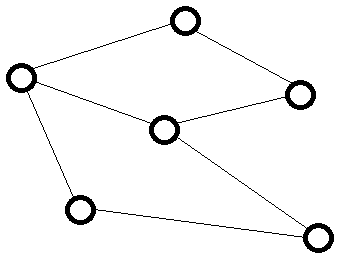
Program 5 : Linked Lists (60 Points)

**Objectives:**

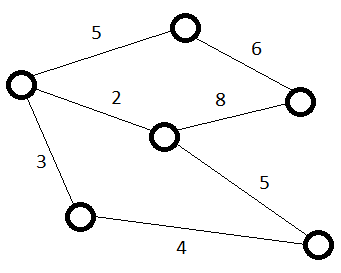
The purpose of this assignment is understanding of the Linked List structure by implementing a moderately complex algorithm. Use the LinkedList class that is defined in the java API, in the java.util package.

**Program Background:**

The data structures seen thus far have all been linear in nature; that is, each element has at most one predecessor and one successor which essentially “connect” elements or nodes. A Graph is a structure that allows a node to be connected to any number of other nodes. In a Graph we call a node a vertex.



The Edges that connect vertices often have a weight associated with them.



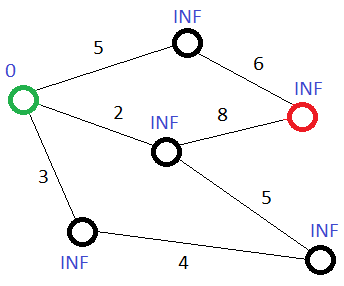
Vertices and edges can be representative of many things. Vertices may store data, represent a physical location, or represent a state. Edges may be a distance, a cost, or some unit of work required to transition from one state to another.

Whatever the Graph represents, it is often needed to find the shortest path from one vertex to the other, where the path is the sum of the edges traveled in moving from the start vertex to the destination vertex. Dijkstra’s Algorithm is a process that finds this shortest path.

*Dijkstra’s Algorithm*

Dijkstra’s Algorithm works by storing the shortest path distance from the start at each vertex. The vertex that is the closest is then processed, updating the distance of its connected vertices (called relaxing), until all of the vertices have gone through this process. Once this is done, the value in the destination vertex with have the cost of the shortest path.

Initially, we set the distance at the start vertex to 0, and all of the other vertices to infinity:



The green vertex is our start, the red is our destination. The blue values above the vertices are the distance that vertex is from start. The general algorithm can be stated as such:

**while** there are unprocessed nodes (Dijkstra)

u = lowest value vertex that has not been processed (extractMinVertex)

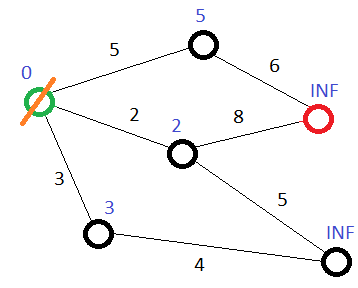
mark u as being processed

**for** each adjacent vertex v (relaxEdges)

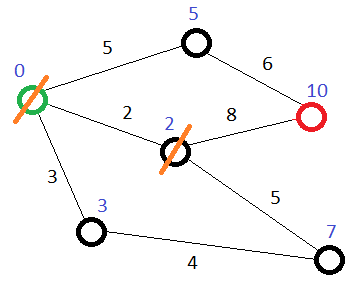
**if** u.distance + edge weight < v.distance (setDistanceFromStart)

v.distance = u.distance + edge weight

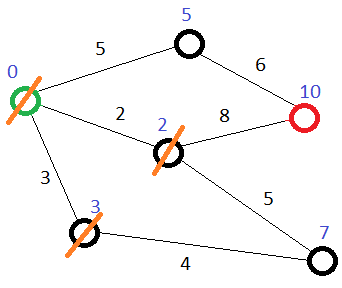
The first pass of the while loop, the start vertex has the smallest distance, so it is processed first. Its value is added to the edge value of each vertex it is connected to, and those vertices’ values are updated. The start vertex is marked as being processed so it won’t be selected again. The parenthesis denote the methods you will create that perform the adjacent functionality.



