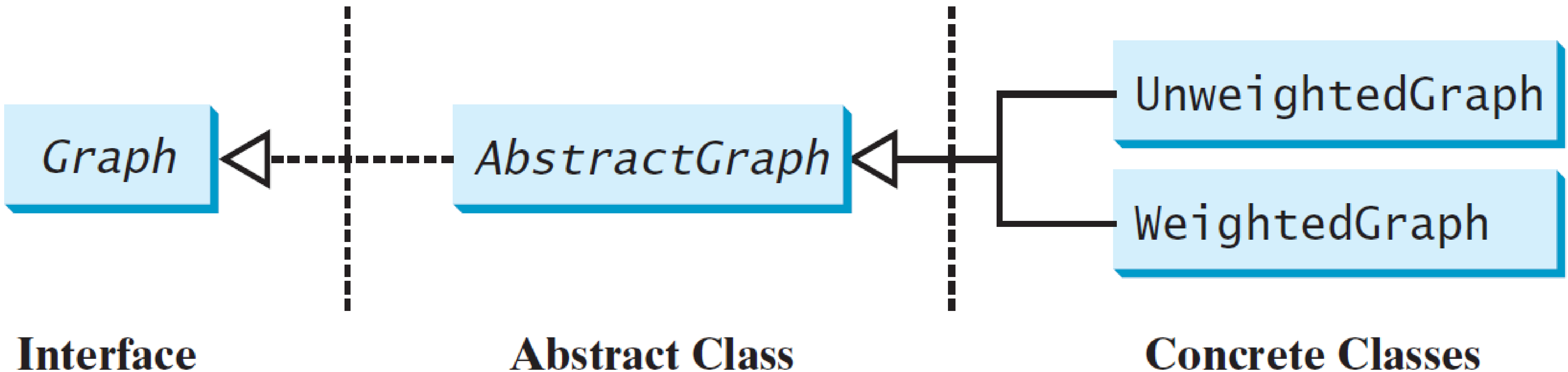
Graph Modeling

Graphs can be modeled using interfaces, abstract classes, and concrete classes.



Graph Interface

«interface»
Graph<V>

The generic type V is the type for vertices.

+getSize(): int	Returns the number of vertices in the graph.
+getVertices(): List<V>	Returns the vertices in the graph.
+getVertex(index: int): V	Returns the vertex object for the specified vertex index.
+getIndex(v: V): int	Returns the index for the specified vertex.
+getNeighbors(index: int): List<Integer>	Returns the neighbors of vertex with the specified index.
+getDegree(index: int): int	Returns the degree for a specified vertex index.
+printEdges(): void	Prints the edges.
+clear(): void	Clears the graph.
+addVertex(v, V): boolean	Returns true if v is added to the graph. Returns false if v is already in the graph.
+addEdge(u: int, v: int): boolean	Adds an edge from u to v to the graph throws `IllegalArgumentException` if u or v is invalid. Returns true if the edge is added and false if (u, v) is already in the graph.
+dfs(v: int): AbstractGraph<V>.Tree	Obtains a depth-first search tree starting from v.
+bfs(v: int): AbstractGraph<V>.Tree	Obtains a breadth-first search tree starting from v.

AbstractGraph

```
AbstractGraph<V>
-----------------------------------------------------------
#vertices: List<V>
#neighbors: List<List<Edge>>
-----------------------------------------------------------
#AbstractGraph()
#AbstractGraph(vertices: V[], edges:
   int[][])
#AbstractGraph(vertices: List<V>, edges:
   List<Edge>)
#AbstractGraph(edges: int[][],
   numberOfVertices: int)
#AbstractGraph(edges: List<Edge>,
   numberOfVertices: int)
+addEdge(e: Edge): boolean
Inner classes Tree is defined here
```

Vertices in the graph.
Neighbors for each vertex in the graph.

