Project 2 - MapQuest Routing Problem

Initial submission due: Sunday, April 29th
Final project due : Thursday, May 17th

Overview

In this project, you will write a program that computes the shortest path between two cities in the United States.

Objectives

1. Implement a graph using Java.
2. Develop a class that implements a given interface.
3. Work with graph algorithms.
4. Gain further experience working with data structures in Java.
5. Obtain additional experience writing programs in Java.
6. Gain more experience writing Java classes.

Activities

1. Review the definition of a graph and the shortest path algorithm.
2. Read the program requirements.
3. Understand the program specifications.
4. Study the program design.
5. Get the files you need to complete the project.
6. Write your code.
7. Submit your project.

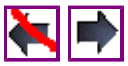
Getting Help

You may get help from your instructor(s) and the teaching assistants. Anything else is not allowed and is subject to the penalties listed in the DCS Policy on Academic Dishonesty. This includes but is not limited to:

- obtaining detailed help from any other people
- providing detailed help to other people
- sharing source code with anyone, by any means or medium.

We certainly do not expect there to be absolutely no communication between the students of this class; we know that people tend to learn very well that way. However, we expect that you will limit your discussion of this project to determining the specification of the problem. If you have any doubt if what

you would like to do might violate our rules, please see your lecture instructor for clarification.



Graphs

Background Information

In mathematics, a *graph* is a collection of *nodes* (or *vertices*) and *edges*; each node is connected to zero or more other nodes by means of edges, and each edge either connects exactly two nodes together or connects one node to itself. A collection of cities (nodes) interconnected by a network of roads (edges) resembles a graph, except that graph edges do not intersect. Figure 1 is a simple graph of four nodes, labeled A through D, and six edges, which are shown as lines connecting the nodes. Note that while two of the lines cross in the figure, they actually do not intersect. This is an undirected graph - information may pass along an edge in either direction. If we want the graph to contain directed edges, there would be arrows on the edges indicating the flow of information.

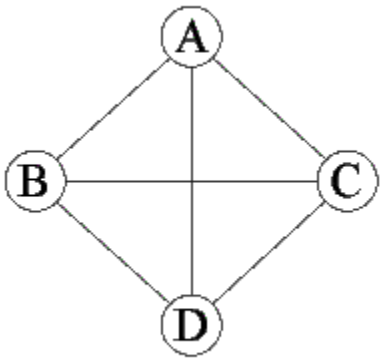


Figure 1 - An Undirected Graph

Graphs are different from trees in that trees may not contain *cycles* whereas graphs can. A cycle is a circular path that starts at a node, goes through one or more edges, and returns to the same node, without using any edge more than once. The path A-B-C-D-A shows a cycle in figure 1 above.

Often weights are associated with each of the edges in a graph. The weight might represent time, distance, cost, etc. When weights are associated with the edges, the graph is called a *weighted graph.*

Math class is now over; let's look at some applications. A great many problems are naturally formulated in terms of objects and connections between them. For example, given an airline route map of the Eastern U.S., we might be interested in questions like: "What is the fastest way to get from Rochester to Providence?" Or we might be more interested in money than in time and look for the cheapest way to get from Rochester to Providence. To answer such questions we only need information about interconnections (airline routes) between objects (towns).

The least cost path problem is to find the path in a weighted graph connecting two given nodes *x* and *y*

with the property that the sum of the weights of all the edges is minimized over all such paths. If all the weights are 1, then the problem is to find the path containing the minimum number of edges that connects x and y. In general, the path from x to y could touch all the nodes, so we usually consider the problem of finding the least cost path from a given node *x* to another node *y*, as the problem of finding the least cost path from *x* to each of the other nodes in the graph.

If you were a salesperson that needed to visit a different city every day - always starting from the home office, you could determine the shortest path to any one of the given cities using the least cost path algorithm.

Computing the Least Cost Path

Djikstra's algorithm (named after its discover, E.W. Dijkstra) can be used to compute the least cost path in a graph. The algorithm assumes that the edges in the graph are labeled with their cost (in dollars, speed, capacity, or whatever).

The algorithm starts at the source node and explores possible paths, one hop at a time, always looking at the shortest possible path so far. There is no wasted effort analyzing separate paths that have some sub-path in common, nor is any time wasted on paths that are obviously too long.

