```
                           Project 3
           Weighted Undirected Graphs and Minimum Spanning Trees


This is a team project.  Form a team of 2 or 3 people.  No teams of 1 or teams
of 4 or more are allowed.

A figure accompanies this "readme" as the files pj3graph.ps (PostScript) or
pj3graph.pdf (PDF).  Both files are the same figure.  Print this figure and
refer to it as you read the description below; the text will be much easier to
understand with the figure sitting next to it.

The purpose of this project is to become comfortable with applications that
combine a variety of data structures and algorithms in a big ol' mess of
references pointing just every which way.  I'm not even kidding.

Part I:  Implement a Weighted Undirected Graph
==============================================
Implement a well-encapsulated ADT called WUGraph in a package called graph.
A WUGraph represents a weighted, undirected graph in which self-edges are
permitted.  Any object whatsoever can serve as a vertex of a WUGraph.

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.  (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.)  Below, |V| is the number of vertices in the graph, and
d is the degree of the vertex in question.





O(1)   WUGraph();                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.


A "neighbor" of a vertex is any vertex connected to it by an edge.  See the
file graph/WUGraph.java for details of exactly how each of these methods should
behave.

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 WUGraph 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 best way to do this is to use the hash table you implemented
     for Homework 6--modified so it resizes itself to keep the load factor
     constant as |V| changes.  (You may NOT use Java's built-in hash tables.)

     The hash table also makes it possible to support isVertex() in O(1) time.

     In Java, every object has a hashCode() method.  The default hashCode()
     is defined in the Object class, and hashes the _reference_ to the Object.
     (This is not something you could do yourself, because Java does not give
     you direct access to memory addresses.)  This means that for some classes,
     two distinct objects can act as different keys, even if their fields are
     identical.  However, many object classes such as Integer and String
     override hashCode() so that items with the same fields are equals().
     Recall that Java has a convention that if two objects are equals(), they
     have the same hash code; defying this convention tends to break hash
     tables badly.  For the purposes of this project, two objects provided by
     the calling application represent the same vertex if they are equals().

[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
     WUGraph'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.  (See the dashed arrows in the accompanying figure.)

     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 WUGraph 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 these 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.  This option is illustrated in the accompanying
               figure, pj3graph.ps or pj3graph.pdf (both figures are the same).
         [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.
        [iii]  If you want to use an encapsulated DList class without changing
               it, you could use just a single object to represent an edge,
               and put this object into both adjacency lists.  The edge object
               contains two DListNode references (signifying its position in
               each DList), so it can extract itself from both adjacency lists
               in O(1) time.
```

[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.) To help you hash an edge in a manner that does not depend on the order of the two vertices, I have provided a class VertexPair.java designed for use as a key in hash tables. The methods VertexPair.hashCode and VertexPair.equals are written so that (u, v) and (v, u) are considered to be equal keys with the same hash code. Read them, but don't change them unless you know what you're doing. We recommend you use the VertexPair class as the key for your edge hash table. However, you are not required to do so, and you may change VertexPair.java freely to suit your needs.

(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.

For those of you who are keeping score, my own Part I solution is 350 lines long, not counting the hash table code.

Interfaces

You may NOT change Neighbors.java or the signatures and behavior of WUGraph.java. We will test that your WUGraph class correctly implements the interface we have specified.

Neighbors.java is a class provided so the method WUGraph.getNeighbors() can return two arrays at once. That is its only purpose. It is NOT appropriate to use the Neighbors class for any other purpose. You CANNOT change it, because it is part of the interface of getNeighbors(), and calling programs (including the autograder) are relying on you to return Neighbors objects according to spec. It appears as follows.

```
  public class Neighbors {
    public Object[] neighborList;
    public int[] weightList;
  }
```

Given an input vertex, getNeighbors() returns a Neighbors object. neighborList is a list of all the vertices (application-provided objects, not internal vertex representations) connected by an edge to the input vertex (including the input vertex itself if it has a self-edge). weightList lists the weight of each edge. The length of both lists is the degree of the input vertex. getNeighbors() should construct and return a _NEW_ Neighbors object every time it is called.

Your WUGraph should be well-encapsulated: no internal field or class used to represent your graph should be public. The Neighbors class is public because it's part of the interface of the WUGraph ADT, and it's not part of the internal representation of your graph.

Part II: Kruskal's Algorithm for Minimum Spanning Trees
===
Implement Kruskal's algorithm for finding the minimum spanning tree of a graph, in a package called graphalg. Minimum spanning trees, and Kruskal's algorithm for constructing them, are discussed by Goodrich and Tamassia, Sections 13.6-13.6.1. Your algorithm should be embodied in a static method called minSpanTree() in a class called Kruskal in a package called graphalg. Your minSpanTree() method should not violate the encapsulation of the WUGraph ADT, and should only access a WUGraph by calling the methods listed in Part I. You may NOT add any public methods to the WUGraph class to make Part II easier (e.g., a method that returns all the edges in a WUGraph). Remember this rule of encapsulation: your Kruskal code should work correctly with the WUGraph code of any other group taking CS 61B, and your WUGraph code should work with their Kruskal code.