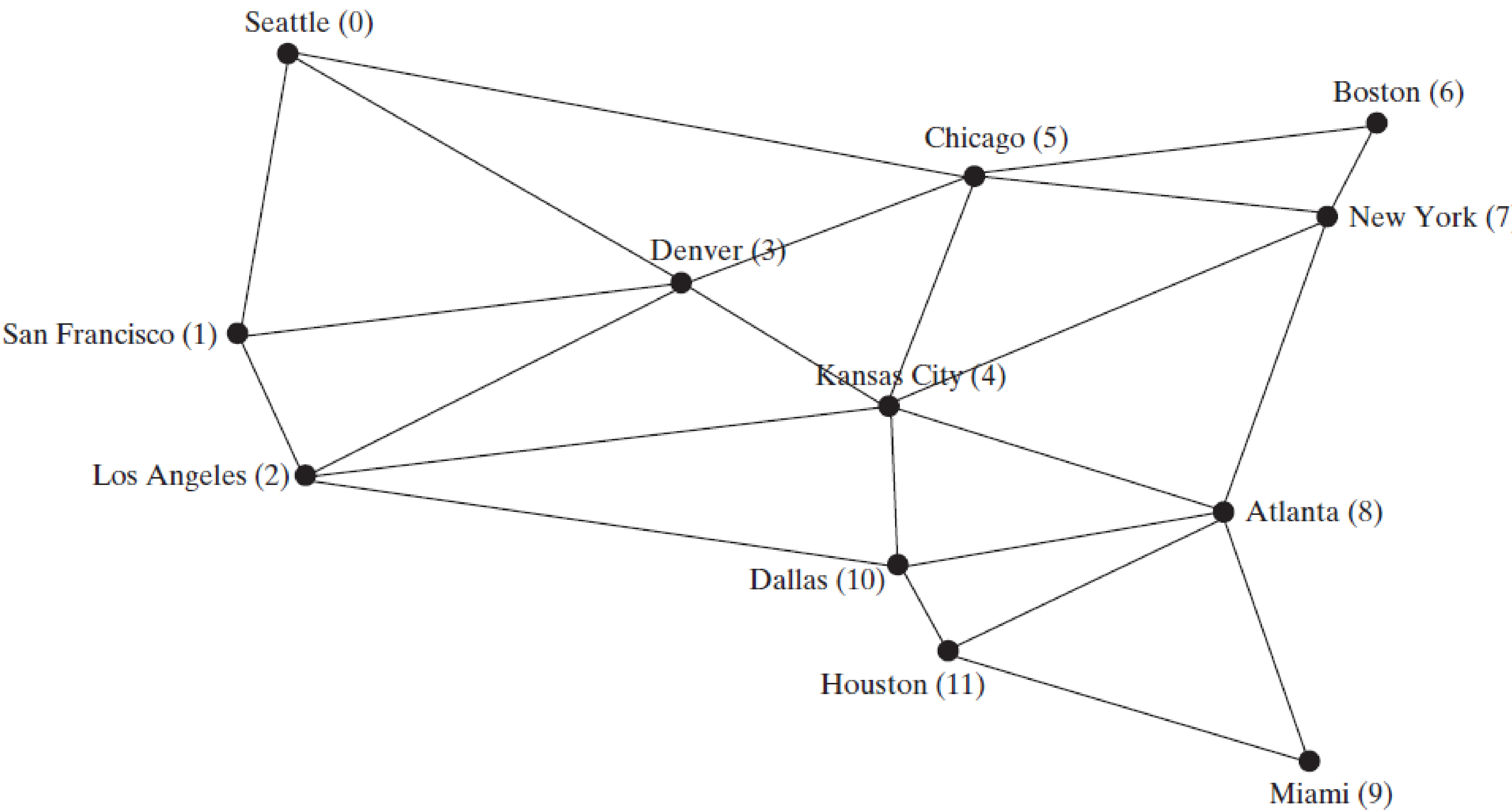
Constructs an empty graph.
Constructs a graph with the specified edges and vertices stored in arrays.
Constructs a graph with the specified edges and vertices stored in lists.
Constructs a graph with the specified edges in an array and the integer vertices 1, 2, ...
Constructs a graph with the specified edges in a list and the integer vertices 1, 2, ….

Adds an edge into the adjacency edge list.

UnweightedGraph Class

UnweightedGraph<V>	
+UnweightedGraph()	Constructs an empty unweighted graph.
+UnweightedGraph(vertices: V[], edges: int[][])	Constructs a graph with the specified edges and vertices in arrays.
+UnweightedGraph(vertices: List<V>, edges: List<Edge>)	Constructs a graph with the specified edges and vertices stored in lists.
+UnweightedGraph(edges: List<Edge>, numberOfVertices: int)	Constructs a graph with the specified edges in an array and the integer vertices 1, 2, ...
+UnweightedGraph(edges: int[][], numberOfVertices: int)	Constructs a graph with the specified edges in a list and the integer vertices 1, 2, ...

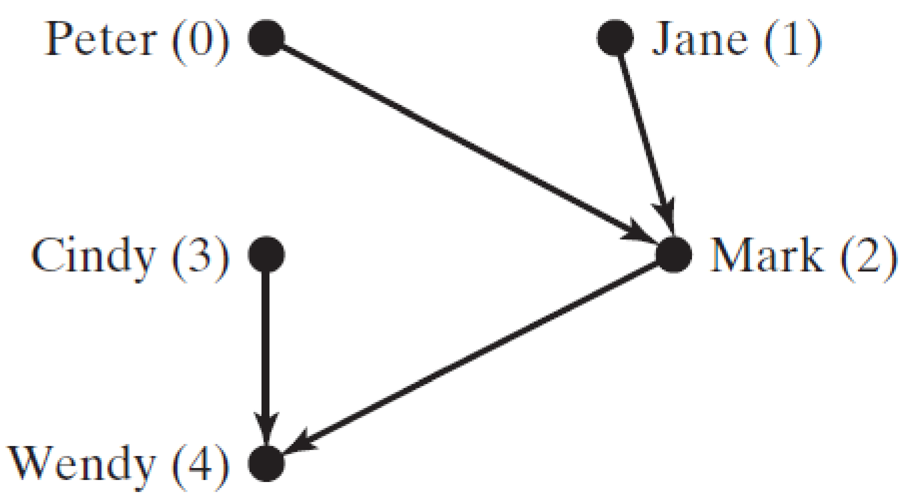
Sample Graph



Sample Test File

```java
public class TestGraph {
  public static void main(String[] args) {
    String[] vertices = {"Seattle", "San Francisco", "Los Angeles",
      "Denver", "Kansas City", "Chicago", "Boston", "New York",
      "Atlanta", "Miami", "Dallas", "Houston"};

    // Edge array for graph
    int[][] edges = {
      {0, 1}, {0, 3}, {0, 5},
      {1, 0}, {1, 2}, {1, 3},
      {2, 1}, {2, 3}, {2, 4}, {2, 10},
      {3, 0}, {3, 1}, {3, 2}, {3, 4}, {3, 5},
      {4, 2}, {4, 3}, {4, 5}, {4, 7}, {4, 8}, {4, 10},
      {5, 0}, {5, 3}, {5, 4}, {5, 6}, {5, 7},
      {6, 5}, {6, 7},
      {7, 4}, {7, 5}, {7, 6}, {7, 8},
      {8, 4}, {8, 7}, {8, 9}, {8, 10}, {8, 11},
      {9, 8}, {9, 11},
      {10, 2}, {10, 4}, {10, 8}, {10, 11},
      {11, 8}, {11, 9}, {11, 10}
    };

