Problem 2: Write Tests for Graph

I used black box testing by typing the tests before the code is written. This way I do not bias my tests by knowing my code. Instead this allows for stronger specifications and force me to write code that adheres to the test. Therefore the heuristic that applied in this homework is the black box heuristic.

```
package hw4;

import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.Comparator;


public final class GraphTest {
    private Graph empty = new Graph();
    private Graph graph1 = new Graph();
    private Graph graph2 = new Graph();
    private Graph graph3 = new Graph();

    private void eqNodes(Graph g, String rep) {
        ArrayList<String> temp = g.getNodes();
        Collections.sort(temp);
        String ans = "";
        for (int i = 0; i < temp.size()-1; i++)
            ans = ans + temp.get(i) + ", ";
        if (temp.size() != 0)
            ans = ans + temp.get(temp.size()-1);

        assertEquals(ans, rep);
    }

    private void eqEdges(Graph g, String rep) {
        ArrayList<String[]> temp = g.getEdges();
        temp.sort(Comparator.comparing(a -> a[1]));
        temp.sort(Comparator.comparing(a -> a[0]));
        String ans = "";
        for (int i = 0; i < temp.size()-1; i++)
            ans = ans + String.join("->", temp.get(i))  + ", ";

        if (temp.size() != 0)
            ans = ans + String.join("->", temp.get(temp.size()-1));
        assertEquals(ans, rep);
    }

    private void eqEdgeWeights(Graph g, String rep) {
        ArrayList<String[]> temp = g.getEdges();
        ArrayList<String> ansList = new ArrayList<String>();
        String ans;
        for (int i = 0; i < temp.size(); i++)
            ansList.addAll(g.getEdgeWeight(temp.get(i)[0], temp.get(i)[1]));
        Collections.sort(ansList);
        ans = String.join(", ", ansList);
        assertEquals(ans, rep);
    }
```

```
private void eqAdjList(Graph g, String node, String rep) {
    ArrayList<String> temp = g.getAdjacencyList(node);
    Collections.sort(temp);
    String ans = "";
    for (int i = 0; i < temp.size()-1; i++)
        ans = ans + temp.get(i) + ", ";
    if (temp.size() != 0)
        ans = ans + temp.get(temp.size()-1);
    assertEquals(ans, rep);
}

//////////////
// Constructor
//////////////
@Test
public void testConstructor() {
    new Graph();
}

//////////
// addNode
//////////
@Test
public void testAddNode() {
    graph1.addNode("a");
    graph2.addNode("A");
    graph3.addNode("1203758");
    eqNodes(graph1, "a");
    eqNodes(graph2, "A");
    eqNodes(graph3, "1203758");
}
@Test
public void testAddMultipleNodes() {
    graph1.addNode("a");
    graph1.addNode("A");
    graph1.addNode("1203758");
    graph2.addNode("!!(@&^*");
    graph2.addNode("aAlakjshdf");
    graph2.addNode("78148763");
    graph2.addNode("Hello World");
    eqNodes(graph1, "1203758, A, a");
    eqNodes(graph2, "!!(@&^*, 78148763, Hello World, aAlakjshdf");
}
```

```
@Test
public void testAddDuplicateNodes() {
    graph1.addNode("a");
    graph1.addNode("a");
    graph1.addNode("a");
    graph1.addNode("1");
    graph1.addNode("2");
    graph1.addNode("3");
    graph1.addNode("1");
    graph1.addNode("2");
    graph1.addNode("3");
    graph2.addNode("!!(@&^*");
    graph2.addNode("aAlakjshdf");