The signature of minSpanTree() is

```
  public static WUGraph minSpanTree(WUGraph g);
```

This method takes a WUGraph g and returns another, newly constructed WUGraph that represents the minimum spanning tree of g. The original WUGraph g is NOT changed! Let G be the graph represented by the WUGraph 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.

Kruskal's algorithm works as follows.

[1] Create a new graph T having the same vertices as G, but no edges (yet). Upon completion, T will be the minimum spanning tree of G.

[2] Make a list of all the edges in G. (This can be an array or a linked list; it's up to you.) You cannot build this list by calling isEdge() on every pair of vertices, because that would take O(|V|^2) time. You will need to use multiple calls to getNeighbors() to obtain the complete list of edges.

Note that your edge data structure should be defined separately from any edge data structure you use in WUGraph.java (Part I). Encapsulation requires that the internal data structures of the WUGraph class not be exposed to applications (including Kruskal).

[3] Sort the edges by weight in O(|E| log |E|) time. You may write the sorting algorithm yourself or use one from your homework or lab, but do NOT use a sorting method from the Java standard libraries. (Instead of a sorting algorithm, you can instead use a priority queue as Goodrich and Tamassia suggest, but sorting in advance is more straightforward and is probably faster.)

[4] Finally, find the edges of T using disjoint sets, as described in Lecture 33 and Goodrich & Tamassia Section 11.4. The disjoint sets code from Lecture 33 is included in DisjointSets.java in the package called set.

To use the disjoint sets code, you will need a way to map the objects that serve as vertices to unique integers. Hash tables are a good way to accomplish this. (You cannot use the same hash table as the WUGraph; that hash table should be encapsulated so Kruskal can't access it.)

Be forewarned that the DisjointSets class has no error checking, and will fail catastrophically if you union() two vertices that are not roots of their respective sets, or if you union() a vertex with itself. If you add simple error checking, it might save you a lot of debugging time (here and in Homework 9).

```
For those of you who are keeping score, my own Part II solution is 100 lines
long, not counting the sorting method or the hash table code.

Since Parts I and II are on opposite sides of the WUGraph interface, a partner
can easily begin Part II before Part I is working.

Style Rules
===========
You will be graded on style, documentation, efficiency, and the use of
encapsulation.

  1)  Every method must be preceded by a comment describing its behavior
      unambiguously.  These comments must include descriptions of what each
      parameter is for, and what the method returns (if anything).
      They must also include a description of what the method does (though
      not how it does it) detailed enough that somebody else could implement
      a method that does the same thing from scratch.
  2)  All classes, fields, and methods must have the proper public/private/
      protected/package qualifier.  We will deduct points if you make things
      public that could conceivably allow a user to corrupt the data structure.
  3)  We will deduct points for code that does not match the following style
      guidelines.

  - Classes that contain extraneous debugging code, print statements, or
    meaningless comments that make the code hard to read will be penalized.
    (It’s okay to have methods whose sole purpose is to contain lots of
    debugging code, so long as your comments inform the reader who grades your
    project that he can skip those methods.  These methods should not contain
    anything necessary to the functioning of your project.)
  - Your file should be indented in the manner enforced by Emacs (e.g., a
    two-space or four-space indentation inside braces), and used in the lecture
    notes throughout the semester.  The indentation should clearly show the
    structure of nested statements like loops and if statements.  Sloppy
    indentation will be penalized.
  - All if, else, while, do, and for statements should use braces.
  - All classes start with a capital letter, all methods and (non-final) data
    fields start with a lower case letter, and in both cases, each new word
    within the name starts with a capital letter.  Constants (final fields) are
    all-capital-letters only.
  - Numerical constants with special meaning should always be represented by
    all-caps "final static" constants.
  - All class, method, field, and variable names should be meaningful to a
    human reader.  (Exception:  short loop index variables like "i" are okay if
    their meaning is obvious from context.)
  - Methods should not exceed about 100 lines; any method that long can
    probably be broken up into logical pieces.  The same is probably true for
    any method that needs more than 8 levels of indentation.
  - Avoid unnecessary duplicated code; if you use the same (or very similar)
    fifteen lines of code in two different places, those lines should probably
    be a separate method call.
  - Programs should be easy to read.
```

```
Submitting your Solution
========================
Write a file called GRADER that briefly documents your data structures and the
design decisions you made in WUGraph.java and Kruskal.java that extend or
depart from those discussed here.  In particular, tell us what choices you made
in your implementation to ensure that removeVertex() runs in O(d) time (as
described in Part I, design element [4]) and getVertices() runs in O(|V|) time
(design element [2]).

Designate one member of your team to submit the project.  The designated
teammate only:  make sure your project compiles and runs (with WUGTest and
KruskalTest) just before you submit.


Project 3 directory, which should contain your GRADER, the graph directory
(package), the graphalg directory (package), the set directory
(package), the dict directory (package) containing your hash table, and
possibly a list directory (package) if you choose to use an encapsulated list
ADT.  The graph directory should contain your WUGraph.java and (if you use it)
VertexPair.java.  The graphalg directory should contain your Kruskal.java.
The set directory should contain whatever code you are using for disjoint sets.

If you are using VertexPair.java and/or DisjointSets.java, you must submit them
because you’re allowed to change them. Make sure any other files your project
needs, including a dictionary ADT and possibly a list ADT, are present as well.

You may submit as often as you like.  Only the last version you submit will be
graded.
```