```

```java

    Graph<String> graph1 = new UnweightedGraph<>(vertices, edges);
    System.out.println("The number of vertices in graph1: "
      + graph1.getSize());
    System.out.println("The vertex with index 1 is "
      + graph1.getVertex(1));
    System.out.println("The index for Miami is " +
      graph1.getIndex("Miami"));
    System.out.println("The edges for graph1:");
    graph1.printEdges();

    // List of Edge objects for graph
    String[] names = {"Peter", "Jane", "Mark", "Cindy", "Wendy"};
```



```java
    java.util.ArrayList<AbstractGraph.Edge> edgeList
      = new java.util.ArrayList<>();
    edgeList.add(new AbstractGraph.Edge(0, 2));
    edgeList.add(new AbstractGraph.Edge(1, 2));
    edgeList.add(new AbstractGraph.Edge(2, 4));
    edgeList.add(new AbstractGraph.Edge(3, 4));
    // Create a graph with 5 vertices
    Graph<String> graph2 = new UnweightedGraph<>
      (java.util.Arrays.asList(names), edgeList);
    System.out.println("\nThe number of vertices in graph2: "
      + graph2.getSize());
    System.out.println("The edges for graph2:");
    graph2.printEdges();
  }
}
```

Output

```
The number of vertices in graph1: 12
The vertex with index 1 is San Francisco
The index for Miami is 9
The edges for graph1:
Seattle (0): (0, 1) (0, 3) (0, 5)
San Francisco (1): (1, 0) (1, 2) (1, 3)
Los Angeles (2): (2, 1) (2, 3) (2, 4) (2, 10)
Denver (3): (3, 0) (3, 1) (3, 2) (3, 4) (3, 5)
Kansas City (4): (4, 2) (4, 3) (4, 5) (4, 7) (4, 8) (4, 10)
Chicago (5): (5, 0) (5, 3) (5, 4) (5, 6) (5, 7)
Boston (6): (6, 5) (6, 7)
New York (7): (7, 4) (7, 5) (7, 6) (7, 8)
Atlanta (8): (8, 4) (8, 7) (8, 9) (8, 10) (8, 11)
Miami (9): (9, 8) (9, 11)
Dallas (10): (10, 2) (10, 4) (10, 8) (10, 11)
Houston (11): (11, 8) (11, 9) (11, 10)
```

```
The number of vertices in graph2: 5
The edges for graph2:
Peter (0): (0, 2)
Jane (1): (1, 2)
Mark (2): (2, 4)
Cindy (3): (3, 4)
Wendy (4):
```

Graph class

```
public interface Graph<V> {
  /** Return the number of vertices in the graph */
  public int getSize();

  /** Return the vertices in the graph */
  public java.util.List<V> getVertices();

  /** Return the object for the specified vertex index */
  public V getVertex(int index);

  /** Return the index for the specified vertex object */
  public int getIndex(V v);

  /** Return the neighbors of vertex with the specified index */
  public java.util.List<Integer> getNeighbors(int index);

  /** Return the degree for a specified vertex */
  public int getDegree(int v);

```

```java

  /** Print the edges */
  public void printEdges();

  /** Clear the graph */
  public void clear();

  /** Add a vertex to the graph */
  public void addVertex(V vertex);

  /** Add an edge to the graph */
  public void addEdge(int u, int v);

  /** Obtain a depth-first search tree starting from v */
  public AbstractGraph<V>.Tree dfs(int v);

  /** Obtain a breadth-first search tree starting from v */
  public AbstractGraph<V>.Tree bfs(int v);
}
```

AbstractGraph Class

```
import java.util.*;

public abstract class AbstractGraph<V> implements Graph<V> {
  protected List<V> vertices = new ArrayList<>(); // Store vertices
  protected List<List<Edge>> neighbors
    = new ArrayList<>(); // Adjacency lists

  /** Construct an empty graph */
  protected AbstractGraph() {
  }

  /** Construct a graph from vertices and edges stored in arrays */
  protected AbstractGraph(V[] vertices, int[][] edges) {
     for (int i = 0; i < vertices.length; i++)
       addVertex(vertices[i]);

     createAdjacencyLists(edges, vertices.length);
   }

```

```java

  /** Construct a graph from vertices and edges stored in List */
  protected AbstractGraph(List<V> vertices, List<Edge> edges) {
    for (int i = 0; i < vertices.size(); i++)
      addVertex(vertices.get(i));

    createAdjacencyLists(edges, vertices.size());
  }

  /** Construct a graph for integer vertices 0, 1, 2 and edge list */
  protected AbstractGraph(List<Edge> edges, int numberOfVertices) {
    for (int i = 0; i < numberOfVertices; i++)
      addVertex((V)(new Integer(i))); // vertices is {0, 1, ...}

    createAdjacencyLists(edges, numberOfVertices);
  }

```

```java

  /** Construct a graph from integer vertices 0, 1, and edge array */
  protected AbstractGraph(int[][] edges, int numberOfVertices) {
    for (int i = 0; i < numberOfVertices; i++)
      addVertex((V)(new Integer(i))); // vertices is {0, 1, ...}

    createAdjacencyLists(edges, numberOfVertices);
  }

  /** Create adjacency lists for each vertex */
  private void createAdjacencyLists(
      int[][] edges, int numberOfVertices) {
    for (int i = 0; i < edges.length; i++) {
      addEdge(edges[i][0], edges[i][1]);
    }
  }