    graph2.addNode("78148763");
    graph2.addNode("Hello World");
    graph2.addNode("!!(@&^*");
    graph2.addNode("Hello World");
    graph2.addNode("Hello World");
    eqNodes(graph1, "1, 2, 3, a");
    eqNodes(graph2, "!!(@&^*, 78148763, Hello World, aAlakjshdf");
}

//////////////
// removeNode
//////////////
@Test
public void testRemoveNode() {
    graph1.addNode("a");
    graph1.removeNode("a");
    graph1.addNode("a");
    graph1.removeNode("a");
    graph2.addNode("a");
    graph2.removeNode("a");
    graph2.addNode("b");
    graph2.addNode("c");
    graph2.removeNode("c");
    graph3.addNode("a");
    graph3.addNode("b");
    graph3.addNode("c");
    graph3.removeNode("c");
    graph3.removeNode("a");
    graph3.removeNode("b");
    eqNodes(graph1, "");
    eqNodes(graph2, "b");
    eqNodes(graph3, "");
}
```

```
@Test
public void testRemoveNonExistentNode() {
    graph1.addNode("a");
    graph1.removeNode("a");
    graph1.removeNode("b");
    graph2.removeNode("c");
    graph3.addNode("1");
    graph3.addNode("2");
    graph3.addNode("3");
    graph3.addNode("4");
    graph3.addNode("5");
    graph3.addNode("6");
    graph3.addNode("7");
    graph3.removeNode("8");
    eqNodes(graph1, "");
    eqNodes(graph2, "");
    eqNodes(graph3, "1, 2, 3, 4, 5, 6, 7");
}

@Test
public void testRemoveEdgeRelatedNodes() {
    graph1.addNode("a");
    graph1.removeNode("a");
    graph1.addNode("b");
    graph1.addNode("c");
    graph1.addEdge("b", "c", "1");
    graph1.addEdge("b", "c", "2");
    graph1.addEdge("c", "b", "3");
    eqNodes(graph1, "b, c");
    eqEdges(graph1, "b->c, c->b");
    eqEdgeWeights(graph1, "1, 2, 3");
    graph1.removeNode("b");
    eqNodes(graph1, "c");
    eqEdges(graph1, "");
    eqEdgeWeights(graph1, "");
}

//////////
// addEdge
//////////
@Test
public void testAddEdge() {
    graph1.addNode("a");
    graph1.addNode("b");
    graph1.addNode("c");
    graph1.addEdge("a", "b", "1");
    graph1.addEdge("a", "c", "2");
    graph1.addEdge("b", "a", "5");
    graph1.addEdge("b", "c", "6");
    graph1.addEdge("c", "a", "3");
    graph1.addEdge("c", "b", "4");
    eqNodes(graph1, "a, b, c");
    eqEdges(graph1, "a->b, a->c, b->a, b->c, c->a, c->b");
    eqEdgeWeights(graph1, "1, 2, 3, 4, 5, 6");
}
```

```
@Test
public void testAddSameEdge() {
    graph1.addNode("a");
    graph1.addNode("b");
    graph1.addNode("c");
    graph1.addEdge("a", "b", "1");
    graph1.addEdge("a", "b", "2");
    graph1.addEdge("a", "b", "3");
    graph1.addEdge("a", "b", "1");
    graph1.addEdge("a", "b", "2");
    graph1.addEdge("a", "b", "3");
    eqNodes(graph1, "a, b, c");
    eqEdges(graph1, "a->b");
    eqEdgeWeights(graph1, "1, 2, 3");
}

@Test
public void testAddEdgeWithNonexistingNode() {
    graph1.addNode("a");
    graph1.addNode("b");
    graph1.addNode("c");
    graph1.addEdge("a", "e", "1");
    graph1.addEdge("a", "f", "2");
    graph1.addEdge("a", "b", "3");
    graph1.addEdge("a", "g", "1");
    graph1.addEdge("a", "h", "2");
    graph1.addEdge("a", "i", "3");
    graph2.addNode("a");
    graph2.addEdge("a", "f", "2");
    eqNodes(graph1, "a, b, c");
    eqEdges(graph1, "a->b");
    eqEdgeWeights(graph1, "3");
    eqNodes(graph2, "a");
    eqEdges(graph2, "");
    eqEdgeWeights(graph2, "");
}
```

```
//////////////
// remove edge
