***IMPORTANT NOTE:*** *To test both versions of the graph using the same tests, you need to perform the following steps: you must comment the lines of code that initialize the graph in the setups using lists as data structure, and in turn, the corresponding lines that initialize the graph using arrays must be uncommented.*

**Setup Stages.**

|  |  |  |
| --- | --- | --- |
| **Name** | **Class** | **Stage** |
| setupStageSimpleGraph | GraphTest | An object of the IGraph interface (Can be initialized with lists or matrices) of simple type with eight vertices:  Vertex1: key = 1, element = “A”  Vertex2: key = 2, element = “B”  Vertex3: key = 3, element = “C”  Vertex4: key = 4, element = “D”  Vertex5: key = 5, element = “E”  Vertex6: key = 6, element = “F”  Vertex7: key = 7, element = “G”  Vertex8: key = 10, element = “I” |
| setupStagePseudoGraph | GraphTest | An object of the IGraph interface (Can be initialized with lists or matrices) of pseudograph type with five vertices:  Vertex1: key = “1”, element = 1  Vertex2: key = “2”, element = 2  Vertex3: key = “3”, element = 3  Vertex4: key = “4”, element = 4  Vertex5: key = “5”, element = 5  And nine edges:  Edge1: keyVertex1 = “1”, keyVertex2 = “2”, weight = 1  Edge2: keyVertex1 = “1”, keyVertex2 = “1”, weight = 1  Edge3: keyVertex1 = “1”, keyVertex2 = “3”, weight = 1  Edge4: keyVertex1 = “2”, keyVertex2 = “4”, weight = 1  Edge5: keyVertex1 = “5”, keyVertex2 = “1”, weight = 1  Edge6: keyVertex1 = “5”, keyVertex2 = “2”, weight = 1  Edge7: keyVertex1 = “5”, keyVertex2 = “3”, weight = 1  Edge8: keyVertex1 = “5”, keyVertex2 = “4”, weight = 1  Edge9: keyVertex1 = “5”, keyVertex2 = “5”, weight = 1 |
| setupStageMultiGraph | GraphTest | An object of the IGraph interface (Can be initialized with lists or matrices) of multigraph type with six vertices:  Vertex1: key = 1, element = “1”  Vertex2: key = 2, element = “2”  Vertex3: key = 3, element = “3”  Vertex4: key = 4, element = “4”  Vertex5: key = 5, element = “5”  Vertex6: key = 6, element = “6”  And ten edges:  Edge1: keyVertex1 = 1, keyVertex2 = “2”, weight = 1  Edge2: keyVertex1 = 2, keyVertex2 = 3, weight = 1  Edge3: keyVertex1 = 3, keyVertex2 = 4, weight = 1  Edge4: keyVertex1 = 4, keyVertex2 = 5, weight = 1  Edge5: keyVertex1 = 5, keyVertex2 = 1, weight = 1  Edge6: keyVertex1 = 1, keyVertex2 = 6, weight = 1  Edge7: keyVertex1 = 2, keyVertex2 = 6, weight = 1  Edge8: keyVertex1 = 3, keyVertex2 = 6, weight = 1  Edge9: keyVertex1 = 4, keyVertex2 = 6, weight = 1  Edge10: keyVertex1 = 5, keyVertex2 = 6, weight = 1 |
| setupStageDirectedGraph | GraphTest | An object of the IGraph interface (Can be initialized with lists or matrices) of directed type with eleven vertices:  Vertex1: key = 1, element = “1”  Vertex2: key = 2, element = “2”  Vertex3: key = 3, element = “3”  Vertex4: key = 4, element = “4”  Vertex5: key = 5, element = “5”  Vertex6: key = 6, element = “6”  Vertex7: key = 7, element = “7”  Vertex8: key = 8, element = “8”  Vertex9: key = 9, element = “9”  Vertex10: key = 10, element = “10”  Vertex11: key = 11, element = “11”  And twenty-five edges:  Edge1: keyVertex1 = 1, keyVertex2 = 2, weight = 1  Edge2: keyVertex1 = 2, keyVertex2 = 1, weight = 1  Edge3: keyVertex1 = 1, keyVertex2 = 3, weight = 1  Edge4: keyVertex1 = 3, keyVertex2 = 1, weight = 1  Edge5: keyVertex1 = 2, keyVertex2 = 4, weight = 1  Edge6: keyVertex1 = 9, keyVertex2 = 7, weight = 1  Edge7: keyVertex1 = 9 keyVertex2 = 11, weight = 1  Edge8: keyVertex1 = 6, keyVertex2 = 10, weight = 1  Edge9: keyVertex1 = 11, keyVertex2 = 7, weight = 1  Edge10: keyVertex1 = 8, keyVertex2 = 6, weight = 1  Edge11: keyVertex1 = 10, keyVertex2 = 11, weight = 1  Edge12: keyVertex1 = 5, keyVertex2 = 1, weight = 1  Edge13: keyVertex1 = 5, keyVertex2 = 6, weight = 1  Edge14: keyVertex1 = 5, keyVertex2 = 3, weight = 1  Edge15: keyVertex1 = 9, keyVertex2 = 4, weight = 1  Edge16: keyVertex1 = 5, keyVertex2 = 7, weight = 1  Edge17: keyVertex1 = 3, keyVertex2 = 6, weight = 1  Edge18: keyVertex1 = 4, keyVertex2 = 7, weight = 1  Edge19: keyVertex1 = 6, keyVertex2 = 7, weight = 1  Edge20: keyVertex1 = 8, keyVertex2 = 10, weight = 1  Edge21: keyVertex1 = 8, keyVertex2 = 11, weight = 1  Edge22: keyVertex1 = 5, keyVertex2 = 2, weight = 1  Edge23: keyVertex1 = 5, keyVertex2 = 4, weight = 1  Edge24: keyVertex1 = 2, keyVertex2 = 9, weight = 1  Edge25: keyVertex1 = 8, keyVertex2 = 7, weight = 1 |
