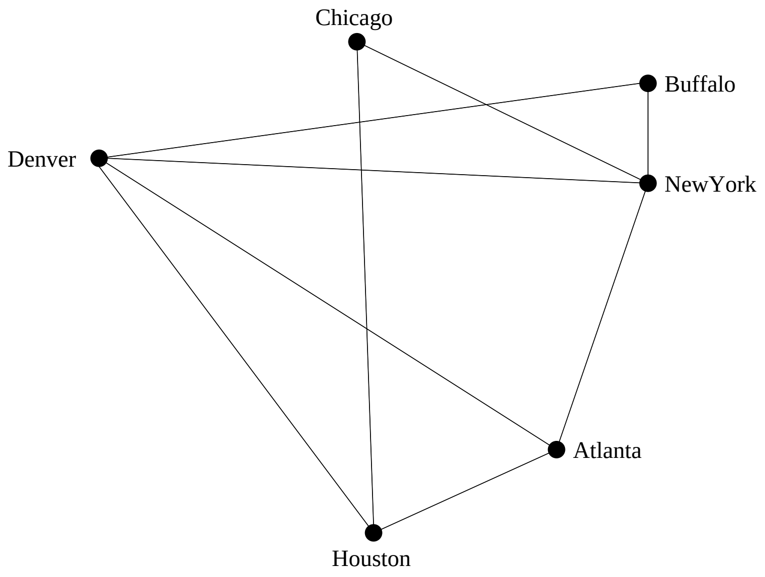
Computer Science III (22C:30, 22C:115)
Project 1, Due: 10/7/02, 5 pm

1. Introduction.

In this programming project you will implement a `graph` class and test it in different ways.

2. What is a graph?

A *graph* is a structure that consists of *nodes*, 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.



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.

3. The `graph` class: member functions.

Your implementation of the `graph` class should support the following operations:

1. Add a node v to the graph G. If v already exists in G, then this operation produces a warning message; otherwise it adds the new node v to G.
2. Add an edge e to the graph G. If e already exists in G, then this operation produces a warning message ; otherwise it adds the new edge e to G. If either of the end-nodes of e do not exist in G, then this operation produces a warning message.
3. Delete a node v from the graph G. If v does not exist in G then this operation produces a warning message.
4. Delete an edge e from the graph G. If e does not exist in G then this operation produces a warning message.

5. Return all the nodes *adjacent* to a node v. Two nodes are said to be adjacent if they are connected by an edge. This operation produces a warning if v does not exist in the graph; otherwise it returns the list of all nodes adjacent to v.

The first four of the above five operations should be implemented by overloading the operators `+=` and the operator `-=`. In particular, the operator `+=` should be overloaded so that the statement

```
G += x;
```

adds a node `x` or an edge `x` to the graph `G` depending on whether `x` is a node or an edge. The operator `-=` should be overloaded in a similar manner. In addition, you should provide a member function called `get_neighbors` that returns a vector of nodes adjacent to a given node. In cases in which a function has reason to produce a warning message, `G` should remain unchanged after control returns from the function.

Besides the member functions listed above, you will have to provide appropriate constructors and destructors and possibly a few other access functions for the `graph` class.

4. The `graph` class: data members.
There are several ways of representing a graph on a computer. The choice of representation will determine the private data members of the `graph` class. I want you to implement the *adjacency matrix* representation of a graph. As the name suggests, an adjacency matrix is a matrix (2-dimensional array) that stores information on which pairs of nodes are adjacent and which are not as a boolean array. In particular, the rows and the columns of an adjacency matrix represent nodes and entry (i, j) in the matrix is `True` if the nodes corresponding to row i and column j are connected by an edge; otherwise the entry (i, j) is a `False`. For example, if we assume that the 6 American cities shown in flight connection graph on the previous page correspond to the rows and columns of a 6×6 adjacency matrix in alphabetical order, then the adjacency matrix representation of this graph:

	A	B	C	D	H	N
A	F	F	F	T	T	T
B	F	F	F	T	F	T
C	F	F	F	F	T	T
D	T	T	F	F	T	T
H	T	F	T	T	F	F
N	T	T	T	T	F	F

The `T`'s and the `F`'s in the above table represent `True` and `False` entries. The first row and the first column show names of the cities. This is just for readability, this information is not actually stored in the adjacency matrix. You should use the `apvector` class posted on the course page to help in implementing a 2-dimensional boolean array.

