```
import weiss.nonstandard.PriorityQueue;
import weiss.nonstandard.PairingHeap;
import weiss.nonstandard.BinaryHeap;
class Edge {
    public Vertex     dest;   // Second vertex in Edge
    public double     cost;   // Edge cost
    public Edge( Vertex d, double c )  {   dest = d;  cost = c;  }
}

class Path implements Comparable
{// Represents an entry in the priority queue for Dijkstra's algorithm.
    public Vertex     dest;   // w
    public double     cost;   // d(w)
    public Path( Vertex d, double c )  {  dest = d;  cost = c;  }
    public int compareTo( Object rhs ) {
        double otherCost = ((Path)rhs).cost;
        return cost < otherCost ? -1 : cost > otherCost ? 1 : 0;
    }
}

class Vertex{
    public String     name;   // Vertex name
    public List       adj;    // Adjacent vertices
    public double     dist;   // Cost
    public Vertex     prev;   // Previous vertex on shortest path
    public int        scratch;// Extra variable used in algorithm
    public Vertex( String nm ) { name = nm; adj = new LinkedList( ); reset( ); }
    public void reset( )
      { dist = Graph.INFINITY; prev = null; pos = null; scratch = 0; }
    public PriorityQueue.Position pos;  // Used for dijkstra2 (Chapter 23)
}
```

```
public class Graph
{
    public static final double INFINITY = Double.MAX_VALUE;
    private Map vertexMap = new HashMap( ); // Maps String to Vertex
    public void addEdge( String sourceName,
              String destName, double cost )
    {
        Vertex v = getVertex( sourceName );
        Vertex w = getVertex( destName );
        v.adj.add( new Edge( w, cost ) );
    }
   public void printPath( String destName );
   private Vertex getVertex( String vertexName );
   {
        Vertex v = (Vertex) vertexMap.get( vertexName );
        if( v == null ) {
            v = new Vertex( vertexName );
            vertexMap.put( vertexName, v );
        }
        return v;
    }
public static void main( String [ ] args )
  {
      Graph g = new Graph( );
      try  {
         FileReader fin = new FileReader( args[0] );
         BufferedReader graphFile = new BufferedReader( fin );
         String line;  // Now Read the edges and insert
         while( ( line = graphFile.readLine( ) ) != null )    {
            StringTokenizer st = new StringTokenizer( line );
            try  {
               if( st.countTokens( ) != 3 )  {
                  System.err.println( "Skipping ill-formatted line " + line );
                  continue;
               }
               String source  = st.nextToken( );
               String dest    = st.nextToken( );
               int    cost    = Integer.parseInt( st.nextToken( ) );
               g.addEdge( source, dest, cost );
            }
            catch( NumberFormatException e )
             { System.err.println( "Skipping ill-formatted line " + line ); }
         }
      }
      catch( IOException e )  { System.err.println( e ); }

      System.out.println( "File read..." );
      System.out.println( g.vertexMap.size( ) + " vertices" );
      BufferedReader in = new BufferedReader( new InputStreamReader( System.in ) );
      while( processRequest( in, g ) )
         ;
  }
}
```

```
public void dijkstra( String startName ) {
        PriorityQueue pq = new BinaryHeap( );
        Vertex start = (Vertex) vertexMap.get( startName );
        if( start == null ) throw new NoSuchElementException( "Start vertex not found" );
        clearAll( );
        pq.insert( new Path( start, 0 ) ); start.dist = 0;
        int nodesSeen = 0;

        while( !pq.isEmpty( ) && nodesSeen < vertexMap.size( ) )   {
            Path vrec = (Path) pq.deleteMin( );
            Vertex v = vrec.dest;
            if( v.scratch != 0 ) continue;   // already processed v
            v.scratch = 1;
            nodesSeen++;

            for( Iterator itr = v.adj.iterator( ); itr.hasNext( ); )
            {
                Edge e = (Edge) itr.next( );
                Vertex w = e.dest;
                double cvw = e.cost;
                if( cvw < 0 )  throw new GraphException( "Graph has negative edges" );
                if( w.dist > v.dist + cvw )   {
                    w.dist = v.dist +cvw;
                    w.prev = v;
                    pq.insert( new Path( w, w.dist ) );
                }
            }
        }
    }
```

```
private void printPath( Vertex dest )
   {
       if( dest.prev != null )
       {
           printPath( dest.prev );
           System.out.print( " to " );
       }
       System.out.print( dest.name );
   }
```

```
public void dijkstra2( String startName )  {
      PriorityQueue pq = new PairingHeap( );
      Vertex start = (Vertex) vertexMap.get( startName );
      if( start == null )
          throw new NoSuchElementException( "Start vertex not found" );
      clearAll( );
      start.pos = pq.insert( new Path( start, 0 ) ); start.dist = 0;
      while ( !pq.isEmpty( ) )  {
         Path vrec = (Path) pq.deleteMin( );
         Vertex v = vrec.dest;
         for( Iterator itr = v.adj.iterator( ); itr.hasNext( ); )  {
            Edge e = (Edge) itr.next( );
            Vertex w = e.dest;
            double cvw = e.cost;
            if( cvw < 0) throw new GraphException( "Graph has neg edges" );
            if( w.dist > v.dist + cvw ) {
               w.dist = v.dist + cvw;
               w.prev = v;
               Path newVal = new Path( w, w.dist );
               if( w.pos == null )    w.pos = pq.insert( newVal );
               else                   pq.decreaseKey( w.pos, newVal );
            }
         }
      }
   }
```