```

```java

  /** Create adjacency lists for each vertex */
  private void createAdjacencyLists(
      List<Edge> edges, int numberOfVertices) {
    for (Edge edge: edges) {
      addEdge(edge.u, edge.v);
    }
  }

  @Override /** Return the number of vertices in the graph */
  public int getSize() {
    return vertices.size();
  }

  @Override /** Return the vertices in the graph */
  public List<V> getVertices() {
    return vertices;
  }
```

```java

  @Override /** Return the object for the specified vertex */
  public V getVertex(int index) {
    return vertices.get(index);
  }

  @Override /** Return the index for the specified vertex object */
  public int getIndex(V v) {
    return vertices.indexOf(v);
  }

  @Override /** Return the neighbors of the specified vertex */
  public List<Integer> getNeighbors(int index) {
    List<Integer> result = new ArrayList<>();
    for (Edge e: neighbors.get(index))
      result.add(e.v);

    return result;
  }

```

```java

  @Override /** Return the degree for a specified vertex */
  public int getDegree(int v) {
    return neighbors.get(v).size();
  }

  @Override /** Print the edges */
  public void printEdges() {
    for (int u = 0; u < neighbors.size(); u++) {
      System.out.print(getVertex(u) + " (" + u + "): ");
      for (Edge e: neighbors.get(u)) {
        System.out.print("(" + getVertex(e.u) + ", " +
          getVertex(e.v) + ") ");
      }
      System.out.println();
    }
  }

```

```java

  @Override /** Clear the graph */
  public void clear() {
    vertices.clear();
    neighbors.clear();
  }

  @Override /** Add a vertex to the graph */
  public boolean addVertex(V vertex) {
    if (!vertices.contains(vertex)) {
      vertices.add(vertex);
      neighbors.add(new ArrayList<Edge>());
      return true;
    }
    else {
      return false;
    }
  }
```

```java

  /** Add an edge to the graph */
  protected boolean addEdge(Edge e) {
    if (e.u < 0 || e.u > getSize() - 1)
      throw new IllegalArgumentException("No such index: " + e.u);

    if (e.v < 0 || e.v > getSize() - 1)
      throw new IllegalArgumentException("No such index: " + e.v);

    if (!neighbors.get(e.u).contains(e)) {
      neighbors.get(e.u).add(e);
      return true;
    }
    else {
      return false;
    }
  }

```

```java

  @Override /** Add an edge to the graph */
  public boolean addEdge(int u, int v) {
    return addEdge(new Edge(u, v));
  }

  /** Edge inner class inside the AbstractGraph class */
  public static class Edge {
    public int u; // Starting vertex of the edge
    public int v; // Ending vertex of the edge

    /** Construct an edge for (u, v) */
    public Edge(int u, int v) {
      this.u = u;
      this.v = v;
    }

    public boolean equals(Object o) {
      return u == ((Edge)o).u && v == ((Edge)o).v;
    }
  }
```

```java
  @Override /** Obtain a DFS tree starting from vertex v */

  public Tree dfs(int v) {
    List<Integer> searchOrder = new ArrayList<>();
    int[] parent = new int[vertices.size()];
    for (int i = 0; i < parent.length; i++)
      parent[i] = -1; // Initialize parent[i] to -1

    // Mark visited vertices
    boolean[] isVisited = new boolean[vertices.size()];

    // Recursively search
    dfs(v, parent, searchOrder, isVisited);

    // Return a search tree
    return new Tree(v, parent, searchOrder);
  }
```

```java

  /** Recursive method for DFS search */
  private void dfs(int u, int[] parent, List<Integer> searchOrder,
      boolean[] isVisited) {
    // Store the visited vertex
    searchOrder.add(u);
    isVisited[u] = true; // Vertex v visited

    for (Edge e : neighbors.get(u)) {
      if (!isVisited[e.v]) {
        parent[e.v] = u; // The parent of vertex e.v is u
        dfs(e.v, parent, searchOrder, isVisited); // Recursive search
      }
    }
  }
```

```java
@Override /** Starting bfs search from vertex v */

public Tree bfs(int v) {
  List<Integer> searchOrder = new ArrayList<>();
  int[] parent = new int[vertices.size()];
  for (int i = 0; i < parent.length; i++)
    parent[i] = -1; // Initialize parent[i] to -1

  java.util.LinkedList<Integer> queue =
    new java.util.LinkedList<>(); // list used as a queue
  boolean[] isVisited = new boolean[vertices.size()];
  queue.offer(v); // Enqueue v
  isVisited[v] = true; // Mark it visited

  while (!queue.isEmpty()) {
    int u = queue.poll(); // Dequeue to u
    searchOrder.add(u); // u searched
    for (Edge e: neighbors.get(u)) {
      if (!isVisited[e.v]) {
        queue.offer(e.v); // Enqueue w
        parent[e.v] = u; // The parent of w is u
        isVisited[e.v] = true; // Mark it visited
      }
    }
  }
    return new Tree(v, parent, searchOrder);
}
```

```java

  /** Tree inner class inside the AbstractGraph class */

  public class Tree {
    private int root; // The root of the tree
    private int[] parent; // Store the parent of each vertex
    private List<Integer> searchOrder; // Store the search order

    /** Construct a tree with root, parent, and searchOrder */
    public Tree(int root, int[] parent, List<Integer> searchOrder) {
      this.root = root;
      this.parent = parent;
      this.searchOrder = searchOrder;
    }