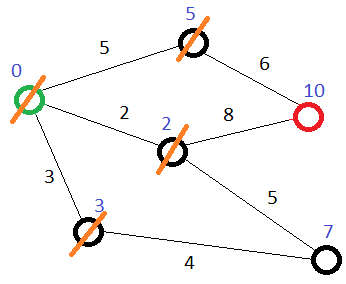
There are still vertices to be processed, so the unprocessed vertex with the smallest distance is selected, in this case the vertex with the distance of 2. Similarly, all of its adjacent vertices are updated if the current vertex’s value added to the edge weight is less than the adjacent vertex’s value. Two of the adjacent vertices will be changed, however the start vertex will not, as 2 + 2 is not less than its current value of 0.



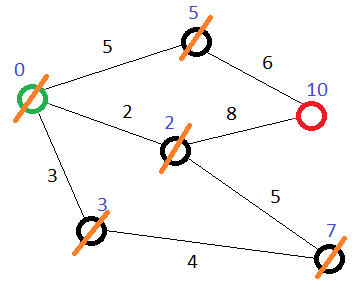
The next vertex will be the 3 distance vertex, however it will not change either of its adjacent vertex values as the sum of its distance from start added to the edge weight is not strictly less than the recorded distance of its neighbors.

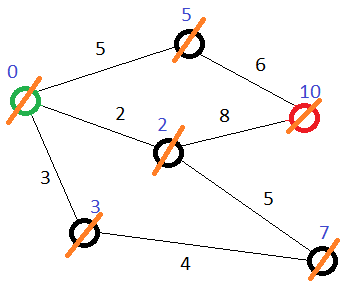


The next vertex is the one of 5 distance, and again doesn’t not provide a shorter route to either of its neighbors.



The same will be true for the 7 and the last vertex, which is our destination.



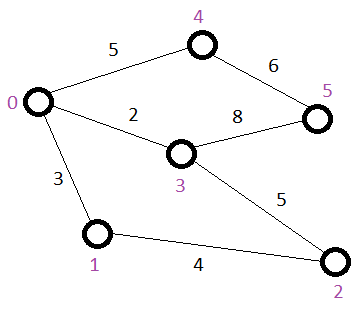


With all vertices processed, the algorithm is finished, and the value stored at the destination is the total distance of the shortest path.

You can view another example here: <https://www.youtube.com/watch?v=WN3Rb9wVYDY>

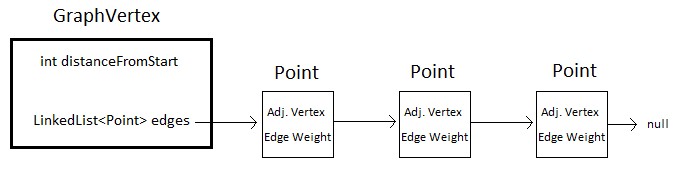
*Graph Implementation*

There are many ways to represent a Graph in code. This assignment will have you doing so with linked lists. There are two classes that will be used to implement the Graph. The first is GraphVertex, which will be a single vertex in the graph. Every vertex will store its distance from the start as well as a LinkedList of edges to other vertices. In order to do this, the nodes will need to be labeled or identified in some way. We will do this by assigning nodes integer values.



The numbers in purple are the vertex labels. So to define an edge from a particular vertex would require storing a pair of integers, the adjacent vertex number and the weight of the edge to that vertex. Java provides the Point class which is simply a pair of integers. An instance of Point has two int data members, x and y, which are public so can be accessed directly. The Point class is found in the java.awt package.

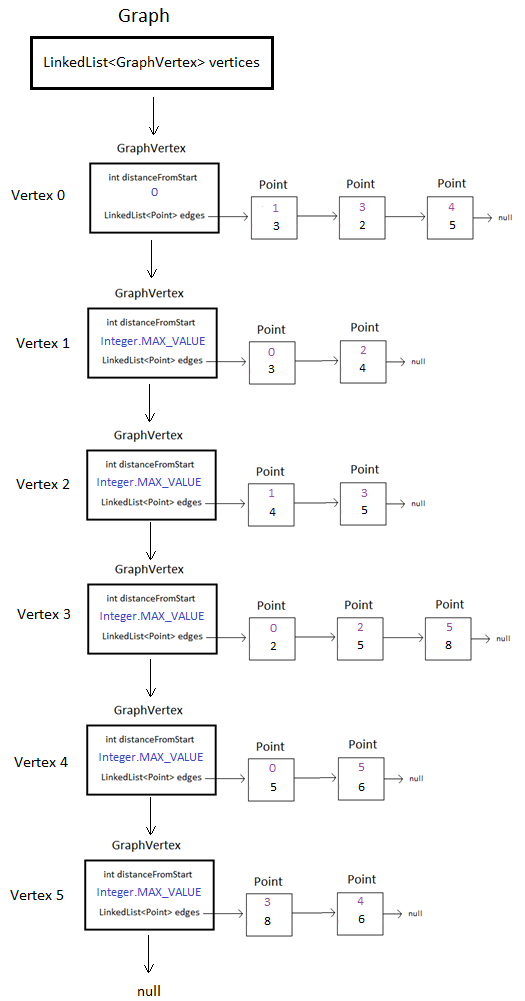
Our LinkedList of edges can be a LinkedList of Points.



