then we would live in better world with a better software. I still wonder if it's possible NOT to refactor tests AT ALL after they were written and you write the code. It seems like it's impossible for me by now, but I will work on my testing-and-then-coding skills.

Time spent:

Exercise	Time needed
TDD Tests for size, insert	**30 min**
TDD Tests for remove	**45 min**
TDD Tests for reverse and implementation of reverse	**3 hours**
Additional tests	**1 hour**

Appendix

LinkedListTDD class

```
package Lab8;

public class LinkedListTDD {
	private Node head, curr, prev;
	private int size = 0;

	public int size() {
		return size;
	}

	public void insert(Object data) {

		if (head == null) {
			head = new Node(data, null);
			curr = head;
		}
		else {
```

```
                Node temp = head;
                while (temp.next != null) {
                        temp = temp.next;
                }
                temp.next = new Node(data);
            }
            size++;
        }
        public void add(Object obj) {
        if (head == null) {
            head = new Node(obj, null);
            curr = head;
        } else {
            if (curr == null){
                curr = new Node(obj, curr);
                prev.next = curr;
                prev = curr;
            } else {
                Node tmp = new Node (obj, curr.next);
                curr.next = tmp;
                prev      = curr;
                curr      = tmp;
            }
        }
        size += 1;
    }

        public void addFirst(Object o){
                Node tmp = new Node (o, head);
                head = tmp;
                prev = null;
                curr = head;
                size++;
        }

        public void remove() {
        if (this.isEmpty() || curr == null) {
            return;
        } else {
            if (prev == null) {
                head = curr.next;
                curr = head;
            } else {
                prev.next = curr.next;
                curr = curr.next;
            }
            size -= 1;
        }
```

```
}

    public void reverse() {
            if (head == null) {
                    return;
            }

            Node curr = head;
            Node temp;
            while (curr.next != null) {
                    temp = curr.next;
                    curr.next = temp.next ;
                    temp.next = head;
                    head = temp;
            }
    }

    /*
     * Methods to deal with the print
     */
    public void reset() {
             curr = head;
             prev = null;
             // note: this only resets the running pointer, not the list!
             // size remains the same
            }

            public boolean eol() {
             return (curr == null);
            }

            public void succ() {
             curr = curr.next;
             if (prev == null)
               prev = head;
             else
               prev = prev.next;
            }

            // If empty strings are given, use a list default
            public String print() {
                    String s = "";
                    for (this.reset();!this.eol();this.succ()){
                            // this wastes a lot of system data
                            s = s + curr.data + ((curr.next != null)?", ":"");
                    }
                    return s;
            }
```

```
		public boolean isEmpty() {
			return head == null;
		}
}
```

