CSE 2231 – Software 2: Software Development and Design

Professor: Rob LaTour

Project #3: Hashing Implementation of Map

The Ohio State University

College of Engineering

Columbus, Ohio

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

import org.junit.Test;

import components.map.Map;
import components.map.Map.Pair;

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

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

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

    /**
     *
     * Creates and returns a {@code Map<String, String>} of the implementation
     * under test type with the given entries.
     *
     * @param args
     *            the (key, value) pairs for the map
     * @return the constructed map
     * @requires <pre>
     * [args.length is even]  and
     * [the 'key' entries in args are unique]
     * </pre>
     * @ensures createFromArgsTest = [pairs in args]
     */
    private Map<String, String> createFromArgsTest(String... args) {
        assert args.length % 2 == 0 : "Violation of: args.length is even";
        Map<String, String> map = this.constructorTest();
        for (int i = 0; i < args.length; i += 2) {
            assert !map.hasKey(args[i]) : ""
                    + "Violation of: the 'key' entries in args are unique";
            map.add(args[i], args[i + 1]);
        }
        return map;
    }

    /**
     *
     * Creates and returns a {@code Map<String, String>} of the reference
     * implementation type with the given entries.
     *
     * @param args
     *           the (key, value) pairs for the map
     * @return the constructed map
     * @requires <pre>
     * [args.length is even]  and
     * [the 'key' entries in args are unique]
     * </pre>
     * @ensures createFromArgsRef = [pairs in args]
     */
    private Map<String, String> createFromArgsRef(String... args) {
        assert args.length % 2 == 0 : "Violation of: args.length is even";
        Map<String, String> map = this.constructorRef();
        for (int i = 0; i < args.length; i += 2) {
            assert !map.hasKey(args[i]) : ""
                    + "Violation of: the 'key' entries in args are unique";
            map.add(args[i], args[i + 1]);
        }
        return map;
    }

    /*
     * Complete and Systematic Test Cases:
     */

    /**
     * Testing the no-argument constructor.
     */
    @Test
    public final void testNoArgumentConstructor() {
        Map<String, String> m = this.constructorTest();
        Map<String, String> mExpected = this.constructorRef();
        assertEquals(mExpected, m);
    }

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

    /**
     * Testing .add() to empty {@code Map<String, String>}.
     */
    @Test
    public final void testAddOnceToEmpty() {
        Map<String, String> m = this.createFromArgsTest();
        Map<String, String> mExpected = this.createFromArgsRef("one", "1");
        m.add("one", "1");
        assertEquals(mExpected, m);
    }

    /**
     * Testing .add() to non-empty {@code Map<String, String>}.
     */
    @Test
    public final void testAddOnceToNonEmpty() {
        Map<String, String> m = this.createFromArgsTest("one", "1");
        Map<String, String> mExpected = this.createFromArgsRef("one", "1",
                "two", "2");
        m.add("two", "2");
        assertEquals(mExpected, m);
    }

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

    /**
     * Testing .remove() to empty {@code Map<String, String>}.
     */
    @Test
    public final void testRemoveToEmpty() {
        Map<String, String> m = this.createFromArgsTest("one", "1");
        Map<String, String> mExpected = this.createFromArgsRef();
        Pair<String, String> p = m.remove("one");
        assertEquals("one", p.key());
        assertEquals("1", p.value());
        assertEquals(mExpected, m);
    }

    /**
     * Testing .remove() to non-empty {@code Map<String, String>}.
     */
    @Test
    public final void testRemoveToNonEmpty() {
        Map<String, String> m = this.createFromArgsTest("one", "1", "two", "2");
        Map<String, String> mExpected = this.createFromArgsRef("one", "1");
        Pair<String, String> p = m.remove("two");
        assertEquals("two", p.key());
        assertEquals("2", p.value());
        assertEquals(mExpected, m);
    }

    /*
     * Testing .removeAny() in this section:=
     */
```

```
/**
 * Testing .removeAny() to empty {@code Map<String, String>}.
 */
@Test
public final void testRemoveAnyToEmpty() {
    /*
     * http://web.cse.ohio-state.edu/software/2231/web-sw2/assignments/
     * homeworks/set-on-queue/test-removeany.html
     */
    Map<String, String> m = this.createFromArgsTest("one", "1");
    Map<String, String> mExpected = this.createFromArgsRef("one", "1");
    Pair<String, String> p1 = m.removeAny();
    assertEquals(true, mExpected.hasKey(p1.key()));
    Pair<String, String> p2 = mExpected.remove(p1.key());
    assertEquals(p2, p1);
    assertEquals(mExpected, m);
}

/**
 * Testing .removeAny() to non-empty {@code Map<String, String>}.
 */
@Test
public final void testRemoveAnyToNonEmpty() {
    /*
     * http://web.cse.ohio-state.edu/software/2231/web-sw2/assignments/
     * homeworks/set-on-queue/test-removeany.html
     */
    Map<String, String> m = this.createFromArgsTest("one", "1", "two", "2");
    Map<String, String> mExpected = this.createFromArgsRef("one", "1",
            "two", "2");
    Pair<String, String> p1 = m.removeAny();
    assertEquals(true, mExpected.hasKey(p1.key()));
    Pair<String, String> p2 = mExpected.remove(p1.key());
    assertEquals(p2, p1);
    assertEquals(mExpected, m);
}

