Project 3

Class Vertex:

Instance Variables:
1. String place
2. private boolean visited
3. LinkedList <Edge> edges.

Constructor with parameter String place
this.place=place; visited by default is false; and edges is a new LinkedList.

This class has 3 methods.
1. public boolean checkVisitStatus, which checks whether the boolean visited is true or false
2. public void makeVisitTrue, makes visited to true. This is to mark the visited places to true.
3. public void makeVisitFalse, makes visited to false. This is to mark the visited places to false. We use it to reset places in case we change the path.

Class Edge:
It implements Comparable<Edge>

Instance Variables:
1. Vertex source
2. Vertex destination
3. int distance

Constructor with parameter: Vertex source, Vertex destination, int distance
this.source=source; this.destination=destination; this.distance=distance.

We have a compare to method with parameter Edge o. It compares distance of Edge 'this' to Edge o

Both classes Vertex and Edge have a basic toString function. Used in case of debugging to check what values are stored.

Class Graph:

Instance Variables:
1. Hashtable<String, Vertex> verticesNames key=vertex name and value=vertex. helps to check if vertex is there in graph or not
2. Set<Vertex> vertices; to store all the vertices in the graph
3. List<String> path = new ArrayList<>(); to record shortest path between two vertices.

Constructor has no parameter.
vertices= new Hashset<>();

This class has the following methods:
1. public void addVertex(Vertex…n)- To add unattached vertices (but have not used because there are none in the given data).
2. public void addEdge(Vertex source, Vertex destination, int distance)- in this method we check if Hashtable verticeNames has the vertices source and destination via the key, if it doesnt then we add them to both verticeNames and vertices. Here we use a private helper method addEdgeHelper to add edge between the source and destination. We also add an edge from destination to source as the edges in this case are undirected. This ensures that the path exists from both source and destination with the same weight but is not directed.
3. private void addEdgeHelper(Vertex source, Vertex destination, int distance)- this helper function adds edge and makes sure we don't have duplicate edges. It loop through to check if that edge exists. If it doesnt then we simply add the new edge with source.edges.add(new Edge(source, destination, distance));
4. public boolean hasEdge(Vertex Source, Vertex Destination)- checks if two sources have an edge between them or not.
5. public void resetVisitedVertices()- this method calls method makeVertexFalse from Class Vertex to make every vertex in Vertices to false so we can use the algorithm another time.
6. public int shortestPath(Vertex source, Vertex Destination)- here we use Djikstra's algorithm. To find the shortest path. Create a new Hashmap<Vertex, Vertex> reachedAt to keep track from previous vertex to current vertex. Create another Hashmap<Vertex, Integer> minimumCostMap to keep track of least cost to reach that vertex from the source. We set source to 0 in minimumCostMap and the rest to the maximum integer value. Ensure to mark the source as visited. Next, we loop until there is at least one unvisited vertex that can be reached from any of the vertices. We find that vertex via helper method. visitedIsFalseWithMinimumCost. If the vertex is equal to destination we add it to our path and loop until we have a previous and current and all of those to the path. At last we return the minimumCostMap.get(destination) to get the lease cost. We mark the vertex as visited.

After this, to check all the unvisited vertices that the currentVertex has an edge to whether the least cost passing through the current vertex is less than what the other vertices had before, we loop and If destination is false we update the costs if they are less.

7. private Vertex visitedIsFalseWithMinimumCost(HashMap<Vertex,Integer> minimumCostMap)- this returns the closest vertex which we haven't visited before.
8. public void printGraph()- this method was used to debug. It prints the graph.

Class Roadtrip:

Instance Variables:

1. Graph roads
2. Hashtable<String, String> attractions;

There is no constructor in this class.

The methods that exist in this class are:

1. public void storeAttractions(String A) -String A is the directory path. This file stores the attractions in the Hashtable attractions.
2. public void storeRoads(String R)- String R is the directory path. Here we makes roads= new Graph(). We add all edges that includes source, destination and the distance.
3. List<String> route(String starting_city, String ending_city, List<String> attractions)- We have a local variable: List<String> finalPath = new ArrayList<>(); finalPath stores the path we have decided. if there are no attractions, we just want to give the shortest path from from starting city to destination using shortestPath method in graphs and add those paths to finalPath. If there are attractions, we loop until shortest path to all the attractions has been found. Also, check closest place from start place. We find the shortest path to that place. Reset the listed vertices and add the path to the final path. We remove this place from the list of attractions and now repeat the process but instead of source we start with the last visited place. Once all the attractions are covered, we return finalPath.
4. private Vertex getVertex(String name): Its is a private method because it is a helper method.It is to get the vertex from the graph based on the name.
5. public void printPath()- it prints the desired path.

We also have a main method which asks the user for input of the source, destination and the attractions they want. I tried error handling but turns out it was messing up the whole code if the user added wrong input. As the project didn't specify a lot on error handling, I have not done any error handling. There are some snippets in my code which were added were for error handling but I ended up not using them.
List<String> AttractionArrayList = new ArrayList<>();
List<String> CityArrayList = new ArrayList<>();
public boolean checkAttractions(List A)
They are all redundant as they have not been used.

———————————————-—END OF DESIGN —————————————————