CSE 2231 – Software 2: Software Development and Design

Professor: Rob LaTour

Project #5: SortingMachine with Heapsort

The Ohio State University

College of Engineering

Columbus, Ohio

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

import java.util.Comparator;

import org.junit.Test;

import components.sortingmachine.SortingMachine;

/**
 * JUnit test fixture for {@code SortingMachine<String>}'s constructor and
 * kernel methods.
 *
 * @author Danny Kan (kan.74@osu.edu)
 * @author Jatin Mamtani (mamtani.6@osu.edu)
 *
 */
public abstract class SortingMachineTest {

    /**
     * Invokes the appropriate {@code SortingMachine} constructor for the
     * implementation under test and returns the result.
     *
     * @param order
     *            the {@code Comparator} defining the order for {@code String}
     * @return the new {@code SortingMachine}
     * @requires IS_TOTAL_PREORDER([relation computed by order.compare method])
     * @ensures constructorTest = (true, order, {})
     */
    protected abstract SortingMachine<String> constructorTest(
            Comparator<String> order);

    /**
     * Invokes the appropriate {@code SortingMachine} constructor for the
     * reference implementation and returns the result.
     *
     * @param order
     *            the {@code Comparator} defining the order for {@code String}
     * @return the new {@code SortingMachine}
     * @requires IS_TOTAL_PREORDER([relation computed by order.compare method])
     * @ensures constructorRef = (true, order, {})
     */
    protected abstract SortingMachine<String> constructorRef(
            Comparator<String> order);

    /**
     *
     * Creates and returns a {@code SortingMachine<String>} of the
     * implementation under test type with the given entries and mode.
     *
     * @param order
     *            the {@code Comparator} defining the order for {@code String}
     * @param insertionMode
     *            flag indicating the machine mode
     * @param args
     *            the entries for the {@code SortingMachine}
     * @return the constructed {@code SortingMachine}
     * @requires IS_TOTAL_PREORDER([relation computed by order.compare method])
     * @ensures <pre>
     * createFromArgsTest = (insertionMode, order, [multiset of entries in args])
     * </pre>
     */
    private SortingMachine<String> createFromArgsTest(Comparator<String> order,
            boolean insertionMode, String... args) {
        SortingMachine<String> sm = this.constructorTest(order);
        for (int i = 0; i < args.length; i++) {
            sm.add(args[i]);
        }
        if (!insertionMode) {
            sm.changeToExtractionMode();
        }
        return sm;
    }

    /**
     *
     * Creates and returns a {@code SortingMachine<String>} of the reference
     * implementation type with the given entries and mode.
     *
     * @param order
     *            the {@code Comparator} defining the order for {@code String}
     * @param insertionMode
     *            flag indicating the machine mode
     * @param args
     *            the entries for the {@code SortingMachine}
     * @return the constructed {@code SortingMachine}
     * @requires IS_TOTAL_PREORDER([relation computed by order.compare method])
     * @ensures <pre>
     * createFromArgsRef = (insertionMode, order, [multiset of entries in args])
     * </pre>
     */
    private SortingMachine<String> createFromArgsRef(Comparator<String> order,
            boolean insertionMode, String... args) {
        SortingMachine<String> sm = this.constructorRef(order);
        for (int i = 0; i < args.length; i++) {
            sm.add(args[i]);
        }
        if (!insertionMode) {
            sm.changeToExtractionMode();
        }
        return sm;
    }

    /**
     * Comparator<String> implementation to be used in all test cases. Compare
     * {@code String}s in lexicographic order.
     */
    private static class StringLT implements Comparator<String> {

        @Override
        public int compare(String s1, String s2) {
            return s1.compareToIgnoreCase(s2);
        }

    }

    /**
     * Comparator instance to be used in all test cases.
     */
    private static final StringLT ORDER = new StringLT();

    /*
     * Complete and Systematic Test Cases:
     */

    /**
     * Testing the constructor.
     */
    @Test
```

```
public final void testConstructor() {
    SortingMachine<String> actual = this.constructorTest(ORDER);
    SortingMachine<String> expected = this.constructorRef(ORDER);
    assertEquals(expected, actual);
}