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

/**
 * Testing .value() on {@code Map<String, String>} with one (1)
 * {@code Map.Pair<String, String>}.
 */
@Test
public final void testValueOnOnePair() {
    Map<String, String> m = this.createFromArgsTest("one", "1");
    Map<String, String> mExpected = this.createFromArgsRef("one", "1");
    assertEquals("1", m.value("one"));
    assertEquals(mExpected, m);
}

/**
 * Testing .value() on {@code Map<String, String>} with three (3)
 * {@code Map.Pair<String, String>} containing identical values.
 */
@Test
public final void testValueOnThreePairsV2() {
    /*
     * Note: {@code Map<K, V>} is mathematically modeled as a finite set of
     * ordered pairs of type (K, V), such that it is a finite partial
     * function from K -> V.
     */
    Map<String, String> m = this.createFromArgsTest("one", "0", "two", "0",
            "three", "0");
    Map<String, String> mExpected = this.createFromArgsRef("one", "0",
            "two", "0", "three", "0");
    assertEquals("0", m.value("three"));
    assertEquals(mExpected, m);
}

/**
 * Testing .value() on {@code Map<String, String>} with three (3)
 * {@code Map.Pair<String, String>} containing different values.
 */
@Test
public final void testValueOnThreePairsV1() {
    /*
     * Note: {@code Map<K, V>} is mathematically modeled as a finite set of
     * ordered pairs of type (K, V), such that it is a finite partial
     * function from K -> V.
     */
    Map<String, String> m = this.createFromArgsTest("one", "1", "two", "2",
            "three", "3");
    Map<String, String> mExpected = this.createFromArgsRef("one", "1",
            "two", "2", "three", "3");
    assertEquals("3", m.value("three"));
    assertEquals(mExpected, m);
}

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

/**
```

```
 * Testing .hasKey() on an empty {@code Map<String, String>}, resulting in a
 * boolean expression evaluating to false.
 */
@Test
public final void testHasKeyOnEmpty() {
    Map<String, String> m = this.createFromArgsTest();
    Map<String, String> mExpected = this.createFromArgsRef();
    assertEquals(false, m.hasKey("one"));
    assertEquals(mExpected, m);
}

/**
 * Testing .hasKey() on a non-empty {@code Map<String, String>} with one (1)
 * {@code Map.Pair<String, String>}, resulting in a boolean expression
 * evaluating to true.
 */
@Test
public final void testHasKeyOnNonEmptyTrueV1() {
    Map<String, String> m = this.createFromArgsTest("one", "1");
    Map<String, String> mExpected = this.createFromArgsRef("one", "1");
    assertEquals(true, m.hasKey("one"));
    assertEquals(mExpected, m);
}

/**
 * Testing .hasKey() on a non-empty {@code Map<String, String>} with three
 * (3) {@code Map.Pair<String, String>}, resulting in a boolean expression
 * evaluating to true.
 */
@Test
public final void testHasKeyOnNonEmptyTrueV2() {
    Map<String, String> m = this.createFromArgsTest("one", "1", "two", "2",
            "three", "3");
    Map<String, String> mExpected = this.createFromArgsRef("one", "1",
            "two", "2", "three", "3");
    assertEquals(true, m.hasKey("three"));
    assertEquals(mExpected, m);
}

/**
 * Testing .hasKey() on a non-empty {@code Map<String, String>} with one (1)
 * {@code Map.Pair<String, String>}, resulting in a boolean expression
 * evaluating to false.
 */
@Test
public final void testHasKeyOnNonEmptyFalseV1() {
    Map<String, String> m = this.createFromArgsTest("one", "1");
    Map<String, String> mExpected = this.createFromArgsRef("one", "1");
    assertEquals(false, m.hasKey("three"));
    assertEquals(mExpected, m);
}

/**
 * Testing .hasKey() on a non-empty {@code Map<String, String>} with three
 * (3) {@code Map.Pair<String, String>}, resulting in a boolean expression
 * evaluating to false.
 */
@Test
public final void testHasKeyOnNonEmptyFalseV2() {
    Map<String, String> m = this.createFromArgsTest("one", "1", "two", "2",
            "three", "3");
    Map<String, String> mExpected = this.createFromArgsRef("one", "1",
            "two", "2", "three", "3");
    assertEquals(false, m.hasKey("zero"));
    assertEquals(mExpected, m);
}

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

/**
 * Testing .size() on an empty {@code Map<String, String>}.
 */
@Test
public final void testSizeOnEmpty() {
    Map<String, String> m = this.createFromArgsTest();
    Map<String, String> mExpected = this.createFromArgsRef();
    assertEquals(0, m.size());
    assertEquals(mExpected, m);
}

/**
 * Testing .size() on a non-empty {@code Map<String, String>} with one (1)
 * {@code Map.Pair<String, String>}.
 */
@Test
public final void testSizeOnNonEmptyV1() {
    Map<String, String> m = this.createFromArgsTest("one", "1");
    Map<String, String> mExpected = this.createFromArgsRef("one", "1");
    assertEquals(1, m.size());
    assertEquals(mExpected, m);
}

/**
 * Testing .size() on a non-empty {@code Map<String, String>} with three (3)
```

```
     * {@code Map.Pair<String, String>}.
     */
    @Test
    public final void testSizeOnNonEmptyV2() {
        Map<String, String> m = this.createFromArgsTest("one", "1", "two", "2",
                "three", "3");
        Map<String, String> mExpected = this.createFromArgsRef("one", "1",
                "two", "2", "three", "3");
        assertEquals(3, m.size());
        assertEquals(mExpected, m);
    }

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

}
```