Exercise Set 6: Graph Algorithms

Exercise 6.1 The goal of this exercise is to learn to represent a graph as a data structure.
The graph structure is defined as follows.

- A `Graph` class is a hashtable of `Node`s. Use the `data Object` as the *key* - see below.
- A `Node` is a structure consisting of `Object data` and `Edges edgelist`.
- `Edges` is a list (or other collection) of `Edge`s.
- `Edge` is a structure consisting of a `Node start`, a `Node finish`, and distance given as integer.

1. Draw the UML diagram of the graph data structure.
2. Implement the method `getNodes` for the class Graph, which creates the hashtable of `Node`s using a given `IRange` iterator. Make sure the hashtable does not have duplicate entries.
3. Implement the method `addEdge`, which adds an edge to the graph, given the start and end `Object` and the distance.
4. Implement the method `addEdges`, which adds edges to the graph, using the given `IRange` iterator.
5. Implement the `iterator()` method for the class `Graph`, which returns an iterator for traversing over the graph nodes.
6. Implement the `iterator()` method for the class `Node`, which returns an iterator for traversing over the edges adjacent to this node.
7. Verify your work by displaying a graph of cities using the `GraphDisplay` class.

Exercise 6.2 The goal of this exercise is to implement some basic graph algorithms.
The `GraphAlgorithms` class contains the following member data:

- `FringeQueue fringeQueue` that contains the `Edge`s on the fringe, organized as stack, queue, or one of two possible priority queues.
- `HashMap fringeHash` that contains the `Node`s on the fringe.
- `HashMap visited` that contains the visited `Node`s.
- `Graph graph`, the graph on which the algorithm is performed.
- `Graph result`, the graph of relevant edges for the result.

- `Node start`, the node where the algorithm starts.
- `Node target`, the node which is the target of the algorithm.

1. Define the `GraphAlgorithms` class and implement the constructor which takes the graph, the `start` and `target Node`, and the `fringeQueue` as arguments.
2. Define the method `initialize`, which inserts the `start Node` into the `fringeHash`, and also into the `fringeQueue` with distance `0` and the `finish Node` being the same as `start Node`.
3. Define the method `processNode`, as follows:
   - If the `fringeQueue` is not empty, remove the `Edge current` from the `fringeQueue` - otherwise stop and return.
   - Remove `finish Node` in the current `Edge` from the `fringeHash`.
   - Insert `finish Node` in the `current Edge` into visited.
   - Add `finish Node` in the `current Edge` and `Edge current` into result.
   - Define an `Iterator` for the `current Node`.
   - Process each `Edge` from the `Edges` list for the `current Node` using `processEdge` method described below.
4. Define the method `processEdge`, as follows:
   - Get the `target Node` to be the `finish Node` in the `Edge` object.
   - If `target` is in the `visited`, stop and return.
   - Insert the `Edge` object into the `fringeQueue`.
   - Insert the `target Node` into the `fringeHash`.
5. Define the perform method as follows:
   - Invoke `initialize` method.
   - While the `fringeQueue` is not empty, invoke `processNode` method.
   - Return `result`.
6. Define the `getPath` method, which for a given `start` and `finish Nodes` returns a list of edges leading from `start` to `finish` in the `result Graph`.