You may visualize a *single* GraphVertex like this. It contains an integer that is its distance from the start vertex, as well as a list of Points, where each Point is an edge to an adjacent vertex.

Note on Edges: Since this is an undirected Graph, each edge will be represented twice, for example, Vertex 0 will have the Point (1, 3) for the edge to vertex 1 with weight of 3. Vertex 1 will have the Point (0, 3) for the same edge connecting it to vertex 0. This same algorithm works on directed Graphs as well! In fact, this implementation of a Graph will allow for directed Graphs.

GraphVertex represents a single vertex in the Graph (edges included), and a Graph is simply a collection of vertices, so the Graph class will only need the single data member of a LinkedList of GraphVertex. Notice that the GraphVertex class does not store the numerical label for the vertex. This is not necessary as it will be determined by its location in the Graph’s vertex list. That is vertex 0 will be in position 0 in the list, vertex 1 will be in position 1, and so on. Order is also import in that the way the algorithm is designed for this assignment, the vertex at position 0 will be the starting vertex and the one at the end of the list will be the destination. Keep this in mind when designing and inputting your test cases.



The Graph as a whole may be visualized as above. This specific example represents the start state of the Graph used in the previous examples. It is important that you understand how this represents the visual Graph used in the examples; it will be necessary in writing the code that implements Dijkstra’s Algorithm. Since there is not a value for infinity the maximum value an int can hold is used. This can be accessed as a static member of the Integer class called MAX\_VALUE.

**Program Description:**

The program will accept input representing a Graph. The first vertex input will be the start vertex, the last vertex input will be the destination vertex for the purpose of shortest path finding using Dijkstra’s Algorithm

The following classes are required:

* NoSuchVertexException
* GraphVertex
* Graph
* DijkstraDriver (provided)

**UML DIAGRAM FOR AND DISCUSSION FOR NoSuchVertexException**

|  |
| --- |
| NoSuchVertexException extends RuntimeException |
|  |
| <<constructor>> NoSuchVertexException(vert : int) |

Methods:

NoSuchVertexException(vert : int) – Class constructor. Calls upon super class constructor with the String, “Vertex does not exist: “, followed by the int that is passed.

**UML DIAGRAM FOR AND DISCUSSION FOR GraphVertex**

|  |
| --- |
| GraphVertex **implements** Comparable<GraphVertex> |
| * distanceFromStart : int * edges : LinkedList<Point>   + START\_VERTEX : boolean = true  + INTERIOR\_VERTEX : boolean = false |
| <<constructor>> GraphVertex(vertexType : boolean)  + addEdge( vertex : int, distance : int )  + getDistanceFromStart( ) : int  + setDistanceFromStart( distance : int )  + relaxEdges( g : Graph )  + compareTo( gv : GraphVertex ) |

Data Members:

distanceFromStart – Denotes the distance from the start vertex this vertex is

edges – LinkedList of Points for the edges that connect this vertex to adjacent vertices (as discussed above)

START\_VERTEX/INTERIOR\_VERTEX – In the UML diagram, underlined members are static, variables in all caps are final. There are two types of vertices that can be created, a start vertex and all other vertices. When the constructor is called upon you will need to let it know which type you want. Since there are only two possibilities a boolean can be used for this choice. True will mean a start vertex, false will mean other vertices. It can make usage easier if variables are created in the class to name these constant values that it uses.

An instance of GraphVertex that is a start vertex could be created like this:

new GraphVertex(true)

However, looking at this it isn’t clear what is being made. With the static constants we can instead do this:

new GraphVertex(GraphVertex.START\_VERTEX)

This does the same thing, as GraphVertex.START\_VERTEX has the value of true, but our intentions are much clearer. Additionally, we won’t get confused on what true represents and what false represents.