```

```java

  /** Return the root of the tree */
  public int getRoot() {
    return root;
  }

  /** Return the parent of vertex v */
  public int getParent(int v) {
    return parent[v];
  }

  /** Return an array representing search order */
  public List<Integer> getSearchOrder() {
    return searchOrder;
  }

```

```
    /** Return number of vertices found */
    public int getNumberOfVerticesFound() {
      return searchOrder.size();
    }

    /** Return the path of vertices from a vertex to the root */
    public List<V> getPath(int index) {
      ArrayList<V> path = new ArrayList<>();

      do {
        path.add(vertices.get(index));
        index = parent[index];
      }
      while (index != -1);

      return path;
    }

```

```java
  /** Print a path from the root to vertex v */
  public void printPath(int index) {
    List<V> path = getPath(index);
    System.out.print("A path from " + vertices.get(root) + " to " +
      vertices.get(index) + ": ");
    for (int i = path.size() - 1; i >= 0; i--)
      System.out.print(path.get(i) + " ");
  }
```

```java

    /** Print the whole tree */
    public void printTree() {
      System.out.println("Root is: " + vertices.get(root));
      System.out.print("Edges: ");
      for (int i = 0; i < parent.length; i++) {
        if (parent[i] != -1) {
          // Display an edge
          System.out.print("(" + vertices.get(parent[i]) + ", " +
            vertices.get(i) + ") ");
        }
      }
      System.out.println();
    }
  }
}
```

UnweightedGraph class

```
import java.util.*;

public class UnweightedGraph<V> extends AbstractGraph<V> {
  /** Construct an empty graph */
  public UnweightedGraph() {
  }

  /** Construct a graph from vertices and edges stored in arrays */
  public UnweightedGraph(V[] vertices, int[][] edges) {
    super(vertices, edges);
  }

  /** Construct a graph from vertices and edges stored in List */
  public UnweightedGraph(List<V> vertices, List<Edge> edges) {
    super(vertices, edges);
  }

```

```java
  public UnweightedGraph(List<Edge> edges, int numberOfVertices) {
    super(edges, numberOfVertices);
  }

  /** Construct a graph from integer vertices 0, 1, and edge array */
  public UnweightedGraph(int[][] edges, int numberOfVertices) {
    super(edges, numberOfVertices);
  }
}
```

WeightedGraph class

```java
import java.util.*;

public class WeightedGraph<V> extends AbstractGraph<V> {
  /** Construct an empty */
  public WeightedGraph() {
  }

  /** Construct a WeightedGraph from vertices and edged in arrays */
  public WeightedGraph(V[] vertices, int[][] edges) {
    createWeightedGraph(java.util.Arrays.asList(vertices), edges);
  }

  /** Construct a WeightedGraph from vertices and edges in list */
  public WeightedGraph(int[][] edges, int numberOfVertices) {
    List<V> vertices = new ArrayList<>();
    for (int i = 0; i < numberOfVertices; i++)
      vertices.add((V)(new Integer(i)));

    createWeightedGraph(vertices, edges);
  }

```

```java
  /** Construct a WeightedGraph for vertices 0, 1, 2 and edge list */
  public WeightedGraph(List<V> vertices, List<WeightedEdge> edges) {
    createWeightedGraph(vertices, edges);
  }

  /** Construct a WeightedGraph from vertices 0, 1, and edge array */
  public WeightedGraph(List<WeightedEdge> edges,
      int numberOfVertices) {
    List<V> vertices = new ArrayList<>();
    for (int i = 0; i < numberOfVertices; i++)
      vertices.add((V)(new Integer(i)));

    createWeightedGraph(vertices, edges);
  }
```

```java

  /** Create adjacency lists from edge arrays */
  private void createWeightedGraph(List<V> vertices, int[][] edges) {
    this.vertices = vertices;

    for (int i = 0; i < vertices.size(); i++) {
      neighbors.add(new ArrayList<Edge>()); // Create a list for vertices
    }

    for (int i = 0; i < edges.length; i++) {
      neighbors.get(edges[i][0]).add(
        new WeightedEdge(edges[i][0], edges[i][1], edges[i][2]));
    }
  }

  /** Create adjacency lists from edge lists */
  private void createWeightedGraph(
      List<V> vertices, List<WeightedEdge> edges) {
    this.vertices = vertices;

    for (int i = 0; i < vertices.size(); i++) {
      neighbors.add(new ArrayList<Edge>()); // Create a list for vertices
    }

    for (WeightedEdge edge: edges) {
      neighbors.get(edge.u).add(edge); // Add an edge into the list
    }
  }

```

```java

  /** Return the weight on the edge (u, v) */
  public double getWeight(int u, int v) throws Exception {
    for (Edge edge : neighbors.get(u)) {
      if (edge.v == v) {
        return ((WeightedEdge)edge).weight;
      }
    }

    throw new Exception("Edge does not exit");
  }

  /** Display edges with weights */
  public void printWeightedEdges() {
    for (int i = 0; i < getSize(); i++) {
      System.out.print(getVertex(i) + " (" + i + "): ");
      for (Edge edge : neighbors.get(i)) {
        System.out.print("(" + edge.u +
          ", " + edge.v + ", " + ((WeightedEdge)edge).weight + ") ");
      }
      System.out.println();
    }
  }

  /** Add edges to the weighted graph */
  public boolean addEdge(int u, int v, double weight) {
    return addEdge(new WeightedEdge(u, v, weight));
  }