We will be assigning labels to nodes and changing them from time to time. When a node is made *permanent*, its label will not change again. The algorithm is as follows:

1. Make the source node permanent; the source node is the first working node.
2. Examine each non-permanent node adjacent to the working node. If it is not labeled, label it with the distance from the source and the name of the working node. Note that the distance you label the node with may not be the shortest path from the source node. It is simply the best distance the algorithm has discovered to this point. If it is labeled, see if the cost computed using the working node is cheaper than the cost in the label; if so, relabel the node as above.
3. Find the non-permanent node with the smallest label, and make it permanent. When all nodes have been labeled permanent, the algorithm is complete. Otherwise, this node becomes the new working node, and the algorithm continues from step 2.

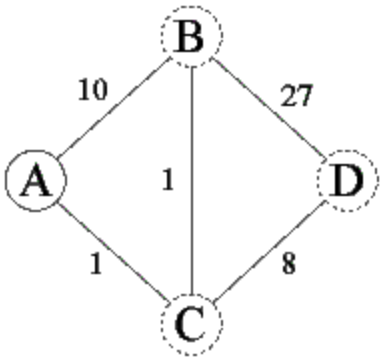
NOTE: if there is a tie - two or more nodes have the smallest label - the algorithm can choose either node.

When the algorithm is complete, the path is found (in reverse) by reading the labels from the destination node back to the source.

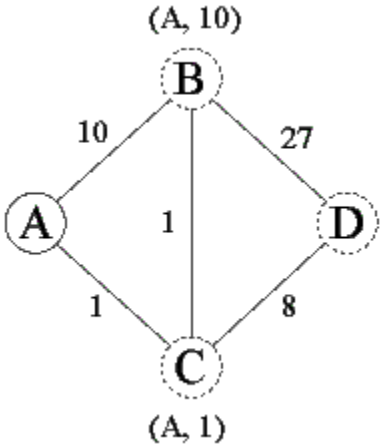
A Simple Example

We will now trace the algorithm using the initial graph shown below. The source node is A; the destination node is D.

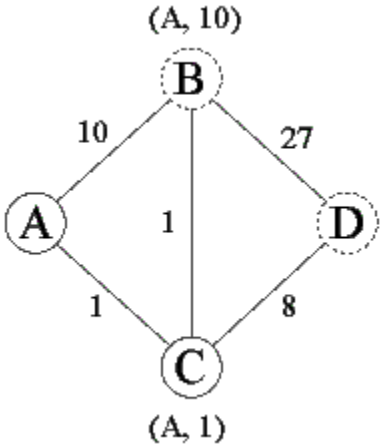
(a) Using step 1: the source node is A, make it permanent (non-permanent nodes are indicated with dashed-line circles; permanent nodes are solid line circles)



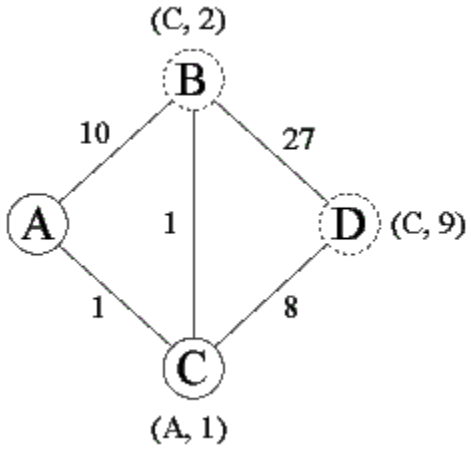
(b) Using step 2: examine each of the non-permanent adjacent nodes to the working node (A). Since neither of them are labeled, label both of them with the working node and the distance from the source node. B gets the label (A, 10) and C gets the label (A, 1). The graph now looks like:



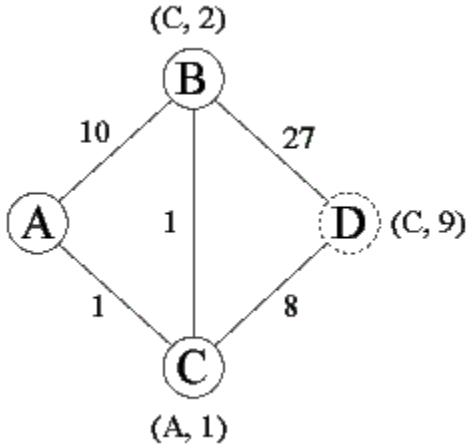
(c) Using step 3: find the non-permanent node with the smallest label and mark it permanent - C. The graph now looks like:



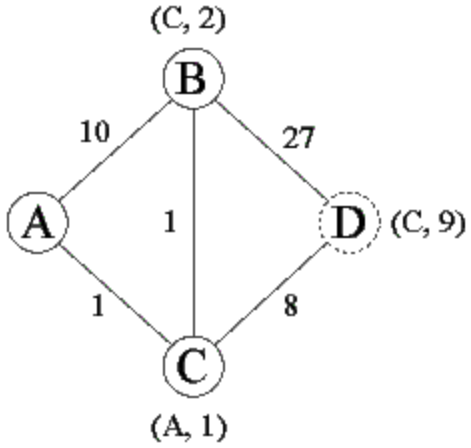
(d) Using step 2: examine the non-permanent nodes adjacent to C. The cost to get to node B from the working node C is cheaper than the cost in the label, so relabel B. Label D with the cost to get there through C. The result is:



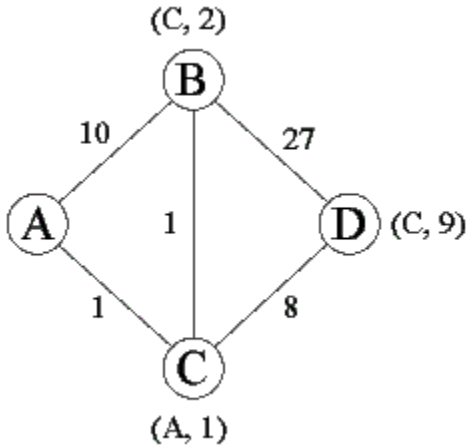
(e) Using step 3: determine the non-permanent labeled node with the smallest cost - B. Mark it permanent. The result is:



(f) Using step 2: examine the non-permanent nodes adjacent to B. The cost to get to D through B is more expensive than the cost in the label so no change is made. The result is:



(g) Using step 3: find the non-permanent node with the smallest label and mark it permanent - D. The result is:



All nodes are now labeled permanent and the algorithm is complete.

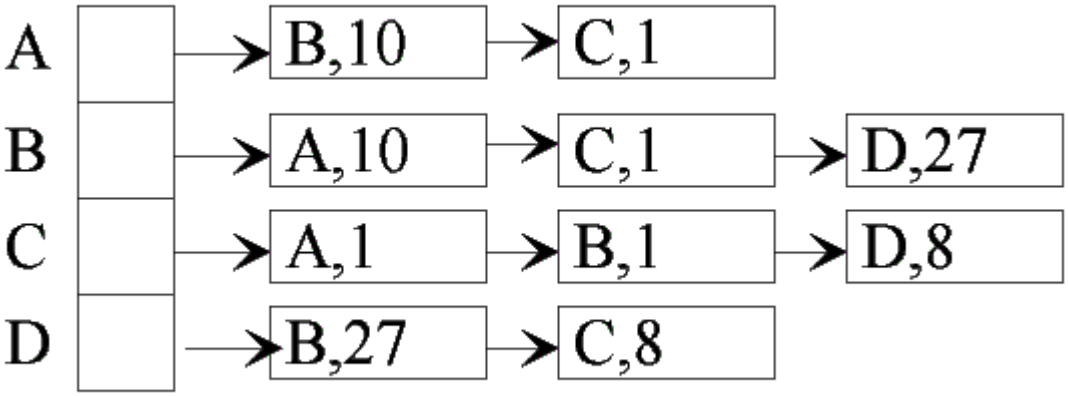
Implementation

One simple way to represent a graph is to use a two-dimensional array. This is known as an adjacency matrix representation. The value stored at each position in the matrix is the cost of the edge going from the first index into the array to the second index into the array for that position. The figure below is the adjacency matrix for the graph used in the example above.

	A	B	C	D
A		10	1	
B	10		1	27
C	1	1		8
D		27	8	

If we consider the indices representing the rows as a source city and the indices representing the columns as a destination city, (these are all direct connections) then when representing an undirected graph, each entry must be made in two places - from the source city to the destination city and also from the destination city (used as a source) to the source city (used as a destination). This results in a graph that contains duplicate entries above and below the diagonal of the matrix. If many of the nodes do not have direct connections, then the matrix is considered sparse and there is a lot of wasted space.

