| CS 206 | Data Structures<br>**Project 3** | Spring 2017 |
|---|---|---|
| Sungwon Kang | | June 7, 2017 |

Part I: Implement a Weighted Undirected Graph
====================================

Implement a well-encapsulated ADT called WUG in a package called graph. A WUG represents a weighted, undirected graph in which self-edges are NOT permitted.

For maximum speed, you must store edges in two data structures: unordered doubly-linked adjacency lists and a hash table. You are expected to support the following public methods in the running times specified.

| | |
|---|---|
| O(1) | `WUG();` construct a graph having no vertices or edges. |
| O(1) | `int vertexCount();` return the number of vertices in the graph. |
| O(1) | `int edgeCount();` return the number of edges in the graph. |
| O(\|V\|) | `Object[] getVertices();` return an array of all the vertices. |
| O(1) | `void addVertex(Object);` add a vertex to the graph. |
| O(d) | `void removeVertex(Object);` remove a vertex from the graph. |
| O(1) | `boolean isVertex(Object);` is this object a vertex of the graph? |
| O(1) | `int degree(Object);` return the degree of a vertex. |
| O(d) | `Neighbors getNeighbors(Object);` return the neighbors of a vertex. |
| O(1) | `void addEdge(Object, Object, int);` add an edge of specified weight. |
| O(1) | `void removeEdge(Object, Object);` remove an edge from the graph. |
| O(1) | `boolean isEdge(Object, Object);` is this edge in the graph? |
| O(1) | `int weight(Object, Object);` return the weight of this edge. |

You may ignore hash table resizing time when trying to achieve a specified running time -- but your hash table should resize itself when necessary to keep the load factor roughly constant. |V| is the number of vertices in the graph, and d is the degree of the vertex in question. A "neighbor" of a vertex is any vertex connected to it by an edge.

Here are some of the design elements that will help achieve these goals.

[1] A calling application can use any object whatsoever to be a "vertex" from its point of view. You will also need to have an internal object that represents a vertex in a WUG and maintains its adjacency list; this object is HIDDEN from the application. Therefore, you will need a fast way to map the application's vertex objects to your internal vertex objects. The hash table also makes it possible to support `isVertex()` in O(1) time.

[2] To support `getVertices()` in O(|V|) time, you will need to maintain a list of vertices. To support `removeVertex()` in O(d) time, the list of vertices should be doubly-linked. `getVertices()` returns the objects that were provided by the calling application in calls to `addVertex()`, NOT the WUG's internal vertex data structure(s), which should ALWAYS BE

HIDDEN. Hence, each internal vertex representation must include a reference to the corresponding object that the calling application is using as a vertex.

Alternatively, you could implement `getVertices()` by traversing your hash table. However, this runs in O(|V|) time ONLY if your hash table resizes in both directions--specifically, it must shrink when the load factor drops below a constant. Otherwise, it will run too slowly if we add many vertices to a graph (causing your table to grow very large) then remove most of them.

[3] To support `getNeighbors()` in O(d) time, you will need to maintain an adjacency list of edges for each vertex. To support `removeEdge()` in O(1) time, each list of edges must be doubly-linked.

[4] Because a WUG is undirected, each edge (u, v) must appear in two adjacency lists (unless u == v): u's and v's. If we remove u from the graph, we must remove every edge incident on u from the adjacency lists of u's neighbors. To support `removeVertex()` in O(d) time, we cannot walk through all these adjacency lists. There are several ways you can obtain O(d) running time, and you may use any of the following options:

> [i] Since (u, v) appears in two lists, you could use two nodes to represent (u, v); one in u's list, and one in v's list. Each of these nodes might be called a "half-edge," and each is the other's "partner." Each half-edge has forward and backward references to link it into an adjacency list. Each half-edge also maintains a reference to its partner. That way, when you remove u from the graph, you can traverse u's adjacency list and use the partner references to find and remove each half-edge's partner from the adjacency lists of u's neighbors in O(1) time per edge.
> [ii] You could use just one object to represent (u, v), but equip it with two "next" and two "prev" references. However, you must be careful to follow the right references as you traverse a node's adjacency list.

[5] To support `removeEdge()`, `isEdge()`, and `weight()` in O(1) time, you will need a _second_ hash table for edges. The second hash table maps an unordered pair of objects (both representing application-supplied vertices in the graph) to your internal edge data structure. (If you are using half-edges, following suggestion [4i] above, you could use the reference from one half-edge to find the other.)
(Technically, you don't need a second hash table; you could store vertices and edges in the same hash table. However, you risk confusing yourself; having two separate hash tables eases debugging and reduces the likelihood of human error. But it's your decision.)
To support `removeVertex()` in O(d) time, you will need to remove the edges incident on a vertex from the hash table as well as the adjacency lists. You will also need to update the vertex degrees. Hence, each edge or half-edge should have references to the vertices it is incident on.

[6] To support `vertexCount()`, `edgeCount()`, and `degree()` in O(1) time, you will need to maintain counts of the vertices, the edges, and the degree of each vertex, and keep these counts updated with every operation.

Part II: Kruskal's Algorithm for Minimum Spanning Trees
==========================================

Implement Kruskal's algorithm for finding the minimum spanning tree of a graph. Your `minSpanTree()` method should not violate the encapsulation of the WUG ADT, and should only access a WUG by calling the methods listed in Part I. You may NOT add any public methods to the WUG class to make Part II easier (e.g., a method that returns all the edges in a WUG). Let G be the graph represented by the WUG g. Your implementation should run in $O(|V| + |E| \log |E|)$ time, where $|V|$ is the number of vertices in G, and $|E|$ is the number of edges in G.