```

```java
  /** Get a minimum spanning tree rooted at vertex 0 */
  public MST getMinimumSpanningTree() {
    return getMinimumSpanningTree(0);
  }

  /** Get a minimum spanning tree rooted at a specified vertex */
  public MST getMinimumSpanningTree(int startingVertex) {
    // cost[v] stores the cost by adding v to the tree
    double[] cost = new double[getSize()];
    for (int i = 0; i < cost.length; i++) {
      cost[i] = Double.POSITIVE_INFINITY; // Initial cost
    }
    cost[startingVertex] = 0; // Cost of source is 0

    int[] parent = new int[getSize()]; // Parent of a vertex
    parent[startingVertex] = -1; // startingVertex is the root
    double totalWeight = 0; // Total weight of the tree thus far

    List<Integer> T = new ArrayList<>();

    // Expand T
    while (T.size() < getSize()) {
      // Find smallest cost v in V - T
      int u = -1; // Vertex to be determined
      double currentMinCost = Double.POSITIVE_INFINITY;
      for (int i = 0; i < getSize(); i++) {
        if (!T.contains(i) && cost[i] < currentMinCost) {
          currentMinCost = cost[i];
          u = i;
        }
      }
```

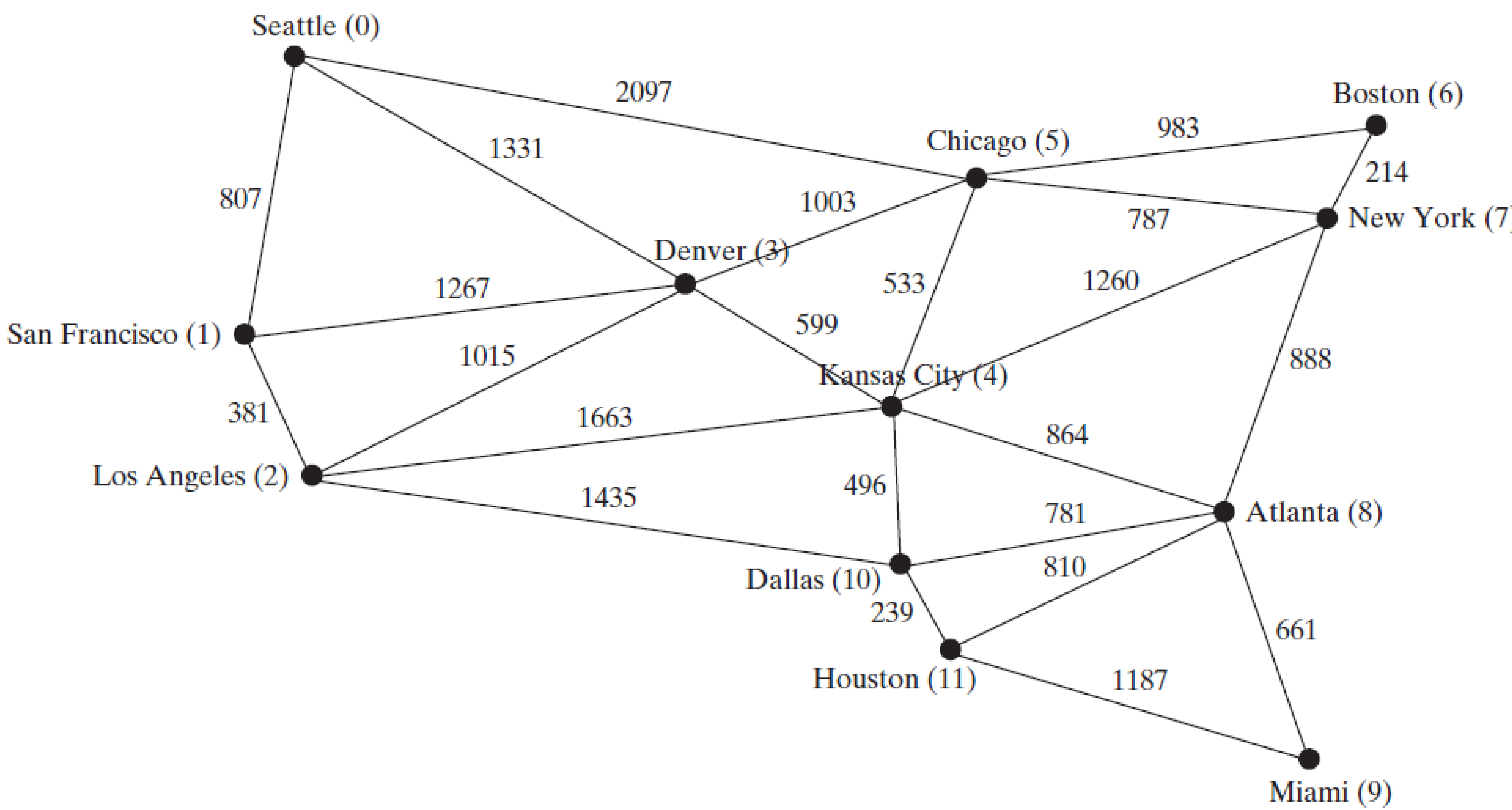
```java

      T.add(u); // Add a new vertex to T
      totalWeight += cost[u]; // Add cost[u] to the tree

      // Adjust cost[v] for v that is adjacent to u and v in V - T
      for (Edge e : neighbors.get(u)) {
        if (!T.contains(e.v) && cost[e.v] > ((WeightedEdge)e).weight) {
          cost[e.v] = ((WeightedEdge)e).weight;
          parent[e.v] = u;
        }
      }
    } // End of while

    return new MST(startingVertex, parent, T, totalWeight);
  }

  /** MST is an inner class in WeightedGraph */
  public class MST extends Tree {
    private double totalWeight; // Total weight of all edges in the tree

    public MST(int root, int[] parent, List<Integer> searchOrder,
        double totalWeight) {
      super(root, parent, searchOrder);
      this.totalWeight = totalWeight;
    }

    public double getTotalWeight() {
      return totalWeight;
    }
  }