An alternative method of implementing graphs is the adjacency list representation. An adjacency list representation of a graph consists of an array of lists, one list for each node in the graph. For each node *v*, the adjacency list consists of all the nodes that are adjacent to *v*. The figure below is an adjacency list representation for the sample input. The adjacency list representation contains only the connections that actually exist, although with an undirected graph they all occur in two adjacency lists - once for each direction. Note that there are many different adjacency list representations for the same graph, depending on the order of the nodes and the edges.



Project 2 - Requirements

You are to write a program that will compute the shortest path between two cities in the United States. The program will be given the cities for which the path is to be computed, and a file that contains segments of the United States Interstate system.

An interstate segment represents a direct link between two cities in the United States. For example there is a segment between Rochester NY and Buffalo NY. The road that links these two cities is I90 and the distance of the segment is about 73 miles. Each segment is identified by the two cities in the segment, the name of the interstate connecting the cities, and the distance between them. Segments are bi-directional.

The output from the program will consist of the segments that make up the shortest path between the first city and the destination city. For each segment it will print the cities in the segment, the name of the interstate connecting the cities, and the length of the segment. The output will also identify the cities for which the path was found and the total distance of the path. The following output was generated for the path: Rochester to Albany:

```
Rochester NY to Albany NY: 206 miles.
    Rochester NY to Syracuse NY via I90 86 miles.
    Syracuse NY to Albany NY via I90 120 miles.
```

When the program detects errors in usage, in processing the input, or a path cannot be found, an appropriate error message will be generated and the program will terminate.

The program must provide a summary mode for output. When run in summary mode the program will only report the distance between the source city and the destination.

Project 2 - Specifications

Overview

The command line arguments to your program will consist of the name of the file containing the interstate segments, the name of the source city, and the name of a destination city. The program will compute the shortest path from the source city to the destination city using Dijkstra's shortest path algorithm .

Dijsktra's algorithm will find a least cost path in a graph, however, it does not specify what to do if there is more than one shortest path. To break ties, your program will take into consideration the alphabetical ordering of the city names. If the distance in a label is the same as the distance of the current path, you will assume that the city with the earlier name in the alphabet has the shortest distance.

City names consist of both the name of the city and the abbreviation for the state in which the city is located. For example, Rochester, New York would be identified by the name "Rochester NY".

Starting the Program

The program is started from the command line as shown below:

```
java MapQuest [-s] database source-city dest-city
```

where `MapQuest` is the name of the main program, `database` is the name of the file that contains the interstate segment information, `source-city` is the name of the city from which the path originates, and `dest-city` represents the name of the destination city. The command line option `-s`, if present, specifies that only summary information for each path is to be reported.

For example, the following command:

```
java MapQuest roads.txt "Syracuse NY" "Buffalo NY"
```

specifies that the program should compute the path from Syracuse to Buffalo. Note the use of the quotation marks in the city names. The quotes tell the operating system to treat the characters inside as a single string (you will not see the quotes in your java program).

If the program is invoked with the wrong number of command line arguments, an invalid option, or with the option in the incorrect location the following error message will be printed to standard error and the program will terminate without generating any other output:

```
Usage:  java MapQuest [-s] database source-city dest-city
```

If the the file containing the interstate segment information cannot be found or cannot be opened, the

following message will be printed to standard error and the program will terminate without generating any other output:

```
MapQuest:  cannot open filename
```

where you will substitute the name of the file that could not be opened.

Both the city names specified on the command line will be checked in the order they were typed before the shortest path computation begins. The following message will be printed to standard error for the first city name that is not valid, if any, and the program will terminate without producing any other output:

```
MapQuest:  invalid city city-name
```

where you will substitute the name of the invalid city for *city-name*.

File Formats

The interstate segment file is a plain text file. Each line in the file has the following format and defines a single segment (click here to see a sample segment file):

```
source-city destination-city route-name length-of-segment
```

Your program may assume that the segment file is in the correct format. Each line in the file represents a segment. A segment is defined by a source city, a destination city, a road name, and a distance. Note that since cities may consist of several words, the city names appear within quotation marks in the file. A sample segment line is shown below:

```
"Boston MA" "New York City NY" I95 180
```

The line above defines a segment from Boston to New York City. The route name is I95 and the distance is 180 miles. Note that segments are bi-directional. The line above defines a link between Boston and New York, *and* a link between New York and Boston.

