**22C:21 Project 2**  
Due date and time: See submission schedule below.

**Introduction.** In this project you will implement several versions of a class that implements the graph data structure and write a program that tests and evaluates this class. A graph is a structure that consists of vertices, pairs of which are connected by edges. For example, here is a graph that might represent flight connections among a few American cities. The nodes (cities) are shown as points and the edges connecting pairs of cities (flight connections) are shown as straight line segments.

![Graph diagram](image)

A graph can model all kinds of real-life structures. In addition to transport connections, graphs are used extensively to model circuits etched on a computer chip, organization of large corporations, networks of computers, the world wide web, evolution of species, states in games of strategy, etc. Often, a graph allows us to abstract the essential features of the underlying structure and to reason about the structure more precisely. Chapter 15 in your textbook deals with graphs, but you don’t have to read the chapter to complete this project.

**The MyGraph interface.** In this project, you will start with an interface called MyGraph (to avoid confusion with the Graph interface provided by Bailey), create an abstract class called MyAbstractGraph that implements MyGraph, and then create two regular classes, MyMatrixGraph and MyListGraph that extend MyAbstractGraph.

The MyGraph interface is given below.

```java
public interface MyGraph {
    public void addVertex(Object v);
    public void addEdge(Object u, Object v);
    public void deleteVertex(Object v);
    public void deleteEdge(Object u, Object v);
}
```
public Vector getVertices();
public Matrix getEdges();
public Vector getNeighbors(Object v);

public int numberOfVertices();
public int numberOfEdges();

public boolean areNeighbors(Object u, Object v);
public Vector depthFirstTraversal();

}  

With the exception of the last function, the names of the functions in the above interface are fairly self-explanatory. Anyway, here is a brief description of each of the functions mentioned above.

void addVertex(Object v) Add a vertex v to this graph. If v already exists in G, then this function does nothing.

void addEdge(Object u, Object v) Add an edge between vertex u and vertex v to this graph. If there is already an edge between u and v, then this function does nothing. If either of the vertices u and v do not exist in this graph, then the function throws an exception.

void deleteVertex(Object v) Delete the vertex v from this graph. If v does not exist in this graph, then this function does nothing.

void deleteEdge(Object u, Object v) Delete the edge between u and v from this graph. If this edge does not exist in this graph then the function does nothing. If either of the vertices u and v do not exist in this graph, then the function throws an exception.

Vector getVertices() Return a Vector of vertices of this graph, in no particular order.

Matrix getEdges() Return a Matrix of edges of this graph, in no particular order. The returned Matrix has two rows and as many columns as the number of edges of the graph. Each column of the Matrix contains the two end-vertices of an edge.

Vector getNeighbors(Object v) Return a Vector of neighboring vertices of the vertex v, in no particular order.

int numberOfVertices() Return the number of vertices in this graph.

int numberOfEdges() Return the number of edges in this graph.

boolean areNeighbors(Object u, Object v) Return true if there is an edge between vertices u and v in the graph. Return false otherwise. Throw an exception if either u and v does not exist in this graph.

Vector depthFirstTraversal() Return a Vector of objects of type SinglyLinkedListElement that represents a depth-first traversal of the graph. More details in class.
The MyMatrixGraph and the MyListGraph classes. Here I describe two alternate graph representations, the adjacency matrix representation and the adjacency list representation. You are required to use the adjacency matrix representation in the MyMatrixGraph class and the adjacency list representation in the MyListGraph class. For this discussion, let us suppose that we have a graph with $n$ vertices. Both representations consist of a Vector called map that contains the $n$ vertices. The vector map should be viewed as an assignment of distinct IDs in the range 0 through $n - 1$ to the $n$ vertices. More specifically, it should be your view that the vertex stored in slot map[i] has ID i.

In addition to map, the adjacency matrix representation consists of a $n \times n$ boolean matrix, let us call this $M$, such that $M[i,j] = 1$ if there is an edge between the vertex with ID $i$ and the vertex with ID $j$; otherwise, if there is no edge between the vertex with ID $i$ and the vertex with ID $j$, then $M[i,j] = 0$. In this project, we are only interested in undirected graphs and so $M$ is symmetric; that is, $M[i,j] = M[j,i]$.