/*
 * Kernel Methods --->
 */

/*
 * Testing .add() in this section:=
 */

/**
 * Testing .add() ONCE (i.e., 1x) to EMPTY {@code SortingMachine<String>}.
 */
@Test
public final void testAddToEmptyV1() {
    SortingMachine<String> actual = this.createFromArgsTest(ORDER, true);
    SortingMachine<String> expected = this.createFromArgsRef(ORDER, true,
            "red");
    actual.add("red");
    assertEquals(expected, actual);
}

/**
 * Testing .add() TWICE (i.e., 2x) to EMPTY {@code SortingMachine<String>}.
 */
@Test
public final void testAddToEmptyV2() {
    SortingMachine<String> actual = this.createFromArgsTest(ORDER, true);
    SortingMachine<String> expected = this.createFromArgsRef(ORDER, true,
            "red", "green");
    actual.add("red");
    actual.add("green");
    assertEquals(expected, actual);
}

/**
 * Testing .add() ONCE (i.e., 1x) to NON-EMPTY
 * {@code SortingMachine<String>} with one (1) {@code String}..
 */
@Test
public final void testAddToNonEmptyV1() {
    SortingMachine<String> actual = this.createFromArgsTest(ORDER, true,
            "red");
    SortingMachine<String> expected = this.createFromArgsRef(ORDER, true,
            "red", "green");
    actual.add("green");
    assertEquals(expected, actual);
}

/**
 * Testing .add() TWICE (i.e., 2x) to NON-EMPTY
 * {@code SortingMachine<String>} with one (1) {@code String}..
 */
@Test
public final void testAddToNonEmptyV2() {
    SortingMachine<String> actual = this.createFromArgsTest(ORDER, true,
            "red");
    SortingMachine<String> expected = this.createFromArgsRef(ORDER, true,
            "red", "green", "blue");
    actual.add("green");
    actual.add("blue");
    assertEquals(expected, actual);
}

/*
 * Testing .changeToExtractionMode() in this section:=
 */

/**
 * Testing .changeToExtractionMode() ONCE (i.e., 1x) on an EMPTY
 * {@code SortingMachine<String>}.
 */
@Test
public final void testChangeToExtractionModeOnEmpty() {
    SortingMachine<String> actual = this.createFromArgsTest(ORDER, true);
    SortingMachine<String> expected = this.createFromArgsRef(ORDER, false);
    actual.changeToExtractionMode();
    assertEquals(expected, actual);
}

/**
 * Testing .changeToExtractionMode() ONCE (i.e., 1x) on a NON-EMPTY
 * {@code SortingMachine<String>} with one (1) {@code String}.
 */
@Test
public final void testChangeToExtractionModeOnNonEmpty() {
    SortingMachine<String> actual = this.createFromArgsTest(ORDER, true,
            "red");
    SortingMachine<String> expected = this.createFromArgsRef(ORDER, false,
            "red");
    actual.changeToExtractionMode();
    assertEquals(expected, actual);
```

```
}

/*
 * Testing .removeFirst() in this section:=
 */

/**
 * Testing .removeFirst() ONCE (i.e., 1x) to EMPTY
 * {@code SortingMachine<String>}.
 */
@Test
public final void testRemoveFirstToEmptyV1() {
    SortingMachine<String> actual = this.createFromArgsTest(ORDER, false,
            "red");
    SortingMachine<String> expected = this.createFromArgsRef(ORDER, false);
    assertEquals("red", actual.removeFirst());
    assertEquals(expected, actual);
}

/**
 * Testing .removeFirst() TWICE (i.e., 2x) to EMPTY
 * {@code SortingMachine<String>}.
 */
@Test
public final void testRemoveFirstToEmptyV2() {
    SortingMachine<String> actual = this.createFromArgsTest(ORDER, false,
            "red", "green");
    SortingMachine<String> expected = this.createFromArgsRef(ORDER, false);
    assertEquals("green", actual.removeFirst());
    assertEquals("red", actual.removeFirst());
    assertEquals(expected, actual);
}
```