During the execution of the program, if an IO error should occur while reading the segment file, the following message will be printed to standard error and the program will terminate without generating any other output:

```
MapQuest:  I/O error
```

Output

The output from your program, which will be printed to standard output, will list the shortest path from the source city to the destination city.

The output generated when invoking the program using the following command line:

```
java MapQuest roads.txt "Rochester NY" "Albany NY"
```

is shown below:

```
Rochester NY to Albany NY: 206 miles.
    Rochester NY to Syracuse NY via I90 86 miles.
    Syracuse NY to Albany NY via I90 120 miles.
```

When the program is run in summary mode, by specifying the `-s` option on the command line, only the summary information for the path will be reported.

The output generated when invoking the program using the following command line:

```
java MapQuest -s roads.txt "Rochester NY" "Albany NY"
```

is shown below:

```
Rochester NY to Albany NY: 206 miles.
```

Project 2 - Design

Overview

The shortest path program submitted at the first milestone consists of 2 main classes: `ArrayDiGraph` (which is supplied to you), and `MapQuest` (which you will write).

The `MapQuest` class is the main class that drives the program. It is responsible for processing the command line arguments, creating a graph that contains the information from the segment file, finding the shortest path between the specified cities, and creating the output. You may create additional classes that will be used by the `MapQuest` class.

Your program ***must*** use the shortest path algorithm that is described in the definition of a graph section of this document.

The `CS3DiGraph` class submitted at the **second** milestone is an implementation of the `DiGraph` interface provided in the jar file for this assignment.

Again you should feel free to create additional classes that will be used by your graph implementation. Although it is not mentioned in the interface, you might want to write a `toString()` method for the `CS3Graph` class. The `toString()` method should be written so that it returns a string that contains the data contained in the graph in an understandable form. You might find the `toString()` method useful when you are trying to debug your code and you would like to be able to view the contents of a graph.

Your program ***must*** implement the `CS3DiGraph` class using either the **adjacency matrix** or **adjacency list** representation specified in the definition of a graph section of this document.

Project 2 - Getting the Files You Need

The files that you need to complete this project are contained in the `jar` file located here. Download the `jar` file and unpack it in an appropriate location in your account.

After you have unpacked the jar file you should see that a directory named `project2` has been created. There are two directories contained in `project2`: `RCS`, and `data`. The directory `project2/RCS` contains the Java source code for the `DiGraph` interface, and the `NoSuchVertexException` class. (Do not change these files.)

The jar file also contains a sample implementation of the `DiGraph` interface, named `ArrayDiGraph`, which you will use in writing the `MapQuest` program for the first submission.

The directory, `project2/data`, contains a file named `roads.txt`, that contains a number of segments from the United States Interstate System. This is the file that will be used when we test your program. You might find it helpful to create shorter segment files during the initial testing stages of your program. You will also find some files that have names like "RochesterTampa", "SyracuseBuffalo", or "RochesterDC" that contain the output produced by a working version of the program specifying the cities in the file name as input to the program. When you think you have a working version of the program, you may want to compare the output produced by your program to the contents of these files. Remember that in order for your output to be considered ***correct*** it must match the output produced by our program ***exactly***.

Project 2 - Writing Your Code

DO NOT write any code until you understand the behavior provided by each of the methods in the `DiGraph` interface, the shortest path algorithm, and the specification for the "MapQuest" program. Writing code before understanding the program will simply be a waste of your time, and will result in a program that does not work!!

We have designed this project to allow you to concentrate on how to use the graph class before you have to figure out how to implement it. So focus on the `DiGraph` interface first when you are working on the `MapQuest` program.

You should try your best to begin implementation of CS3DiGraph early, i.e. before part 1 is even due! Students traditionally have a very hard time getting all the bugs out of it.

The first thing you have to decide is how to implement the graph. After you have made your decision, start implementing some of the easier methods in the `DiGraph` interface first. Make sure to take advantage of the classes provided by the Java Collections Framework whenever possible.

Be sure to write a test program, and test your code as you develop it. Feel free to use the `roads.txt` file to generate input for your test program. Since the `roads.txt` file contains lots of information, you may find it useful to create different versions of the file that contain fewer segments. In this way your test program will create graphs that are not so large that you cannot determine what the graph looks like and subsequently what the output of the test program should be.