//////////////
@Test
public void testRemoveEdge() {
    graph1.addNode("a");
    graph1.addNode("b");
    graph1.addNode("c");
    graph1.addEdge("a", "b", "1");
    graph1.addEdge("a", "c", "2");
    graph1.addEdge("b", "a", "5");
    graph1.addEdge("b", "c", "6");
    graph1.addEdge("c", "a", "3");
    graph1.addEdge("c", "b", "4");
    graph1.removeEdge("a", "b");
    graph1.removeEdge("a", "c");
    graph1.removeEdge("b", "a");
    graph1.removeEdge("c", "a");
    graph2.addNode("a");
    graph2.addNode("b");
    graph2.addNode("c");
    graph2.addEdge("a", "b", "1");
    graph2.addEdge("a", "c", "2");
    graph2.addEdge("b", "a", "5");
    graph2.addEdge("b", "c", "6");
    graph2.addEdge("c", "a", "3");
    graph2.addEdge("c", "b", "4");
    graph2.removeEdge("a", "b");
    graph2.removeEdge("a", "c");
    graph2.removeEdge("b", "a");
    graph2.removeEdge("b", "c");
    graph2.removeEdge("c", "a");
    graph2.removeEdge("c", "b");
    eqNodes(graph1, "a, b, c");
    eqEdges(graph1, "b->c, c->b");
    eqEdgeWeights(graph1, "4, 6");
    eqNodes(graph2, "a, b, c");
    eqEdges(graph2, "");
    eqEdgeWeights(graph2, "");
}

@Test
public void testRemoveMultipleWeightedEdge() {
    graph1.addNode("a");
    graph1.addNode("b");
    graph1.addNode("c");
    graph1.addEdge("a", "b", "1");
    graph1.addEdge("a", "b", "2");
    graph1.addEdge("a", "b", "3");
    graph1.removeEdge("a", "b");
    eqNodes(graph1, "a, b, c");
    eqEdges(graph1, "");
    eqEdgeWeights(graph1, "");
}
```

```
@Test
public void testRemoveNonexistentEdge() {
    graph1.addNode("a");
    graph1.addNode("b");
    graph1.addNode("c");
    graph1.addEdge("a", "b", "1");
    graph1.addEdge("a", "b", "2");
    graph1.addEdge("a", "b", "3");
    graph1.removeEdge("a", "c");
    graph1.removeEdge("b", "c");
    graph1.removeEdge("b", "a");
    eqNodes(graph1, "a, b, c");
    eqEdges(graph1, "a->b");
    eqEdgeWeights(graph1, "1, 2, 3");
}

@Test
public void testRemoveEdgeWithNonexistentNode() {
    graph1.addNode("a");
    graph1.addNode("b");
    graph1.addNode("c");
    graph1.addEdge("a", "b", "1");
    graph1.addEdge("a", "b", "2");
    graph1.addEdge("a", "b", "3");
    graph1.removeEdge("a", "e");
    graph1.removeEdge("c", "f");
    graph1.removeEdge("b", "g");
    graph1.removeEdge("i", "h");
    eqNodes(graph1, "a, b, c");
    eqEdges(graph1, "a->b");
    eqEdgeWeights(graph1, "1, 2, 3");
}
```

```
////////////////////
// remove edge weight
////////////////////
@Test
public void testRemoveEdgeWeight() {
    graph1.addNode("a");
    graph1.addNode("b");
    graph1.addNode("c");
    graph1.addEdge("a", "b", "1");
    graph1.addEdge("a", "c", "2");
    graph1.addEdge("b", "a", "5");
    graph1.addEdge("b", "c", "6");
    graph1.addEdge("c", "a", "3");
    graph1.addEdge("c", "b", "4");
    graph1.removeEdgeWeight("a", "b", "1");
    graph1.removeEdgeWeight("a", "c", "2");
    graph1.removeEdgeWeight("b", "a", "5");
    graph1.removeEdgeWeight("c", "a", "3");
    graph2.addNode("a");
    graph2.addNode("b");
    graph2.addNode("c");
    graph2.addEdge("a", "b", "1");
    graph2.addEdge("a", "c", "2");
    graph2.addEdge("b", "a", "5");
    graph2.addEdge("b", "c", "6");
    graph2.addEdge("c", "a", "3");
    graph2.addEdge("c", "b", "4");
    graph2.removeEdgeWeight("a", "b", "1");