```

```java

  /** Find single source shortest paths */
  public ShortestPathTree getShortestPath(int sourceVertex) {
    // cost[v] stores the cost of the path from v to the source
    double[] cost = newdouble[getSize()];
    for (int i = 0; i < cost.length; i++) {
      cost[i] = Double.POSITIVE_INFINITY; // Initial cost set to infinity
    }
    cost[sourceVertex] = 0; // Cost of source is 0

    // parent[v] stores the previous vertex of v in the path
    int[] parent = newint[getSize()];
    parent[sourceVertex] = -1; // The parent of source is set to -1

    // T stores the vertices whose path found so far
    List<Integer> T = new ArrayList<>();

```

```
      // Expand T
      while (T.size() < getSize()) {
        // Find smallest cost v in V - T
        int u = -1; // Vertex to be determined
        double currentMinCost = Double.POSITIVE_INFINITY;
        for (int i = 0; i < getSize(); i++) {
          if (!T.contains(i) && cost[i] < currentMinCost) {
            currentMinCost = cost[i];
            u = i;
          }
        }

        T.add(u); // Add a new vertex to T

        // Adjust cost[v] for v that is adjacent to u and v in V - T
        for (Edge e : neighbors.get(u)) {
          if (!T.contains(e.v)
              && cost[e.v] > cost[u] + ((WeightedEdge)e).weight) {
            cost[e.v] = cost[u] + ((WeightedEdge)e).weight;
            parent[e.v] = u;
          }
        }
      } // End of while

      // Create a ShortestPathTree
      return new ShortestPathTree(sourceVertex, parent, T, cost);
    }
```

```java

  /** ShortestPathTree is an inner class in WeightedGraph */
  public class ShortestPathTree extends Tree {
    private double[] cost; // cost[v] is the cost from v to source

    /** Construct a path */
    public ShortestPathTree(int source, int[] parent,
        List<Integer> searchOrder, double[] cost) {
      super(source, parent, searchOrder);
      this.cost = cost;
    }

    /** Return the cost for a path from the root to vertex v */
    public double getCost(int v) {
      return cost[v];
    }

    /** Print paths from all vertices to the source */
    public void printAllPaths() {
      System.out.println("All shortest paths from " +
        vertices.get(getRoot()) + " are:");
      for (int i = 0; i < cost.length; i++) {
        printPath(i); // Print a path from i to the source
        System.out.println("(cost: " + cost[i] + ")"); // Path cost
      }
    }
  }
}
```

Sample Graph



TestWeightedGraph.java

```java
public class TestWeightedGraph {
  public static void main(String[] args) {
    String[] vertices = {"Seattle", "San Francisco", "Los Angeles",
      "Denver", "Kansas City", "Chicago", "Boston", "New York",
      "Atlanta", "Miami", "Dallas", "Houston"};

    int[][] edges = {
      {0, 1, 807}, {0, 3, 1331}, {0, 5, 2097},
      {1, 0, 807}, {1, 2, 381}, {1, 3, 1267},
      {2, 1, 381}, {2, 3, 1015}, {2, 4, 1663}, {2, 10, 1435},
      {3, 0, 1331}, {3, 1, 1267}, {3, 2, 1015}, {3, 4, 599},
        {3, 5, 1003},
      {4, 2, 1663}, {4, 3, 599}, {4, 5, 533}, {4, 7, 1260},
        {4, 8, 864}, {4, 10, 496},
      {5, 0, 2097}, {5, 3, 1003}, {5, 4, 533},
        {5, 6, 983}, {5, 7, 787},
      {6, 5, 983}, {6, 7, 214},
      {7, 4, 1260}, {7, 5, 787}, {7, 6, 214}, {7, 8, 888},
      {8, 4, 864}, {8, 7, 888}, {8, 9, 661},
        {8, 10, 781}, {8, 11, 810},
      {9, 8, 661}, {9, 11, 1187},
      {10, 2, 1435}, {10, 4, 496}, {10, 8, 781}, {10, 11, 239},
      {11, 8, 810}, {11, 9, 1187}, {11, 10, 239}
    };

```

```java

    WeightedGraph<String> graph1 =
      new WeightedGraph<>(vertices, edges);
    System.out.println("The number of vertices in graph1: "
      + graph1.getSize());
    System.out.println("The vertex with index 1 is "
      + graph1.getVertex(1));
    System.out.println("The index for Miami is " +
      graph1.getIndex("Miami"));
    System.out.println("The edges for graph1:");
    graph1.printWeightedEdges();

    edges = new int[][] {
      {0, 1, 2}, {0, 3, 8},
      {1, 0, 2}, {1, 2, 7}, {1, 3, 3},
      {2, 1, 7}, {2, 3, 4}, {2, 4, 5},
      {3, 0, 8}, {3, 1, 3}, {3, 2, 4}, {3, 4, 6},
      {4, 2, 5}, {4, 3, 6}
    };
    WeightedGraph<Integer> graph2 = new WeightedGraph<>(edges, 5);
    System.out.println("\nThe edges for graph2:");
    graph2.printWeightedEdges();
  }
}
```