After you finish writing a method, test it to make sure it works as expected. Testing small pieces of your program often is much easier that trying to test the entire program at once. It is much easier to find a mistake in a dozen lines of code than in several hundred! You should also make sure your tests check both expected and unexpected cases. For example, when testing the `addVertex()` method be sure to try adding a new vertex to the graph and one that is already in the graph; when testing the `addEdge()` method, be sure to try adding edges using 2 valid vertices, only 1 valid vertex, and 2 invalid vertices, as well as adding edges that already exist.

Start working on the parts of this program that you feel are the easiest, and work your way to the harder ones. As you implement the individual pieces of your program be sure that the program compiles without errors. You should also test each piece of the program as it is completed. This may require that you create special test files or add code for testing the intermediate functionality.

Develop your code in small pieces, and test each part of your code as you write it. If you wait until you have all the code written to start testing, it will be almost impossible to locate and correct errors in your program. Students who start early and test their code as they develop it, are the ones that get very high grades for their projects. Avoid the trap of starting your work too late.

Part of your grade will be based on how you use RCS, so make sure that you check in your code after completing substantial changes to it.

Finally, as we always say, "submit early; submit often." That is, if your code compiles, submit it. That way you have something turned in and you will receive partial credit even if you make no further progress. (But remember that **try** keeps only the latest submission: if you turn in a working version of your code and then submit another copy that is "broken", we will only see the broken version.)

Project 2 - Project Submission and Grading

Project Submission

In order to satisfy the **initial** submission requirements for this project, you must successfully implement the main program, `MapQuest`, and submit your work using the following command:

```
try 233-grd project2-1 MapQuest.java other optional java files...
```

where you will substitute the names of any other files containing classes that need to be compiled in order to run your program for the phrase "*other optional java files...*". Note that these files ***do not*** include the ArrayDiGraph class. We will supply a copy of this class file when testing your shortest path program. The extra files are optional, it may be the case that your program is contained in a single Java source file.

You must submit a working version of the `MapQuest` program by midnight on Sunday, April 29th.

In order to satisfy the **final** submission requirements for this project, you must successfully implement a `CS3DiGraph` class. After you are convinced that your graph class is correct, submit it using the following command:

```
try 233-grd project2-2 CS3DiGraph.java other optional java files...
```

where you will substitute the names of any other files containing classes that need to be compiled in order to use your graph class for the phrase "*other optional java files...*". Note that the extra files are optional, it may be the case that your graph class is contained in a single Java source file.

The `try` command will test your graph class. If your class is accepted by try with no errors, you have satisfied the initial submission requirement for this project.

In order to help you to determine when you have a working implementation of the `CS3DiGraph` class, `TestGraph`, the program that `try` uses, has been included in the RCS directory in the `jar` file that you downloaded for this project. You will also find a file named, `TestGraph.out`, in the `data` directory in the same `jar` file. This file contains the output that the `TestGraph` program will produce when it tests a `CS3DiGraph` class that passes all of the tests. Feel free to use the `TestGraph` program to test your graph. If the output from the `TestGraph` program matches the contents of the `TestGraph.out` file, exactly, your graph passes the initial submission tests.

If you submit your `CS3DiGraph`, and `try` reports that your `CS3DiGraph` class is not correct, you can find out exactly what the errors are by running `TestGraph` yourself. If you compare the output that is generated by `TestGraph` when you run it, with the contents of the `TestGraph.out` file, you should be able to determine where your program is failing.

Your complete project is due by midnight Thursday, May 17th.

When you feel that you have reached an important milestone in the program's development, *submit* it. This way if you fail to get any further before the due date, at least you will have submitted something.

Remember to submit early, and often. The labs get *very* busy the night a project is due! Although you may start to run `try` before midnight, it may not finish before midnight. The best strategy is to finish before the labs get crazy; then you can sit back and watch the others panic.

Project Grade

The project grade will be computed as follows:

1. You cannot get a high grade for a program that does not run. Therefore, your grade is first computed *solely* on the correctness of your output. You can get up to 100 points for this. After this preliminary grade is computed, ...
2. You can lose up to 35 points if your algorithms and/or implementation are not clear or of low quality.
3. You can lose up to 30 points for violating the programming style standards. This includes incorrect use of RCS.

Note that a program that produces wrong answers will get a low grade no matter how good its algorithms are or how well it adheres to the style standards.