Methods:

GraphVertex( vertexType : boolean) – Class constructor. Creates new LinkedList of Points. If the vertex is to be a start vertex sets distanceFromStart to 0, otherwise sets it to Integer.MAX\_VALUE.

addEdge( vertex : int, distance : int) – Adds an edge to the edges list. An edge is represented by a Point, the adjacent vertex number (x) and the weight of the edge that connects them (y).

getDistanceFromStart( ) – Returns the value of distanceFromStart.

setDistanceFromStart( distance : int ) – Updates the value of distanceFromStart to the passed value *only if* the passed value is less than the current value.

relaxEdges( g : Graph ) – Updates the adjacent vertices distance values. In order to update other vertices this method needs the entire Graph which is passed to the method. Go through the edge list and update the adjacent vertices using the Graph method setVertexDistanceFromStart.

compareTo( gv : GraphVertex ) – compareTo implementation based upon distanceFromStart. This will be used to find the vertex with the shortest path distance required per the algorithm

**UML DIAGRAM FOR AND DISCUSSION FOR Graph**

|  |
| --- |
| Graph |
| * vertices : LinkedList<GraphVertex> |
| <<constructor>> Graph( )  + addVertex( )  + addEdge( vertex1 : int, vertex2 : int, distance : int )  + setVertexDistanceFromStart( vertex : int, distance : int )  + Dijkstra ( ) : int |
| + extractMinVertex( visited : boolean[] ) : GraphVertex |

Data Members:

vertices – LinkedList of all GraphVertexes in the Graph

Methods:

Graph( ) – Class constructor. Creates LinkedList of GraphVertexes.

addVertex( ) – Creates a new GraphVertex and adds it to the list. If the vertices LinkedList is empty (size of 0) the GraphVertex created will be a start vertex, otherwise it is created as an interior vertex.

addEdge( vertex1 : int, vertex2 : int, distance : int ) – Adds a new edge to one of the vertices in the vertices list. vertex1 is the vertex to add the edge to, vertex2 is that vertex’s adjacent vertex, distance is the weight of the edge that connects them. If the value of vertex1 or vertex2 is out of the bounds of the current number of vertices in the list it will throw a NoSuchVertexException, passing the offending vertex value.

setVertexDistanceFromStart( vertex : int, distance : int ) – Sets the passed vertex’s distance from the start vertex.

Dijkstra( ) – Implements Dijkstra’s Algorithm as discussed above, returning the distance the final vertex in the list’s distance from start. We can keep track of which vertices have been processed by creating a boolean array the size of the number of vertices that are in the Graph, initializing all values to false (meaning they have not been processed). The extractMinVertex method will return the GraphVertex with the smallest distance from start value, or null if all vertices have been processed. Use this for the condition of the outer loop in the algorithm pseudocode above. For each of the vertices that are successfully extracted, call upon their relaxEdges method, passing the Graph (use the **this** keyword). Finally, return the distance from start of the last vertex in the vertices list.

extractMinVertex( visited : boolean[] ) – Find the smallest vertex (as defined by GraphVertexes compareTo method) that has not been processed (using the visited array). Once the vertex that matches these conditions is found, update its value in the visited array to true and return it. If all of the vertices have been processed (again determined by the visited array) return null instead.

**DISCUSSION FOR DijkstraDriver**

See expected output below discussion, if your output is different in any way you are subject to losing points.

The Graph is input by first entering the number of vertices. Following that, the edges of each vertex is entered, starting at vertex 0. Edges are input as pairs, vertex number and edge weight. To finish entering edges for the current vertex enter an invalid vertex for an edge pair (such as -1 -1). Once the edges are entered for the last vertex the Dijkstra method will be executed and the shortest distance from start to destination will be output.

**Helpful methods:**

You may find the following methods in the java API helpful

LinkedList class methods:

public boolean add(E e) - Appends the specified element to the end of this list.

public E get(int index) - Returns the element at the specified position in this list.

public E getLast( ) - Returns the last element in this list.

public int size() - Returns the number of elements in this list.

Point class:

As discussed above, the Point class has two int data members, x and y. These are public and can be accessed directly, for example if we have an instance of Point:

Point p = new Point(4, 7);

The values can be accessed using p.x (which in this case would be 4) and p.y (which would be 7).