LinkedListTDDTest class

```
package Lab8;

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

class LinkedListTDDTest {

	LinkedListTDD myList;
	// AHHHHHHH, in JUnit 5 it's not @Before, but @BeforeEach ... to have a setUp
that runs before each test
	@BeforeEach
	void setUp() {
	myList = new LinkedListTDD();
	}

	@Test
	void testLinkedListIsCreatedEmpty() {
		assertEquals(0, myList.size());
	}

	@Test
	void testLinkedListInsertionOneElement() {
		myList.insert("great this works");
		assertEquals("great this works", myList.print());
		assertEquals(1, myList.size());
	}

	@Test
	void testLinkedListInsertionTwoElements() {
		myList.insert("great this works");
		myList.insert("with two elements");
		assertEquals("great this works, with two elements", myList.print());
		assertEquals(2, myList.size());
	}

	@Test
	void testLinkedListInsertionTwoIntegers() {
		myList.insert(1);
		myList.insert(2);
```

```
        assertEquals("1, 2", myList.print());
        assertEquals(2, myList.size());
    }

    @Test
    void testLinkedListInsertionWithAddFirst() {
        myList.addFirst("pass one");
        myList.addFirst("pass two");
        assertEquals("pass two, pass one", myList.print());
        assertEquals(2, myList.size());

        myList.addFirst("pass three");
        assertEquals("pass three, pass two, pass one", myList.print());
        assertEquals(3, myList.size());
    }

    @Test
    void testDeletingOneElementFromList() {
        myList.insert("delete this");
        myList.insert("delete this2");
        myList.remove();
        assertEquals("delete this2", myList.print());
    }

    @Test
    void testDeletingMoreElementsFromList() {
        myList.addFirst("first");
        myList.remove();
        myList.addFirst("second");
        myList.addFirst("third");
        myList.addFirst("fourth");
        myList.remove();
        myList.remove();
        assertEquals("second", myList.print());
        assertEquals(1, myList.size());
    }

    @Test
    void testReverseEmptyList() {
        myList.reverse();
        assertEquals("", myList.print());
    }

    @Test
    void testReverseListWithOneElement() {
        myList.insert("first");
        myList.reverse();
        assertEquals("first", myList.print());
    }
```

```
@Test
void testReverseListWithMoreElements() {
        myList.insert("first");
        myList.insert("second");
        //List looks like: first, second
        myList.reverse();
        assertEquals("second, first", myList.print());
}

@Test
void testReverseListWithMoreElements2() {
        myList.insert("first");
        myList.insert("second");
        myList.insert("third");
        myList.insert("fourth");
        //List looks like: first, second, third, fourth
        myList.reverse();
        assertEquals("fourth, third, second, first", myList.print());
}

@Test
void testReverseManyElementsWithAddFirst() {
        myList.addFirst(1);
        myList.addFirst(2);
        myList.addFirst(3);
        myList.addFirst(4);
        myList.addFirst(5);
        assertEquals(5, myList.size());
        // List looks like 5,4,3,2,1

        myList.remove();
        assertEquals(4, myList.size());
        // List looks like 4,3,2,1

        myList.reverse();

        assertEquals("1, 2, 3, 4", myList.print());
        assertEquals(4, myList.size());
}

@Test
void testReverseManyElementsWithInsertAndRemove() {
        myList.insert(1);
        myList.insert(2);
        myList.insert(3);
        myList.insert(4);
        myList.insert(5);
        myList.insert(6);
        assertEquals(6, myList.size());
        //List looks like 1, 2, 3, 4, 5, 6 (adding at the end)
```

```
        myList.remove(); // removing from the front of the list
        assertEquals(5, myList.size()); //List looks like 2, 3, 4, 5, 6

        myList.reverse();
        assertEquals("6, 5, 4, 3, 2", myList.print());
        assertEquals(5, myList.size());
    }

    @Test
    void testIsEmpty() {
        assertEquals(true, myList.isEmpty());
    }

    @Test
    void testRemoveSomeMore() {
        myList.addFirst("a");
        myList.remove();
        myList.addFirst("b");
        myList.remove();
        myList.addFirst("c");
        myList.remove();
        myList.addFirst("d");
        myList.remove();
        myList.addFirst("e");
        myList.remove();
        assertEquals("", myList.print());
    }

    @Test
    void testRemoveSomeMore2() {
        myList.addFirst("a");
        myList.addFirst("b");
        myList.addFirst("c");
        myList.addFirst("c");
        myList.remove();
        myList.addFirst("c");
        myList.remove();
        myList.remove();
        // after removing all the C's
        assertEquals("b, a", myList.print());
    }

    @Test
    void testAdditional() {
        myList.addFirst("one");
        myList.addFirst("one");
        myList.remove();
        myList.remove();
        myList.remove();
```

```
                myList.addFirst("one");
                myList.reverse();
                assertEquals("one", myList.print());
        }
}
```

Node Class

```
package Lab8;

public class Node {

        Object data;
        Node next;

        public Node(Object data, Node next) {
                this.data = data;
                this.next = next;
        }

        public Node(Object data) {
                this.data = data;
        }
}
```