Interface DiGraph<VertexKey,VertexData,EdgeData>

public interface **DiGraph<VertexKey,VertexData,EdgeData>**

An interface for a directed graph. Every vertex in the graph is identified by a unique key. Data can be stored at each edge and vertex in the graph.

Method Summary

void	**addEdge**(VertexKey fromKey, VertexKey toKey, EdgeData data) Add an edge to the graph starting at the vertex identified by fromKey and ending at the vertex identified by toKey.
void	**addVertex**(VertexKey key, VertexData data) Add a vertex to the graph.
void	**clear**() Remove all vertices and edges from the graph.
java.util.Collection<EdgeData>	**edgeData**() Return a collection containing all of the data associated with the edges in the graph.
EdgeData	**getEdgeData**(VertexKey fromKey, VertexKey toKey) Return a reference to the data associated with the edge that is defined by the given end points.
VertexData	**getVertexData**(VertexKey key) Return a reference to the data associated with the vertex identified by the key.
int	**inDegree**(VertexKey key) Return the in degree of the vertex that is associated with the given key.
boolean	**isEdge**(VertexKey fromKey, VertexKey toKey) Return true if the edge defined by the given vertices is an edge in the graph.
boolean	**isVertex**(VertexKey key) Returns true if the graph contains a vertex with the associated key.
java.util.Collection<VertexData>	**neighborData**(VertexKey key) Returns a collection containing the data associated with the

	neighbors of the vertex identified by the specified key.
java.util.Collection<VertexKey>	**neighborKeys**(VertexKey key) Returns a collection containing the keys associated with the neighbors of the vertex identified by the specified key.
int	**numVertices**() Returns a count of the number of vertices in the graph.
int	**outDegree**(VertexKey key) Return the out degree of the vertex that is associated with the given key.
java.util.Collection<VertexData>	**vertexData**() Returns a collection containing the data associated with all of the vertices in the graph.
java.util.Collection<VertexKey>	**vertexKeys**() Returns a collection containing the keys associated with all of the vertices in the graph.

Method Detail

addVertex

```
void addVertex(VertexKey key,
               VertexData data)
```

Add a vertex to the graph. If the graph already contains a vertex with the given key the old data will be replaced by the new data.

Parameters:

`key` - the key that identifies the vertex.
`data` - the data to be associated with the vertex.

---

addEdge

```
void addEdge(VertexKey fromKey,
             VertexKey toKey,
             EdgeData data)
             throws NoSuchVertexException
```

Add an edge to the graph starting at the vertex identified by fromKey and ending at the vertex identified by toKey. If either of the vertices do not exist a NoSuchVertexException will be thrown. If the graph already contains this edge, the old data will be replaced by the new data.

Parameters:

`fromKey` - the key associated with the starting vertex of the edge.
`toKey` - the key associated with the ending vertex of the edge.

`data` - the data to be associated with the edge.

Throws:

`NoSuchVertexException` - if either end point is not a key associated with a vertex in the graph.

---

isEdge

```
boolean isEdge(VertexKey fromKey,
               VertexKey toKey)
               throws NoSuchVertexException
```

Return true if the edge defined by the given vertices is an edge in the graph. False will be returned if the edge is not in the graph. A NoSuchVertexException will be thrown if either of the vertices do not exist.

Parameters:

`fromKey` - the key of the vetex where the edge starts.

`toKey` - the key of the vertex where the edge ends.

Returns:

true if the edge defined by the given vertices is in the graph and false otherwise.

Throws:

`NoSuchVertexException` - if either end point is not a key associated with a vertex in the graph.

---

getEdgeData

```
EdgeData getEdgeData(VertexKey fromKey,
                     VertexKey toKey)
                     throws NoSuchVertexException
```

Return a reference to the data associated with the edge that is defined by the given end points. Null will be returned if the edge is not in the graph. Note that a return value of null does not necessarily imply that the edge is not in the graph. It may be the case that the data associated with the edge is null. A NoSuchVertexException will be thrown if either of the end points do not exist.

Parameters:

`fromKey` - the key of the vertex where the edge starts.

`toKey` - the key of the vertex where the edge ends.

Returns:

a reference to the data associated with the edge defined by the specified end points. Null is returned if the edge is not in the graph.

Throws:

`NoSuchVertexException` - if either end point is not a key associated with a vertex in the graph.

---

isVertex

```
boolean isVertex(VertexKey key)
```

Returns true if the graph contains a vertex with the associated key.

