<u>Graph Class:</u>

```
/**
 * <b>Graph</b> represents an directed graph.
 * It is represented by a HashMap with the name of starting node for keys
 * and ArrayList of Edges get out from the node as values.
 *
 * Example of a directed graph containing node {A, B} and Edge {(A,B),(B,A)}:
 * {A:[Edge(A,B)], B:[Edge(B,A)]}
 */
public class Graph {

    private HashMap<String, ArrayList<Edge>> adj_lst;
    private int edge_num;

    /**
     * @effects Constructs a new empty Graph
     */
    public Graph(){

    }

    /**
     * @param g a existing Graph to copy from
     * @effects Constructs a copy of existing Graph g
     * @throws RuntimeException if g == null
     */
    public void Graph(Graph g){

    }

    /**
     * Edge Addition operation.
     *
     * @param e The other Edge to be added.
     * @effects add the Edge to the corresponding ArrayList if the key already exist,
     *          or add a new key with the new Edge in the ArrayList if not.
     * @throws IllegalArgumentException if e is null or e has attributes of null
     */
    public void addEdge(Edge e){

    }

    /**
     * @param node the new node you want to add to the Graph
     * @effects Add the new node to the key of HashMap if the key does not already exist.
     * @throws IllegalArgumentException if node == null
     */
    public void addNode(String node){

    }

    /**
     * @return the iterator pointing to all starting Nodes that is sorted alphabetically
     */
    public Iterator<String> nodeItr(){

    }

    /**
     * @param children
     * @effects sort the Edges alphabetically and by weight
     * @modified children
     * @return a sorted ArrayList containing all children
     */
    public ArrayList<Edge> sortChildren(ArrayList<Edge> children){

    }

    /**
     * @param parent the parent of nodes we want to iterate
     * @requires this != null
     * @return the iterator pointing to child Nodes that is sorted alphabetically and the by weight
     * @throws IllegalArgumentException if parent == null
     */
    public Iterator<Edge> childrenItr(String parent){

    }

    /**
     * Checks that the representation invariant holds (if any).
     * @throws RuntimeException if any node is null
     */
    private void checkRep() throws RuntimeException{

    }
}
```

Edge Class:

```
/**
 * <b>Edge</b> represents an Edge of directed graph.
 * It includes the name of starting node (String),
 * the name of ending node (String), and the weight
 * of the edge (String).
 */
public class Edge {

    private String start;
    private String end;
    private String weight;

    /**
     * @param s The name of the start node
     * @param e The name of the end node
     * @param w The weight of the edge
     * @effects Construct a new Edge
     */
    public Edge(String s, String e, String w){

    }

    /**
     * @return the name of start node
     */
    public String getStart(){

    }

    /**
     * @return the name of end node
     */
    public String getEnd(){

    }

    /**
     * @return the weight of the edge
     */
    public String getWeight(){

    }

    /**
     * @param w the new weight you want to set for the edge
     * @effects set the weight of the edge to the new weight
     */
    public void setWeight(String w){

    }

    /**
     * Checks that the representation invariant holds (if any).
     **/
    public void checkRep() throws RuntimeException{

    }
}
```