Implementing Shortest Path Algorithms
In lab 10 you need to implement a shortest path algorithm. The lab leaves the design of this up to you. Here are some suggestions for this.

We will store the graph as a HashMap that associates Strings (the names of the nodes) to Vertex objects.

A Vertex has a name, a linked list of edges to adjacent vertices, and some variables that are used in the working of the algorithms, such as pointer to the previous node in the path to this vertex and the weight of the best path found so far.

An Edge is simple -- just a vertex for the destination and, for weighted graphs, the cost of the edge.
We construct a Graph from its edges. The details for how this works depends on the source of input, but if you have a text file with entries like

A B 3

you can read this as representing an edge from vertex A to vertex B with weight 3. To insert this information into the graph, get the Vertex objects with names A and B from the hashmap for the graph (insert new objects for A and B if there are none in the hashmap), then add an edge to B of weight 3 to vertex A's list of edges.

The next page has the code for the unweighted shortest path algorithm.
public void shortestPaths(String nodeName) {
    LinkedList<Vertex> queue = new LinkedList<Vertex>();
    Vertex S = getVertex(nodeName);
    S.distance = 0;
    queue.offer(S);
    while (queue.size() > 0) {
        Vertex X = queue.poll();
        for (Edge e: X.adjacentList) {
            Vertex Y = e.destination;
            if (Y.distance == INFINITY) {
                Y.distance = X.distance + 1;
                Y.previous = X;
                queue.offer(Y);
            }
        }
    }
}
If you want to implement the non-negative weights version of the shortest path (Dijkstra's algorithm) you need a priority queue of vertices. Java's PriorityQueue<E> class does not support changing the priorities of objects once they are in the queue. One way around that is to add a new copy of the vertex with lower priority. We can't have the same object in the queue more than once. One way around this is to make a simple Entry class that contains a node and its weight. You can have a PriorityQueue of these Entries, with the same vertex represented by several different Entries. You need to make sure that once a node comes out of the queue you never put it back into the queue, so give the Vertex class a variable that you can use to signal that it has its final weight -- I use a boolean variable done.