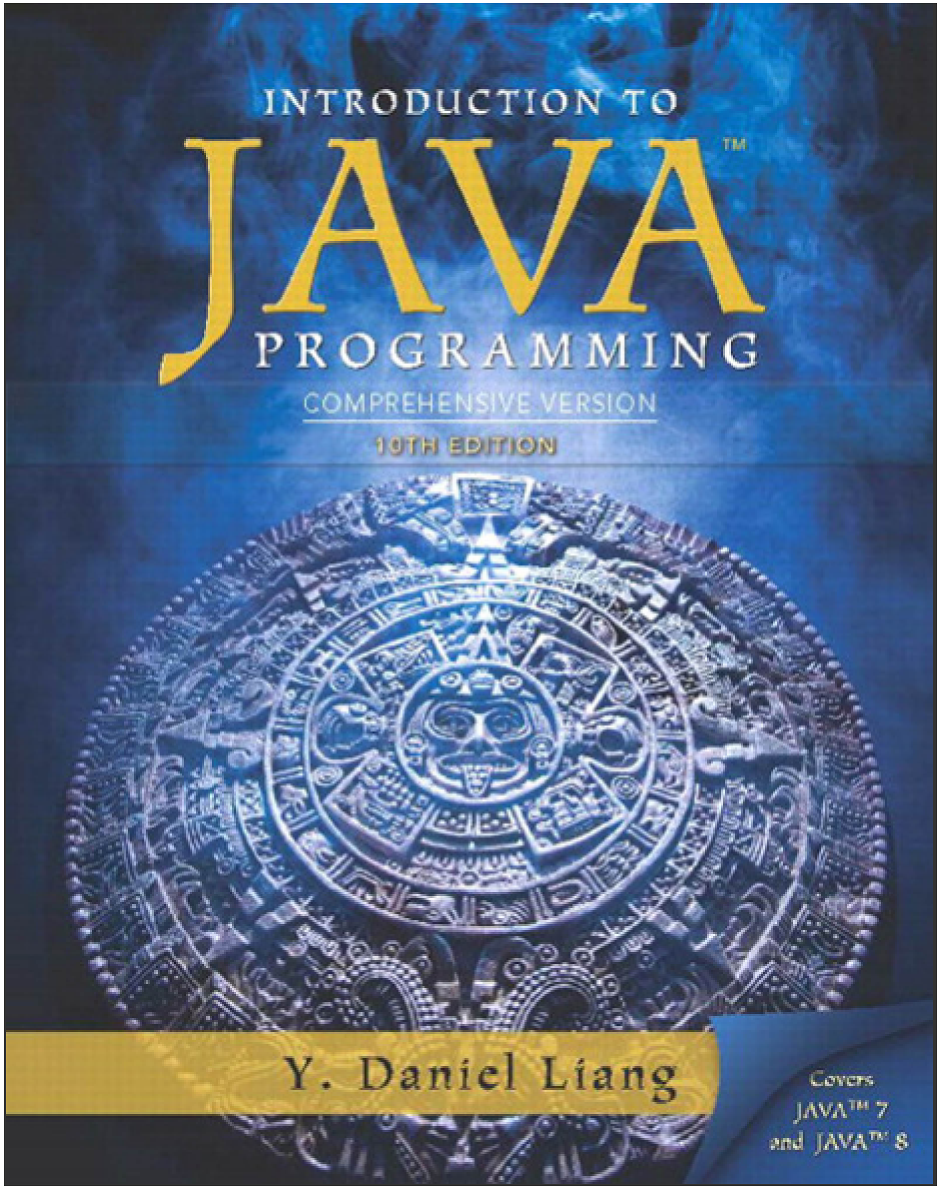
Output

```
The number of vertices in graph1: 12
The vertex with index 1 is San Francisco
The index for Miami is 9
The edges for graph1:
Vertex 0: (0, 1, 807) (0, 3, 1331) (0, 5, 2097)
Vertex 1: (1, 2, 381) (1, 0, 807) (1, 3, 1267)
Vertex 2: (2, 1, 381) (2, 3, 1015) (2, 4, 1663) (2, 10, 1435)
Vertex 3: (3, 4, 599) (3, 5, 1003) (3, 1, 1267)
  (3, 0, 1331) (3, 2, 1015)
Vertex 4: (4, 10, 496) (4, 8, 864) (4, 5, 533) (4, 2, 1663)
  (4, 7, 1260) (4, 3, 599)
Vertex 5: (5, 4, 533) (5, 7, 787) (5, 3, 1003)
  (5, 0, 2097) (5, 6, 983)
Vertex 6: (6, 7, 214) (6, 5, 983)
Vertex 7: (7, 6, 214) (7, 8, 888) (7, 5, 787) (7, 4, 1260)
Vertex 8: (8, 9, 661) (8, 10, 781) (8, 4, 864)
  (8, 7, 888) (8, 11, 810)
Vertex 9: (9, 8, 661) (9, 11, 1187)
Vertex 10: (10, 11, 239) (10, 4, 496) (10, 8, 781) (10, 2, 1435)
Vertex 11: (11, 10, 239) (11, 9, 1187) (11, 8, 810)
```

Output

```
The edges for graph2:
Vertex 0: (0, 1, 2) (0, 3, 8)
Vertex 1: (1, 0, 2) (1, 2, 7) (1, 3, 3)
Vertex 2: (2, 3, 4) (2, 1, 7) (2, 4, 5)
Vertex 3: (3, 1, 3) (3, 4, 6) (3, 2, 4) (3, 0, 8)
Vertex 4: (4, 2, 5) (4, 3, 6)
```

Reference



INTRODUCTION TO
JAVA™
PROGRAMMING
COMPREHENSIVE VERSION
10TH EDITION
Y. Daniel Liang
Covers
JAVA™ 7
and JAVA™ 8