| setupStageMultigraphDirected | GraphTest | An object of the IGraph interface (Can be initialized with lists or matrices) of Multigraph Directed type without vertices. |
| setupStage6 | GraphTest | An object of the IGraph interface (Can be initialized with lists or matrices) of simple type with nine vertices:  Vertex1: key = “r”, element = “r”  Vertex2: key = “s”, element = “s”  Vertex3: key = “t”, element = “t”  Vertex4: key = “u”, element = “u”  Vertex5: key = “v”, element = “v”  Vertex6: key = “w”, element = “w”  Vertex7: key = “x”, element = “x”  Vertex8: key = “y”, element = “y”  Vertex9: key = “z”, element = “z” |
| setupStage7 | GraphTest | An object of the IGraph interface (Can be initialized with lists or matrices) of simple type with six vertices:  Vertex1: key = “a”, element = “a”  Vertex2: key = “b”, element = “b”  Vertex3: key = “c”, element = “c”  Vertex4: key = “d”, element = “d”  Vertex5: key = “e”, element = “e”  Vertex6: key = “z”, element = “z” |
| setupStage8 | GraphTest | An object of the IGraph interface (Can be initialized with lists or matrices) of simple type with four vertices:  Vertex1: key = 1, element = “a”  Vertex2: key = 2, element = “b”  Vertex3: key = 3, element = “c”  Vertex4: key = 4, element = “d” |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test objective:** Verify that the insert vertex method of the Graph class works correctly by successfully inserting Strings. | | | | | |
| **Class** | **Method** | **Stage** | **Input values** | **Expected result** |
| Graph | insertVertex | setupStageMultiGraphDirected | key = 1  element = “A” | The vertex “A” has been successfully added to the Graph and when trying to get the element via key = 1, it must be equal to "A". |
| Graph | insertVertex | setupStageSimpleGraph | key = 1  element = “AAaa” | The vertex is not added since there is already one with the same key and when trying to get the element via key = 1, it must be equal to "A". |
| Graph | insertVertex | setupStageMultiGraphDirected | Vertex1 = (key = 1, element = “A”)  Vertex2 = (key = 2, element = “B”)  Vertex3 = (key = 3, element = “C”)  Vertex4 = (key = 4, element = “D”)  Vertex5 = (key = 5, element = “E”) | Vertices are added successfully, and when trying to access elements via keys 1 - 5, the elements should equal “A” – “E”, respectively. |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test objective:** Verify that the delete vertex method of the Graph class works correctly. | | | | | |
| **Class** | **Method** | **Stage** | **Input values** | **Expected result** |
| Graph | deleteVertex | setupStageSimpleGraph | key = 1  key = 2 | The vertices are successfully removed and when trying to access the vertices via their key, it should be equal to null. |
| Graph | deleteVertex | setupStageDirectedGraph | key = 7 | The vertex with key = 7 has been removed from the graph and the getVertex() method call with key = 7 returns null. Verify that when trying to check if vertices 4 and 7 are adjacent, an exception of type GraphException is thrown. This indicates that the edge between vertices 4 and 7 has also been removed from the graph. A similar pattern is followed, and they check if the edges between the remaining vertices and the deleted vertex are still present in the graph. |
| Graph | deleteVertex | setupStageMultiGraph | key =6 | Verify that the vertex with key = 6 has been removed from the multiple graph and that the getVertex() method call with key = 6 returns null. Verify that a GraphException is thrown when trying to check if vertices 1 and 6 are adjacent. This indicates that the edges between vertices 1 and 6 have also been removed from the multiple graph. A similar pattern is followed, and they check if the edges between the remaining vertices and the deleted vertex are still present in the graph. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test objective:** Verify that the insert edge method of the Graph class works correctly. | | | | |