```
/**
 * Testing .removeFirst() ONCE (i.e., 1x) to NON-EMPTY
 * {@code SortingMachine<String>} with one (1) {@code String}.
 */
@Test
public final void testRemoveFirstToNonEmptyV1() {
    SortingMachine<String> actual = this.createFromArgsTest(ORDER, false,
            "red", "green");
    SortingMachine<String> expected = this.createFromArgsRef(ORDER, false,
            "red");
    assertEquals("green", actual.removeFirst());
    assertEquals(expected, actual);
}

/**
 * Testing .removeFirst() TWICE (i.e., 2x) to NON-EMPTY
 * {@code SortingMachine<String>} with one (1) {@code String}.
 */
@Test
public final void testRemoveFirstToNonEmptyV2() {
    SortingMachine<String> actual = this.createFromArgsTest(ORDER, false,
            "red", "green", "blue");
    SortingMachine<String> expected = this.createFromArgsRef(ORDER, false,
            "red");
    assertEquals("blue", actual.removeFirst());
    assertEquals("green", actual.removeFirst());
    assertEquals(expected, actual);
}

/*
 * Testing .isInInsertionMode() in this section:=
 */

/**
 * Testing .isInInsertionMode() on an EMPTY {@code SortingMachine<String>}
 * while insertion_mode: true.
 */
@Test
public final void testIsInInsertionModeOnEmptyTrue() {
    SortingMachine<String> actual = this.createFromArgsTest(ORDER, true);
    SortingMachine<String> expected = this.createFromArgsRef(ORDER, true);
    assertEquals(true, actual.isInInsertionMode());
    assertEquals(expected, actual);
}

/**
 * Testing .isInInsertionMode() on an EMPTY {@code SortingMachine<String>}
 * while insertion_mode: false.
 */
@Test
public final void testIsInInsertionModeOnEmptyFalse() {
    SortingMachine<String> actual = this.createFromArgsTest(ORDER, false);
    SortingMachine<String> expected = this.createFromArgsRef(ORDER, false);
    assertEquals(false, actual.isInInsertionMode());
    assertEquals(expected, actual);
}

/**
 * Testing .isInInsertionMode() on a NON-EMPTY
 * {@code SortingMachine<String>} with one (1) {@code String} while
 * insertion_mode: true.
 */
@Test
```

```
public final void testIsInInsertionModeOnNonEmptyTrue() {
    SortingMachine<String> actual = this.createFromArgsTest(ORDER, true,
            "red");
    SortingMachine<String> expected = this.createFromArgsRef(ORDER, true,
            "red");
    assertEquals(true, actual.isInInsertionMode());
    assertEquals(expected, actual);
}

/**
 * Testing .isInInsertionMode() on a NON-EMPTY
 * {@code SortingMachine<String>} with one (1) {@code String} while
 * insertion_mode: false.
 */
@Test
public final void testIsInInsertionModeOnNonEmptyFalse() {
    SortingMachine<String> actual = this.createFromArgsTest(ORDER, false,
            "red");
    SortingMachine<String> expected = this.createFromArgsRef(ORDER, false,
            "red");
    assertEquals(false, actual.isInInsertionMode());
    assertEquals(expected, actual);
}

/*
 * Testing .order() in this section:=
 */

/**
 * Testing .order() on an EMPTY {@code SortingMachine<String>} while
 * insertion_mode: true.
 */
```

```
@Test
public final void testOrderOnEmptyTrue() {
    SortingMachine<String> actual = this.createFromArgsTest(ORDER, true);
    SortingMachine<String> expected = this.createFromArgsRef(ORDER, true);
    assertEquals(ORDER, actual.order());
    assertEquals(expected, actual);
}

/**
 * Testing .order() on an EMPTY {@code SortingMachine<String>} while
 * insertion_mode: false.
 */
@Test
public final void testOrderOnEmptyFalse() {
    SortingMachine<String> actual = this.createFromArgsTest(ORDER, false);
    SortingMachine<String> expected = this.createFromArgsRef(ORDER, false);
    assertEquals(ORDER, actual.order());
    assertEquals(expected, actual);
}