LinkedList (Different implementation)

```
package LinkedList;

public class LinkedList<T> {

   private Node first;

   public int size() {
      int count = 0;
      Node node = first;
      while (node != null) {
         count++;
         node = node.next;
      }
      return count;
   }
```

```
public void add(T value) {
    if (first == null) {
        first = new Node(value);
    } else {
        Node temp = first;
        while (temp.next != null) {
            temp = temp.next;
        }
        temp.next = new Node(value);
    }
}

public void addFirst(T value) {
   if (first == null) {
       first = new Node(value);
   } else {
       Node temp = first;
       first = new Node(value);
       first.next = temp;
   }
}

public T get(int index) {
    Node node = first;
    while (index > 0) {
        index--;
        node = node.next;
    }
    return node.value;
}

public T remove(int index) {
    Node node = first;
    Node previous = null;
    while (index > 0) {
        index--;
        previous = node;
        node = node.next;
    }
    if (previous == null) {
        first = node.next;
    } else {
        previous.next = node.next;
    }
    return node.value;
}

public void reverse() {
    if (first == null) {
        return;
```

```
        }

        Node curr = first;
        Node temp = curr.next;
        while (curr.next != null) {
            temp = curr.next;
            curr.next = temp.next ; // points to null
            temp.next = first;
            first = temp;
        }
    }

    private class Node {
        private final T value;
        private Node next;
        public Node(T value) {
            this.value = value;
        }
    }
}
```

LinkedListTest (Different implementation)

```
package LinkedList;

import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;

public class LinkedListTest {

    private LinkedList<String> list;
    private LinkedList<String> nonReversedList;



    @Before
    public void setUp() {
        list = new LinkedList<String>();
        nonReversedList = new LinkedList<String>();
    }

    @Test
    public void testSize_initialList() {
        Assert.assertEquals(0, list.size());
    }
```

```
@Test
public void testSize_oneElement_sizeIsOne() {
    givenAListContaining("a");
    Assert.assertEquals(1, list.size());
}

private void givenAListContaining(String... elements) {
    for (String s : elements) {
        list.add(s);
        nonReversedList.add(s);
    }
}

@Test
public void testGet_oneElement() {
    givenAListContaining("a"); //given when then
    String result = list.get(0);
    Assert.assertEquals("a", result);
}

@Test
public void testSize_addSecondElement() {
    givenAListContaining("a");
    list.add("b");
    Assert.assertEquals(2, list.size());
}

@Test
public void testGet_firstElementFromTwoElementList() {
    givenAListContaining("a", "b");
    String result = list.get(0);
    Assert.assertEquals("a", result);
}

@Test
public void testGet_secondElementFromTwoElementList() {
    givenAListContaining("a", "b");
    String result = list.get(1);
    Assert.assertEquals("b", result);
}

@Test
public void testRemove_firstElementFromTwoElementList_elementWasTheFirst() {
    givenAListContaining("a", "b");
    String result = list.remove(0);
    Assert.assertEquals("a", result);
}

@Test
public void testRemove_firstElementFromTwoElementList_sizeIsOne() {
```

```
        givenAListContaining("a", "b");
        list.remove(0);
        Assert.assertEquals(1, list.size());
    }

    @Test
    public void
testRemove_firstElementFromTwoElementList_firstElementIsOldSecondElement() {
        givenAListContaining("a", "b");
        list.remove(0);
        Assert.assertEquals("b", list.get(0));
    }

    @Test
    public void testReverse_twoElementList_secondElementIsFirst() {
        givenAListContaining("a", "b");
        list.reverse();
        Assert.assertEquals("b", list.get(0));
    }

    @Test
    public void testReverse_twoElementList_firstElementIsSecond() {
        givenAListContaining("a", "b");
        list.reverse();
        Assert.assertEquals("a", list.get(1));
    }

    @Test
    public void testReverse_bigList() {
        givenAListContaining("a", "b", "c", "d", "e");

        list.reverse();

        for (int i = 0; i < list.size(); i++) {
            Assert.assertEquals(nonReversedList.get(list.size()-1-i), list.get(i));
        }
    }

    @Test
    public void testReverse_bigList_deletaAndAdd() {
        givenAListContaining("a", "b", "c", "d", "e");

        list.add("a");
        nonReversedList.add("a");
        list.remove(3);
        nonReversedList.remove(3);
        list.reverse();

        for (int i = 0; i < list.size(); i++) {
            Assert.assertEquals(nonReversedList.get(list.size()-1-i), list.get(i));
```

```
    }
  }
@Test
public void testAddFirst() {
  givenAListContaining("a", "b", "c", "d", "e");
  list.addFirst("first");
  Assert.assertEquals("first", list.get(0));
}

@Test
public void testAddFirst_lengthTest() {
  givenAListContaining("a", "b", "c", "d", "e");
  list.addFirst("first");
  Assert.assertEquals(6, list.size());
}
}
```