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

/**
 * JUnit test fixture for {@code Map<String, String>}'s constructor and kernel
 * methods.
 *
 * @author Gabe Azzarita and Ty Fredrick
 *
 */
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;
    }

    // Testing default contructor
    @Test
    public final void testForEmptyConstructor() {
        Map<String, String> test = this.constructorTest();
        Map<String, String> ref = this.constructorRef();

        assertEquals(test, ref);
    }

    // Test constructor with arguments
    @Test
    public final void testForNonEmptyConstructor() {
        Map<String, String> test = this.createFromArgsTest("A", "B", "1", "2");
        Map<String, String> ref = this.createFromArgsRef("A", "B", "1", "2");

        assertEquals(test, ref);
    }

    // Testing add function on an empty map
    @Test
    public final void testForAddEmpty() {
        Map<String, String> test = this.createFromArgsTest();
        Map<String, String> ref = this.createFromArgsRef("A", "B");

        test.add("A", "B");

        assertEquals(test, ref);
    }

    // Testing add on a non-empty map
    @Test
    public final void testForAdd() {
        Map<String, String> test = this.createFromArgsTest("A", "B");
        Map<String, String> ref = this.createFromArgsRef("A", "B", "1", "2");

        test.add("1", "2");

        assertEquals(test, ref);
    }

    // Testing add with multiple add calls
    @Test
    public final void testForAddMultiple() {
        Map<String, String> test = this.createFromArgsTest();
        Map<String, String> ref = this.createFromArgsRef("A", "B", "1", "2");

        test.add("A", "B");
        test.add("1", "2");

        assertEquals(test, ref);
    }

    // Testing remove, and checking that it returns correct pair
    @Test
    public final void testForRemove() {
        Map<String, String> test = this.createFromArgsTest("A", "B");
        Map<String, String> ref = this.createFromArgsRef("A", "B");

        Pair<String, String> testRemoved = test.remove("A");
        Pair<String, String> refRemoved = ref.remove("A");

        assertEquals(test, ref);
        // Make sure remove function returns pair correctly
        assertEquals(testRemoved, refRemoved);
    }

    // Testing remove with multiple calls
    @Test
    public final void testForRemoveMultiple() {
        Map<String, String> test = this.createFromArgsTest("A", "B", "1", "2");
        Map<String, String> ref = this.createFromArgsRef();

        test.remove("1");
        test.remove("A");

        assertEquals(test, ref);
    }

    // Testing remove with multiple calls
    @Test
    public final void testForRemoveMultipleHard() {
        Map<String, String> test = this.createFromArgsTest("1", "i", "2", "ii",
                "3", "iii", "4", "iiij", "5", "v", "6", "vi", "7", "vii", "8",
                "viij", "9", "ix", "10", "x");
        Map<String, String> ref = this.createFromArgsRef("3", "iii", "4",
                "iiij", "5", "v", "6", "vi", "7", "vii");

        test.remove("1");
        test.remove("2");
        test.remove("8");
        test.remove("9");
        test.remove("10");
        assertEquals(test, ref);
    }

    // Testing removeAny with one pair
    @Test
    public final void testRemoveAnyOnePair() {
        Map<String, String> test = this.createFromArgsTest("1", "i");

        Map<String, String> ref = this.createFromArgsRef("1", "i");

        Map.Pair<String, String> element = test.removeAny();

        assertEquals(test.hasKey(element.key()), false);
        assertEquals(ref.hasKey(element.key()), true);
        assertEquals(test.size(), ref.size() - 1);
    }

    // Testing removeAny with multiple pairs
    @Test
    public final void testRemoveAny() {
        Map<String, String> test = this.createFromArgsTest("1", "i", "2", "ii",
                "3", "iii", "4", "iiij", "5", "v", "6", "vi", "7", "vii", "8",
                "viij", "9", "ix", "10", "x");
        Map<String, String> ref = this.createFromArgsRef("1", "i", "2", "ii",
                "3", "iii", "4", "iiij", "5", "v", "6", "vi", "7", "vii", "8",
                "viij", "9", "ix", "10", "x");

        Map.Pair<String, String> element = test.removeAny();

        // Check that pair is no longer in test, but is still in ref
        assertEquals(test.hasKey(element.key()), false);
        // Make sure removeAny properly returns element by referencing ref Map
        assertEquals(ref.hasKey(element.key()), true);
        assertEquals(ref.value(element.key()), element.value());
        // Make sure size is updated
        assertEquals(test.size(), ref.size() - 1);
    }

    // Testing size with empty map
    @Test
    public final void testSizeZero() {
        Map<String, String> test = this.constructorTest();
        assertEquals(0, test.size());
    }

    // Testing size with a non-empty map
    @Test
    public final void testSizeNonZero() {
        Map<String, String> test = this.createFromArgsTest("1", "i", "2", "ii",
                "3", "iii", "4", "iiij", "5", "v", "6", "vi", "7", "vii", "8",
                "viij", "9", "ix", "10", "x");

        final int ten = 10;

        assertEquals(ten, test.size());
    }

    // Testing hasKey when map contains key
    @Test
    public final void testHasKeyTrue() {
        Map<String, String> test = this.createFromArgsTest("1", "i", "2", "ii",
                "3", "iii", "4", "iiij", "5", "v", "6", "vi", "7", "vii", "8",
                "viij", "9", "ix", "10", "x");
        Map<String, String> ref = this.createFromArgsRef("1", "i", "2", "ii",
                "3", "iii", "4", "iiij", "5", "v", "6", "vi", "7", "vii", "8",
                "viij", "9", "ix", "10", "x");

        assertEquals(test.hasKey("1"), true);
        assertEquals(test.hasKey("5"), true);

        // Make sure that hasKey does not change map
        assertEquals(test, ref);
    }

    // Testing hasKey when map does not contain key
    @Test
    public final void testHasKeyFalse() {
        Map<String, String> test = this.createFromArgsTest("1", "i", "2", "ii",
                "3", "iii", "4", "iiij", "5", "v", "6", "vi", "7", "vii", "8",
                "viij", "9", "ix", "10", "x");
        Map<String, String> ref = this.createFromArgsRef("1", "i", "2", "ii",
                "3", "iii", "4", "iiij", "5", "v", "6", "vi", "7", "vii", "8",
                "viij", "9", "ix", "10", "x");

        assertEquals(test.hasKey("i"), false);
        assertEquals(test.hasKey("11"), false);

        // Make sure that hasKey does not change map
        assertEquals(test, ref);
    }

    // Routine cases for value
    @Test
    public final void testValue() {
        Map<String, String> test = this.createFromArgsTest("1", "i", "2", "ii",
                "3", "iii", "4", "iiij", "5", "v", "6", "vi", "7", "vii", "8",
                "viij", "9", "ix", "10", "x");
        Map<String, String> ref = this.createFromArgsRef("1", "i", "2", "ii",
                "3", "iii", "4", "iiij", "5", "v", "6", "vi", "7", "vii", "8",
                "viij", "9", "ix", "10", "x");

        assertEquals("vi", test.value("6"));
        assertEquals("viij", test.value("8"));

        // Make sure that value does not change map
        assertEquals(test, ref);
    }

}

```