**Example Runs (User input underlined)**

**Example 1:** *Below the input is representative of the example discussed above.*

Enter number of vertices: 6

Enter edges for each Vertex, destination Vertex then weight

Enter an invalid Vertex to continue to next Vertex

Example: -1 -1

Edges for Vertex 0

Edge for Vertex 0: 1 3

Edge for Vertex 0: 3 2

Edge for Vertex 0: 4 5

Edge for Vertex 0: -1 -1

Edges for Vertex 0 entered

Edges for Vertex 1

Edge for Vertex 1: 0 3

Edge for Vertex 1: 2 4

Edge for Vertex 1: -1 -1

Edges for Vertex 1 entered

Edges for Vertex 2

Edge for Vertex 2: 1 4

Edge for Vertex 2: 3 5

Edge for Vertex 2: -1 -1

Edges for Vertex 2 entered

Edges for Vertex 3

Edge for Vertex 3: 0 2

Edge for Vertex 3: 2 5

Edge for Vertex 3: 5 8

Edge for Vertex 3: -1 -1

Edges for Vertex 3 entered

Edges for Vertex 4

Edge for Vertex 4: 0 5

Edge for Vertex 4: 5 6

Edge for Vertex 4: -1 -1

Edges for Vertex 4 entered

Edges for Vertex 5

Edge for Vertex 5: 3 8

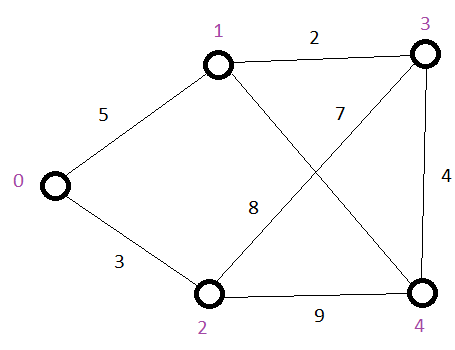
Edge for Vertex 5: 4 6

Edge for Vertex 5: -1 -1

Edges for Vertex 5 entered

Shortest path from first to last Vertex is 10

**Example 2:** *This example represents the following Graph*



Enter number of vertices: 5

Enter edges for each Vertex, destination Vertex then weight

Enter an invalid Vertex to continue to next Vertex

Example: -1 -1

Edges for Vertex 0

Edge for Vertex 0: 1 5

Edge for Vertex 0: 2 3

Edge for Vertex 0: -1 -1

Edges for Vertex 0 entered

Edges for Vertex 1

Edge for Vertex 1: 0 5

Edge for Vertex 1: 3 2

Edge for Vertex 1: 4 7

Edge for Vertex 1: -1 -1

Edges for Vertex 1 entered

Edges for Vertex 2

Edge for Vertex 2: 0 3

Edge for Vertex 2: 3 8

Edge for Vertex 2: 4 9

Edge for Vertex 2: -1 -1

Edges for Vertex 2 entered

Edges for Vertex 3

Edge for Vertex 3: 1 2

Edge for Vertex 3: 2 8

Edge for Vertex 3: 4 4

Edge for Vertex 3: -1 -1

Edges for Vertex 3 entered

Edges for Vertex 4

Edge for Vertex 4: 1 7

Edge for Vertex 4: 2 9

Edge for Vertex 4: 3 4

Edge for Vertex 4: -1 -1

Edges for Vertex 4 entered

Shortest path from first to last Vertex is 11

With all of the input it may be easiest to utilize unix’s file redirection. You can create a text file that has all of the values the user would enter and redirect it as the program’s input. For the above example if you create a text file with the following contents:

5

1 5

2 3

-1 -1

0 5

3 2

4 7

-1 -1

0 3

3 8

4 9

-1 -1

1 2

2 8

4 4

-1 -1

1 7

2 9

3 4

-1 -1

Let’s say the name of this file is Graph1.txt. If you execute the program like this:

java DijkstraDriver < Graph1.txt

The program will read those values from the input stream instead of prompting you. This makes testing go quite a bit easier.

**Assignment Questions**

Provide the answers to these questions in your submission directory within a file called Assignment5Questions.

1. What is the advantage of using LinkedLists instead of arrays for the edge and vertices list?

2. It is possible for Graphs to have negative edge weights, would Dijkstra’s Algorithm still work with such Graphs?

3. What would a potential use for Dijkstra’s Algorithm be?

3. Create a test case Graph of your own. You Graph must have at least 6 vertices and multiple possible paths to the destination node. In your submission directory include an image of your graph as well as a test case text file that can be redirected as discussed immediately above. What was the shortest path of your Graph?