Parameters:
`key` - the key of the vertex being looked for.
Returns:
true if the key is associated with a vertex in the graph and false otherwise.

---

getVertexData

```
VertexData getVertexData(VertexKey key)
                         throws NoSuchVertexException
```

Return a reference to the data associated with the vertex identified by the key. A NoSuchVertexException will be thrown if the key is not associated with a vertex in the graph.

Parameters:
`key` - the key of the vertex being looked for.
Returns:
the data associated with the vertex that is identifed by the key.
Throws:
`NoSuchVertexException` - if the key is not associated with a vertex in the graph.

---

numVertices

```
int numVertices()
```

Returns a count of the number of vertices in the graph.

Returns:
the count of the number of vertices in this graph

---

inDegree

```
int inDegree(VertexKey key)
```

Return the in degree of the vertex that is associated with the given key. Negative 1 is returned if the vertex cannot be found.

Parameters:
`key` - the key of the vertex being looked for.

Returns:
the in degree of the vertex associated with the key or -1 if the vertex is not in the graph.

---

outDegree

```
int outDegree(VertexKey key)
```

Return the out degree of the vertex that is associated with the given key. Negative 1 is returned if the vertex cannot be found.

Parameters:
`key` - the key of the vertex being looked for.
Returns:
the out degree of the vertex associated with the key or -1 if the vertex is not in the graph.

---

neighborData

```
java.util.Collection<VertexData> neighborData(VertexKey key)
                                              throws NoSuchVertexException
```

Returns a collection containing the data associated with the neighbors of the vertex identified by the specified key. The collection will be empty if there are no neighbors. A NoSuchVertexException will be thrown if the key is not associated with a vertex in the graph.

Parameters:
`key` - the key associated with the vertex whose neighbors we wish to obtain.
Returns:
a collection containing the data associated with the neighbors of the vertex with the given key. The collection will be empty if the vertex does not have any neighbors.
Throws:
`NoSuchVertexException` - if the key is not associated with a vertex in the graph.

---

neighborKeys

```
java.util.Collection<VertexKey> neighborKeys(VertexKey key)
                                             throws NoSuchVertexException
```

Returns a collection containing the keys associated with the neighbors of the vertex identified by the specified key. The collection will be empty if there are no neighbors. A NoSuchVertexException will be thrown if the key is not associated with a vertex in the graph.

Parameters:
`key` - the key associated with the vertex whose neighbors we wish to obtain.
Returns:

a collection containing the keys associated with the neighbors of the vertex with the given key. The collection will be empty if the vertex does not have any neighbors.

Throws:

`NoSuchVertexException` - if the key is not associated with a vertex in the graph.

---

vertexData

`java.util.Collection<VertexData> **vertexData**()`

Returns a collection containing the data associated with all of the vertices in the graph.

Returns:

a collection containing the data associated with the vertices in the graph.

---

vertexKeys

`java.util.Collection<VertexKey> **vertexKeys**()`

Returns a collection containing the keys associated with all of the vertices in the graph.

Returns:

a collection containing the keys associated with the vertices in the graph.

---

edgeData

`java.util.Collection<EdgeData> **edgeData**()`

Return a collection containing all of the data associated with the edges in the graph.

Returns:

a collection containing the data associated with the edges in this graph.

---

clear

`void **clear**()`

Remove all vertices and edges from the graph.

Class NoSuchVertexException

```
java.lang.Object
  |
  +--java.lang.Throwable
        |
        +--java.lang.Exception
              |
              +--NoSuchVertexException
```

All Implemented Interfaces:
java.io.Serializable

---

public class **NoSuchVertexException**
extends java.lang.Exception

An exception thrown by a class that implements the DiGraph interface to indicate a problem with the parameters passed to a method. Note that there is no way to create a NoVertexException that does not contain a message.

See Also:
Serialized Form

Constructor Summary

NoSuchVertexException(java.lang.String msg) Create a new NoSuchVertexException that contains the specified message.

Methods inherited from class java.lang.Throwable
fillInStackTrace, getCause, getLocalizedMessage, getMessage, getStackTrace, initCause, printStackTrace, printStackTrace, printStackTrace, setStackTrace, toString

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Constructor Detail

NoSuchVertexException

```
public NoSuchVertexException(java.lang.String msg)
```

Create a new NoSuchVertexException that contains the specified message.

Parameters:

`msg` - the message to be placed in the exception.