In addition to map, the adjacency list representation consists of a Vector of size $n$, let us call this $L$, each of whose elements is a DoublyLinkedList object. The DoublyLinkedList object stored at $L[i]$ contains all the neighbors of the vertex with ID $i$. Note that if vertices with IDs $i$ and $j$ are adjacent to each other, then $i$ appears in the doubly linked list at $L[j]$ and $j$ appears in the doubly linked list at $L[i]$.

The MyMatrixGraph abstract class. The function depthFirstTraversal that returns a depth-first traversal of the graph, can be implemented in terms of the rest of the functions in MyGraph interface. Hence, this function should be implemented in the MyAbstractGraph class and should be inherited by the MyMatrixGraph class and the MyListGraph class. depthFirstTraversal can be implemented by a recursive algorithm, which will be discussed in class.

Testing your implementation. As a test of your implementation, I want you to build a program that plays the Ladders game. In this two-player game, one player chooses a starting word and an ending word and the other player constructs a “ladder” between the two words. A ladder is a sequence of words that starts at the starting word, ends at the ending word, and each word in the sequence (except the first) is obtained from the previous word by changing a letter in a single position. For example, suppose the starting word is flour and the ending word is bread, then a ladder between these two words is: flour, floor, flood, bread.

On the course page you will find a link to a file called words.dat that contains 5757 five letter English words. This word list comes from Stanford Graphbase, a collection of interesting data sets and graph algorithms put together by Donald E. Knuth. The original file can be found at

ftp://labrea.stanford.edu/pub/sgb

This word list is the database that your program will use to play the Ladders game.

In this version of the game, the user is always the one providing a pair of words and your program is always the one constructing a ladder between a given pair of words. Your
program should start by constructing a graph whose vertices are the five letter words in \texttt{words.dat}. There is an edge between a pair of five letter words if one can be obtained from the other by changing a letter in exactly one position. We will call this the \textit{ladders graph} on 5-letter words. After this graph has been constructed, your program should repeatedly prompt the user to enter a pair of 5-letter words. When the user responds by entering two 5-letter words, your program will either (i) output a ladder between the two words or (ii) output a message saying there is no ladder between the two words. Your prompts to the user should be clear and should provide the user the option of quitting your program whenever they want.

**Submission Schedule.** Your submission is broken up into three parts.

**Wednesday, Nov 3rd, 5pm** Submit the files \texttt{MyGraph.java} and \texttt{MyMatrixGraph.java}. The class \texttt{MyMatrixGraph} should implement the interface \texttt{MyGraph}. \texttt{MyMatrixGraph} need not implement \texttt{depthFirstTraversal} and therefore can contain just an empty implementation of this function. Create a directory called \texttt{project2.1} with these two files in it and submit this directory. This submission is worth 30 points. Our solution will be released soon after the submission deadline.

**Monday, Nov 8th** Submit the files \texttt{MyGraph.java} and \texttt{MyListGraph.java}. The class \texttt{MyListGraph} should implement the interface \texttt{MyGraph}. \texttt{MyListGraph} need not implement \texttt{depthFirstTraversal} and therefore can contain just an empty implementation of this function. Create a directory called \texttt{project2.2} with these two files in it and submit this directory. This submission is worth 30 points. Our solution will be released soon after the submission deadline.

**Monday, Nov 15th** Submit the following files: \texttt{MyGraph.java}, \texttt{MyAbstractGraph.java}, \texttt{MyMatrixGraph.java}, \texttt{MyListGraph.java}, and \texttt{Ladders.java}. The \texttt{MyAbstractGraph} abstract class should implement the \texttt{MyGraph} interface and should contain the implementation of \texttt{depthFirstTraversal}. The classes \texttt{MyMatrixGraph} and \texttt{MyListGraph} should extend \texttt{MyAbstractGraph}. This submission is worth 40 points.