| **Class** | **Method** | **Stage** | **Input values** | **Expected result** |
| Graph | insertEdge | setupStageSimpleGraph | Edge1=(key1=1, key2=2, weight=1)  Edge2=(key1=2, key2=3, weight=1)  Edge3=(key1=3,key2=4, weight=1)  Edge4=(key1=4, key2=5, weight=1)  Edge5=(key1=5, key2=1, weight=1) | The adjacency lists of the graph are verified.  First assertion verifies that vertex 1 is adjacent to vertex 2.  Similar assertions check for adjacency between consecutive vertices in the loop: 2-3, 3-4, 4-5, and 5-1. |
| Graph | insertEdge | setupStageMultiGraphDirected | Vertex1 = (key=1, element = ”A”)  Edge1=(key1=1, key2=1, weight=1) | If an exception is thrown, the test fails with the message "The multi directed graph accepts multiple edges and loops."  If no exception is thrown, the edge between vertex "1" and itself is verified to be present in the graph using the adjacent method. If it is present, the test is considered successful. |
| Graph | insertEdge | setupStageSimpleGraph | Edge1= (key1=1, key2=2, weight=1) | When trying to insert the same edge between the same vertices again, an exception of type GraphException must be thrown |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test objective:** Verify that the delete edge method of the Graph class works correctly. | | | | |
| **Class** | **Method** | **Stage** | **Input values** | **Expected result** |
| Graph | deleteEdge | setupStagePseudoGraph | Key1=”1”, key2=”1”, weight=1  Key1=”1”, key2=”5”, weight=1 | To verify that the deleted edges are no longer present in the graph, it is verified that there is no adjacency between the vertices "1" and "1", "1" and "5", and "5" and "1". |
| Graph | deleteEdge | setupStageDirectedGraph | Key1=1, key2=2, weight=1  Key1=1, key2=3, weight=1 | To verify that the deleted edges are no longer present in the graph, it is verified that there is no adjacency between vertices 1 and 2, and between vertices 1 and 3.  To verify that the existing adjacencies before removing the edges are still present in the graph, it is verified that there is an adjacency between vertices 2 and 1, and between vertices 3 and 1. |
| Graph | deleteEdge | setupStagePseudoGraph | Key1=”1”, key2=”9”, weight=1 | Verifies that an exception of type Graph Exception is thrown when trying to remove a non-existent edge. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test objective:** Verify that the adjacent method of the Graph class works correctly. | | | | |
| **Class** | **Method** | **Stage** | **Input values** | **Expected result** |
| Graph | adjacent | setupStageDirectedGraph | None | Verify that the given vertices are adjacent in the graph. The pairs of vertices that are verified are (4, 7), (5, 7), (6, 7), (8, 7), (9, 7), and (11, 7). |
| Graph | adjacent | setupStageDirectedGraph | None | Verify that the given vertices are not adjacent in the graph. The pairs of vertices that are verified are (7, 4), (7, 5), (7, 6), (7, 8), (7, 9), and (7, 11). |
| Graph | adjacent | setupStageDirectedGraph | None | Verifies that a Graph Exception is thrown when attempting to verify adjacency between vertices 1 and 18, and between vertices 18 and 1. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test objective:** Verify that the BFS method of the Graph class works correctly. | | | | |
| **Class** | **Method** | **Stage** | **Input values** | **Expected result** |
| Graph | BFS | setupStage6 | Edge1= (key1=” v”, key2=”r”, weight=1)  Edge2= (key1=” r”, key2=” s”, weight=1)  Edge3= (key1=” s”, key2=” w”, weight=1)  Edge4= (key1=” w”, key2=”t”, weight=1)  Edge5= (key1=” w”, key2=”x”, weight=1)  Edge6= (key1=”t”, key2=”x”, weight=1)  Edge7= (key1=”t”, key2=” u”, weight=1)  Edge8= (key1=”x”, key2=” u”, weight=1)  Edge9= (key1=” u”, key2=” y”, weight=1)  Edge10= (key1=”x”, key2=” y”, weight=1) | From the source vertex "s" to the other vertices following a path in width (by levels) in the graph:  The vertex distance "s" must be 0. The vertex distance "r" must be 1. The vertex distance "w" must be 1. The vertex distance "v" must be 2. The vertex distance "t" must be 2. The vertex distance "x" must be 2. The vertex distance "u" must be 3. The distance of the vertex "y" must be 3. |
