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 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_.

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 list includes 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.

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.

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 $u$ and $v$. 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:

1. 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, p3graph.ps or p3graph.pdf (both figures are the same).

2. You could use just one object 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.

3. If you want to use an encapsulated `DList` class, 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.

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
readme

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 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 adjust the vertex degrees. Hence, each edge or half-edge should have references to the vertices it is incident on.

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.

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). Remember the encapsulation rule: 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 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.
3. 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).
4. Sort the edges by weight in \( O(|E| \log |E|) \) time. You may write the sorting algorithm yourself or use one from lab, but do not use a Java library sorting method. (You can instead use a priority queue, as Goodrich and Tamassia suggest, but sorting in advance is more straightforward and is probably faster.)
5. 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, hash tables are a good way to accomplish this. (You cannot use the same hash table as the WUGraph; that should be encapsulated so Kruskal can’t see 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).

My own Part II solution is 100 lines long, not counting the sorting method or 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) 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, WUGraphTest.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]) and getVertices() runs in O(\(V\)) time (design element [2]).

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: make sure your project compiles and runs on the _lab_ machines (with WUGraphTest and KruskalTest) just before you submit. Change (cd) to your pj3 directory, which should contain your GRADER, your Kruskal.java, the graph 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 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.

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.