    graph2.removeEdgeWeight("a", "c", "2");
    graph2.removeEdgeWeight("b", "a", "5");
    graph2.removeEdgeWeight("b", "c", "6");
    graph2.removeEdgeWeight("c", "a", "3");
    graph2.removeEdgeWeight("c", "b", "4");
    graph3.addNode("a");
    graph3.addNode("b");
    graph3.addNode("c");
    graph3.addEdge("a", "b", "1");
    graph3.addEdge("a", "b", "2");
    graph3.addEdge("a", "b", "3");
    graph3.addEdge("a", "c", "4");
    graph3.addEdge("a", "c", "5");
    graph3.addEdge("a", "c", "6");
    graph3.removeEdgeWeight("a", "b", "1");
    graph3.removeEdgeWeight("a", "c", "5");
    graph3.removeEdgeWeight("a", "c", "6");
    eqNodes(graph1, "a, b, c");
    eqEdges(graph1, "b->c, c->b");
    eqEdgeWeights(graph1, "4, 6");
    eqNodes(graph2, "a, b, c");
    eqEdges(graph2, "");
    eqEdgeWeights(graph2, "");
    eqNodes(graph3, "a, b, c");
    eqEdges(graph3, "a->b, a->c");
    eqEdgeWeights(graph3, "2, 3, 4");
}
```

```
@Test
public void testRemoveNonexistentEdgeWeight() {
    graph1.addNode("a");
    graph1.addNode("b");
    graph1.addNode("c");
    graph1.addEdge("a", "b", "1");
    graph1.addEdge("a", "b", "2");
    graph1.addEdge("a", "b", "3");
    graph1.removeEdgeWeight("a", "c", "1");
    graph1.removeEdgeWeight("b", "c", "1");
    graph1.removeEdgeWeight("b", "a", "1");
    graph1.removeEdgeWeight("a", "b", "4");
    graph1.removeEdgeWeight("a", "b", "5");
    eqNodes(graph1, "a, b, c");
    eqEdges(graph1, "a->b");
    eqEdgeWeights(graph1, "1, 2, 3");
}

@Test
public void testRemoveEdgeWeightWithNonexistentNode() {
    graph1.addNode("a");
    graph1.addNode("b");
    graph1.addNode("c");
    graph1.addEdge("a", "b", "1");
    graph1.addEdge("a", "b", "2");
    graph1.addEdge("a", "b", "3");
    graph1.removeEdgeWeight("a", "e", "1");
    graph1.removeEdgeWeight("c", "f", "1");
    graph1.removeEdgeWeight("b", "g", "3");
    graph1.removeEdgeWeight("i", "h", "3");
    eqNodes(graph1, "a, b, c");
    eqEdges(graph1, "a->b");
    eqEdgeWeights(graph1, "1, 2, 3");
}
```

```
    ////////////
    // getAdjList
    ////////////
    @Test
    public void testAdjList() {
        graph1.addNode("a");
        graph1.addNode("b");
        graph1.addNode("c");
        graph1.addEdge("a", "b", "1");
        graph1.addEdge("a", "b", "2");
        eqNodes(graph1, "a, b, c");
        eqEdges(graph1, "a->b");
        eqEdgeWeights(graph1, "1, 2");
        eqAdjList(graph1, "a", "b");
        eqAdjList(graph1, "b", "");
        graph1.addEdge("b", "a", "1");
        graph1.addEdge("b", "a", "2");
        graph1.removeEdge("b", "a");
        eqNodes(graph1, "a, b, c");
        eqEdges(graph1, "a->b");
        eqEdgeWeights(graph1, "1, 2");
        eqAdjList(graph1, "a", "b");
        eqAdjList(graph1, "b", "");
        graph1.addEdge("a", "c", "4");
        graph1.addEdge("a", "c", "3");
        eqNodes(graph1, "a, b, c");
        eqEdges(graph1, "a->b, a->c");
        eqEdgeWeights(graph1, "1, 2, 3, 4");
        eqAdjList(graph1, "a", "b, c");
        eqAdjList(graph1, "b", "");
        eqAdjList(graph1, "c", "");
    }
}
```

The Tests cases I added include adding and removing nonexistent edges, nodes, and edgeweights. It is important to consider the user that accidentally adds an edge without establishing the node first.