CS 61B Project 3
Weighted Undirected Graphs and Minimum Spanning Trees
Due 5pm Friday, December 3, 2010

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.

Copy the Project 3 directory by doing the following, starting from your home directory.

```
cp -r ~cs61b/hw/pj3 .
```

A figure accompanies this "readme" as the files pj3graph.ps (PostScript) or pj3graph.pdf (PDF). Both files are the same figure.

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 allowed. 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.) 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] You will need a way to map each vertex provided by the calling application to internal data structures, such as the adjacency list for the vertex. The best way to do this is to use a hash table. However, _any_ object may serve as a vertex, even if it doesn't have a hashCode() method.

Fortunately, Java has a built-in Hashtable class (with a lower-case t) that makes it possible to use any object as a key. By default, Java's hash tables hash the _reference_ to an object. (This is not something you could do yourself, because Java will not give you direct access to memory addresses.) This means that two distinct objects act as different keys, even if their fields are identical. Further information on Java's built-in hash tables is included below.

You will need to have an internal data structure that represents a vertex in a WUGraph, and you will need to use a Hashtable to map a vertex provided by the application to the corresponding internal data structure. The Hashtable 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 WUGraph's internal vertex data structure(s). Hence, each internal vertex representation must include a reference to the corresponding object that the calling application is using as a vertex.

[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 could obtain O(d) time, and you may use any of these options:

[i] Since (u, v) appears in two lists, we 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 we remove u from the graph, we 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 data structure to represent (u, v), but equip it with two forward and two backward 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 a DList class, you could use just a single data structure to represent an edge, and put this structure into both adjacency lists. The edge data structure 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_ Hashtable for edges. The second Hashtable 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 Java's Hashtable class. (The methods VertexPair.hashCode and VertexPair.equals are written so that (u, v) and (v, u) are considered to be equal keys in a Java Hashtable; don't change them unless you know what you're doing.)

Technically, you don't need a second Hashtable; you can store vertices and edges in the same table. However, you risk confusing yourself; having two separate Hashtables 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 adjust 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 each operation.

My own Part I solution is 350 lines long.

Java Hash Tables

Java Hashtables are contained in java.util.Hashtable and documented in Sun's Java library API Web pages.

The most salient methods of the Hashtable class (for this project) appear below. Note that you should never pass null as a key or value; be careful with your error checking. Note that these methods return values only, not (key, value) pairs.

```
public Hashtable()
    Constructs a new, empty hash table.

public synchronized Object put(Object key, Object value)
    Maps the specified key to the specified value in this hash table.  Neither
    the key nor the value can be null.  The value can be retrieved by calling
    the get() method with a key that is equals() to the original key.

    Returns:
        the previous value of the specified key in this hash table, or null if
        it did not have one.

public synchronized boolean containsKey(Object key)
    Tests if the specified object is a key in this hash table.

    Returns:
        true if the specified object is a key in this hash table; false
        otherwise.

public synchronized Object get(Object key)
    Returns the value to which the specified key is mapped in this hash table.

    Returns:
        the value to which the key is mapped in this hash table; null if the
        key is not mapped to any value in this hash table.

public synchronized Object remove(Object key)

    Removes the key (and its corresponding value) from this hash table.  This
    method does nothing if the key is not in the hash table.

    Returns:
        the value to which the key had been mapped in this hash table, or null
        if the key did not have a mapping.
```

You should NOT use the contains() method (which searches for a value, rather than a key, in the Hashtable), because it has to search the entire Hashtable, and will not run within the specified time bounds.

Likewise, avoid the keys() and elements() methods, which are not certain to run in linear time because the hash tables are not necessarily resized to make them smaller when keys are deleted from the table.

Java Hashtables use generics to allow you to specify a class of objects for the keys, and another class for the values. You are welcome to take advantage of this, or not. See Sun's Java library API for information.

Java's Hashtables can use any object as a key. Hashtables do this by using the hashCode() and equals() methods, which are defined on every object, but can be overridden. By default, hashCode() hashes an object's reference, and equals() declares two objects to be equal only if they are the same object. Hence, some classes of objects can serve as unique keys, even if other objects of the same class have identical contents. However, there are also many classes that override both methods and replace them with data-dependent hashCode() and equals() methods. For example, two distinct Integer or String objects that contain the same data will return the same hashCode() and be found to be equals().

We have provided you a VertexPair class expressly for use with Java's Hashtable class, to serve as a key for an edge. The class holds a pair of objects that serve as vertices from the application's point of view. Two VertexPairs that contain the same vertices in different orders are considered to be the same. Hence, hashCode() returns the same integer for (u, v) as (v, u), and (u, v) equals() (v, u). See VertexPair.java for more details.

We recommend you use the VertexPair class as the key for your edge hashTable. However, you are not required to do so, and you may change VertexPair.java freely to suit your needs.

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. We do not recommend using the Neighbors class for any other purpose. The reason you cannot change it is because it is part of the interface of getNeighbors(). 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 each 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. 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, which is NOT in the graph package. 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).

The signature of minSpanTree() is:

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

This method takes a WUGraph g and returns another 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 (not necessarily linked) of all the edges in G. 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. If you wish, you may use one of Java's library methods to do this; if you do, your edge data structure must implement the Comparable interface, and its compareTo() method must be public. (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 "set" package/ directory. To use the disjoint sets code, you will need a way to map the objects that serve as vertices to unique integers. Again, Java's Hashtable class is a good way to accomplish this.

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).

My own Part II solution is 100 lines long.

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) Each 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.
- 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.
- 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 in all capital letters.
- 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.
- 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 7 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.

The Autograders

If possible, make sure that your program passes both of the test programs provided, WUGTest.java and KruskalTest.java. IMPORTANT NOTE: If you attempt to cheat and thwart the test code by writing code that looks for specific tests and provides canned answers, rather than by writing code that correctly implements a weighted undirected graph data structure and Kruskal's algorithm, the graders will notice, and you will receive a score of -20 and a letter at the Office of Student Conduct. Please don't try it.

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]).

Designate one member of your team to submit the project. If you resubmit, the project should always be submitted by the same student. If for some reason a different partner must submit (because the designated member is out of town, for instance), you must send cs61b@cory.eecs a listing of your team members, explaining which of them have submitted the project and why. Let us know which submission you want graded.

The designated teammate only: Change (cd) to your pj3 directory, which should contain your GRADER, your Kruskal.java, the graph directory (package), the set directory (package), 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 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; the autograder won't supply the original files. Make sure any other files your project needs, possibly including a list ADT, are present as well. You won't be able to submit Neighbors.java, because you're not allowed to change it.

Make sure your project compiles and runs on the _lab_ machines (with WUGTest and KruskalTest) just before you submit. Type "submit pj3".

You may submit as often as you like. Only the last version you submit will be graded, unless you send email to cs61b@cory.eecs asking that an earlier version be graded.