| Graph | BFS | setupStageSimpleGraph | Edge1 = (key1=1, key2=10, weight =1)  Edge2 = (key1=10, key2=3, weight=1)  Edge3 = (key1=10, key2=2, weight=1)  Edge4 = (key1=3, key2=5, weight=1)  Edge5 = (key1=5, key2=2, weight=1)  Edge6 = (key1=6, key2=4, weight=1)  Edge7 = (key1=4, key2=7, weight=1) | From the source vertex 10 to the other vertices following a path in width (by levels) in the graph:  The distance from vertex 1 must be 1. The distance from vertex 2 must be 1. The distance from vertex 3 must be 1. The distance from vertex 5 must be 2. The distance from vertex 6 must be 2. The distance from vertex 4 must be 3. The distance from vertex 7 must be 4. |
| Graph | BFS | setupStageSimpleGraph | Edge1= (key1=1, key2=10, weight=1)  Edge2= (key1=1, key2=5, weight=1)  Edge3 = (key1 =1, key2=3, weight=1) | Since the graph does not have a direct connection from vertex 1 to vertex 7, the distance of vertex 7 is expected to be Integer.MAX\_VALUE after running the BFS algorithm. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test objective:** Verify that the dijkstra method of the Graph class works correctly. | | | | |
| **Class** | **Method** | **Stage** | **Input values** | **Expected result** |
| Graph | dijkstra | None | Edge1 = (key1 = “a”, key2 = “b”, weight = 4)  Edge2 = (key1 = “a”, key2 = “d”, weight = 2)  Edge3 = (key1 = “d”, key2 = “e”, weight = 3)  Edge4 = (key1 = “b”, key2 = “c”, weight = 3)  Edge5 = (key1 = “b”, key2 = “e”, weight = 3)  Edge6 = (key1 = “c”, key2 = “z”, weight = 2)  Edge7 = (key1 = “e”, key2 = “z”, weight = 1)  KeyVertexSource = “a” | After running Dijkstra's algorithm on the graph, the shortest distances from vertex "a" to all other vertices are expected to be as follows:  Distance from "a" to "a": 0  Distance from "a" to "b": 4  Distance from "a" to "c": 7  Distance from "a" to "d": 2  Distance from "a" to "e": 5  Distance from "a" to "z": 6  These distances represent the shortest paths from the source vertex "a" to all other vertices in the graph, considering the weight of the edges. |
| Graph | dijkstra | setupStageSimpleGraph | KeyVertexSource = 1 | After running Dijkstra's algorithm on the graph, the shortest distances from the source vertex to all other vertices are expected to be as follows:  Distance from the source vertex to itself: 0  Distance from source vertex to all other vertices: Integer.MAX\_VALUE  (This indicates that there is no direct path from the source vertex to the other vertices of the graph) |
| Graph | dijkstra | setupStagePseudoGraph | KeyVertexSource = “3” | After running Dijkstra's algorithm on the graph, the shortest distances from the source vertex to all other vertices are expected to be as follows: Distance from the source vertex to itself: 0 Distance from the source vertex to all other vertices: [1, 2, 0, 2, 1] (These are the minimum distances from the source vertex "3" to the other vertices of the graph) |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test objective:** Verify that the kruskal method of the Graph class works correctly. | | | | |
| **Class** | **Method** | **Stage** | **Input values** | **Expected result** |
| Graph | kruskal | setupStage7 | Edge1 = (key1 = 1, key2 = 2, weight = 10)  Edge2 = (key1 = 1, key2 = 3, weight = 7)  Edge3 = (key1 = 2, key2 = 3, weight = 5)  Edge4 = (key1 = 2, key2 = 4, weight = 3) | After running Kruskal's algorithm on the graph, the shortest path through all vertices of the graph is found, being the following:  path = {(b, d, 3), (b, c, 5), (a, c, 7)} |
| Graph | kruskal | setupStageSimpleGraph | None | Since none of the vertices are connected, an empty list is expected to be returned. |
| Graph | kruskal | SetupStageDirectedGraph | None | Since the network used for the test is directed, the method is expected to throw an exception of type GraphException |