Cole Stephens
Data Structures - Dijkstra's Algorithm, `Comparator` and `implements Comparable` handout.

Dijkstra's Algorithm

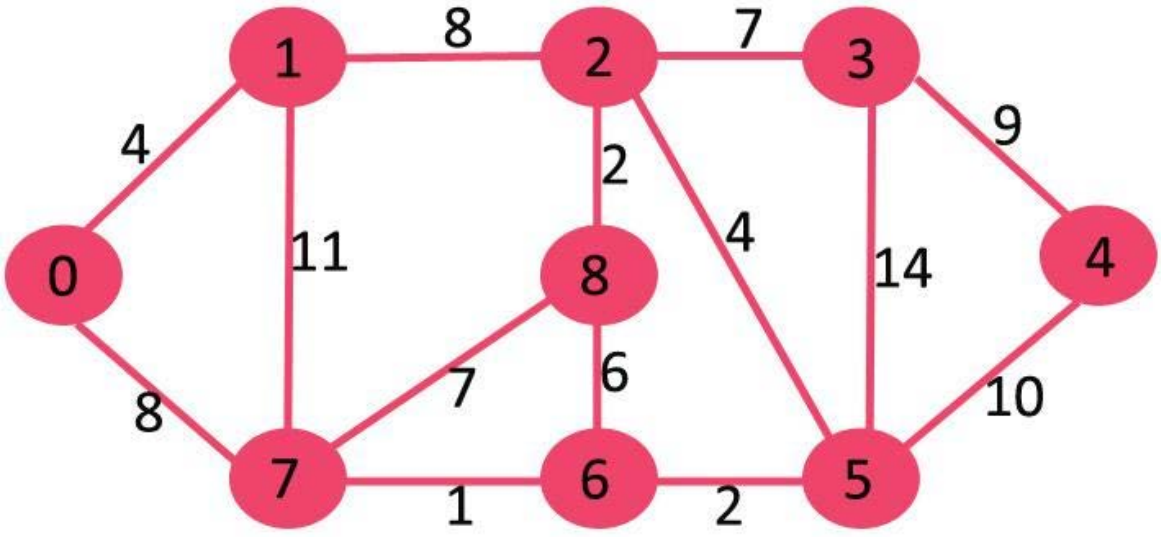
Pseudocode

```
for each vertex v in graph :                    //initialization
        v.dist = +infinity;
        v.parent = null;

source.dist = 0
insert all vertices (or, nodes) into priority queue PQ, ordering vertices based on distance from the source (see comparator section)

while !PQ.isEmpty():
        u = PQ.poll();
        //if we wanted to remember the distances to each vertex, we would record u's distance here: upon removal, the shortest path to u has been found
        for each of u's neighbors, v, *which are in the PQ*: //do not consider neighbors which have already been removed
                pathThroughU = u.dist + edgeCostFromUToV
                if pathThroughU < v.dist
                        v.dist = pathThroughU
                        v.parent = u
```

Example graph (try this on your own)



```
Class Node implements Comparable<Node> {

      public int dist;

      /* return -1 if this Node's distance is less than other Node's
      distance
      return 0 if the distances are equal
      return 1 if this Node's distance is greater than other Node's
      distance
      */
      public int compareTo(Node other) {
            return Integer.valueOf(dist).compareTo(Integer.valueOf(
            other.dist);

      }
}
```

//This way is useful if you plan on calling Arrays.sort or any other method that relies on "natural ordering" as you will not need to do any additional work after implementing `Comparable`

```
Class NodeComparator implements Comparator<Node> {

      /* Returns any positive value if a > b, 0 if a == b, and returns
      any negative value if b > a */
      public static int compare (Node a, Node b) {
            return a.dist - b.dist

      }

}
PriorityQueue<Node> orderedNodes = new PriorityQueue(new
NodeComparator());
```

//To construct the `PriorityQueue` in a way that it compares based on what you have written in your `Comparator`, pass in the `Comparator`. This way is useful for if you want to compare the same objects in different ways and want to use multiple different comparison methods. I.e., to compare `Cat` objects based on size, loudness of meow, or color, you could write three different `Comparators`, as it would be much harder to implement three different `Cat` classes with different `compareTo(Cat other)` methods.