/**
 * Testing .order() on a NON-EMPTY {@code SortingMachine<String>} with one
 * (1) {@code String} while insertion_mode: true.
 */
@Test
public final void testOrderOnNonEmptyTrue() {
    SortingMachine<String> actual = this.createFromArgsTest(ORDER, true,
            "red");
    SortingMachine<String> expected = this.createFromArgsRef(ORDER, true,
            "red");
    assertEquals(ORDER, actual.order());
    assertEquals(expected, actual);
}

/**
 * Testing .order() on a NON-EMPTY {@code SortingMachine<String>} with one
 * (1) {@code String} while insertion_mode: false.
 */
@Test
public final void testOrderOnNonEmptyFalse() {
    SortingMachine<String> actual = this.createFromArgsTest(ORDER, false,
            "red");
    SortingMachine<String> expected = this.createFromArgsRef(ORDER, false,
            "red");
    assertEquals(ORDER, actual.order());
    assertEquals(expected, actual);
}

/*
 * Testing .size() in this section:=
 */

/**
 * Testing .size() on an EMPTY {@code SortingMachine<String>} while
 * insertion_mode: true.
 */
@Test
public final void testSizeOnEmptyTrue() {
    SortingMachine<String> actual = this.createFromArgsTest(ORDER, true);
    SortingMachine<String> expected = this.createFromArgsRef(ORDER, true);
    assertEquals(0, actual.size());
    assertEquals(expected, actual);
}
```

```
/**
 * Testing .size() on an EMPTY {@code SortingMachine<String>} while
 * insertion_mode: false.
 */
@Test
public final void testSizeOnEmptyFalse() {
    SortingMachine<String> actual = this.createFromArgsTest(ORDER, false);
    SortingMachine<String> expected = this.createFromArgsRef(ORDER, false);
    assertEquals(0, actual.size());
    assertEquals(expected, actual);
}

/**
 * Testing .size() on a NON-EMPTY {@code SortingMachine<String>} with one
 * (1) {@code String} while insertion_mode: true.
 */
@Test
public final void testSizeOnNonEmptyTrueV1() {
    SortingMachine<String> actual = this.createFromArgsTest(ORDER, true,
            "red");
    SortingMachine<String> expected = this.createFromArgsRef(ORDER, true,
            "red");
    assertEquals(1, actual.size());
    assertEquals(expected, actual);
}

/**
 * Testing .size() on a NON-EMPTY {@code SortingMachine<String>} with three
 * (3) {@code String} while insertion_mode: true.
 */
@Test
public final void testSizeOnNonEmptyTrueV2() {
    SortingMachine<String> actual = this.createFromArgsTest(ORDER, true,
            "red", "green", "blue");
    SortingMachine<String> expected = this.createFromArgsRef(ORDER, true,
            "red", "green", "blue");
    assertEquals(3, actual.size());
    assertEquals(expected, actual);
}

/**
 * Testing .size() on a NON-EMPTY {@code SortingMachine<String>} with one
 * (1) {@code String} while insertion_mode: false.
 */
@Test
public final void testSizeOnNonEmptyFalseV1() {
    SortingMachine<String> actual = this.createFromArgsTest(ORDER, false,
            "red");
    SortingMachine<String> expected = this.createFromArgsRef(ORDER, false,
            "red");
    assertEquals(1, actual.size());
    assertEquals(expected, actual);
}

/**
 * Testing .size() on a NON-EMPTY {@code SortingMachine<String>} with three
 * (3) {@code String} while insertion_mode: false.
 */
@Test
public final void testSizeOnNonEmptyFalseV2() {
    SortingMachine<String> actual = this.createFromArgsTest(ORDER, false,
            "red", "green", "blue");
    SortingMachine<String> expected = this.createFromArgsRef(ORDER, false,
            "red", "green", "blue");
```

```
        assertEquals(3, actual.size());
        assertEquals(expected, actual);
    }

    /*
     * Integration Testing (NOT REQUIRED):
     */

}
```