5. Important features of your implementation.
It is required the your implementation have the following features:

- The `graph` class should be a template class, thereby allowing for different kinds of information stored at nodes. For example, if nodes represents cities, we might want to store a city name at each node as a string.
- The memory being used for a `graph` object should dynamically grow and shrink as the number of nodes in the graph increases or decreases. However, resizing after every operation is too costly. So use the following rule: to store a graph with n nodes you should use an $m \times m$ 2-dimensional boolean array, where m does not exceed $2n$. This means that when the matrix is filled to capacity it doubles in size and when it is less than half full, it shrinks to half its current size.

6. Helper classes.
To use the statement `G += x;` to add a node or an edge `x` to the graph `G`, it will be useful to have a `node` class and a `edge` class. Then, if `x` is an object belonging to the `node` class, `G += x` will call the function

that adds a node and if `x` is an object belonging to the `edge` class, `G += x` will call the function that adds an edge.

Both the `node` class and the `edge` class should be template classes, but both classes should be otherwise quite simple. For example, if in our application we want to store a string at each node, then the `node` class will contain a string data member whereas the `edge` class will contain two string data members. What member functions each of these helper classes contain depends on the needs of the rest of your program and this design decision is yours to make.

7. Testing the `graph` class implementation.
To test your implementation of the graph class write a "driver" program that reads one or more lines of input, where each line specifies an operation. The operation specified in each line is one of the following:

- `A str`
  In response your program should add to the graph a node whose name is specified by the string `str`. If there is a node with name `str` already in the graph, then your program should respond by producing an appropriate warning message.

- `A str1 str2`
  In response your program should add to the graph an edge between nodes whose names are `str1` and `str2`. If there is an edge between nodes with names `str1` and `str2` already in the graph, then your program should respond by producing an appropriate warning message. Also, if the node with name `str1` or the node with name `str2` does not exist in the graph, then your program should produce an appropriate warning message.

- `D str`
  In response your program should delete from the graph a node whose name is specified by the string `str`. If there is no node with name `str` in the graph, then your program should respond by producing an appropriate warning message.

- `D str1 str2`
  In response your program should delete from the graph an edge between nodes whose names are `str1` and `str2`. If there is no edge between nodes with name `str1` and `str2` in the graph, then your program should respond by producing an appropriate warning message. Also, if the node with name `str1` or the node with name `str2` does not exist in the graph, then your program should produce an appropriate warning message.

- `P str`
  In response, your program should print out the list of all nodes adjacent to the node with name `str`. I will leave all decisions regarding the format of the output to you.

So as to not spend too much time or effort writing code to process the input, you may assume that there are no syntax errors in the input. In other words, you may assume that each line begins with one of the letters `A`, `D`, or `P`. Furthermore, you may assume that consecutive objects in a line are separated by 1 or more blanks.

The driver program essentially consists of a loop that processes input until the end of file is reached. Each execution of the loop reads a line and calls the appropriate member function.

8. The ladders game.
As a more extensive test of your implementation of the `graph` class, I want you to build the *ladders graph*. *Ladders* is a game you may have played. In this 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`, `blood`, `brood`, `broad`, `bread`. In the next project you will be asked to write a

program that plays the ladders game. But, in this project you just have to construct a graph (called the ladders graph) that is the first step in developing a program to play the ladders game.

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`

Your program should construct a graph whose nodes are these five letter words. 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. This is the ladders-graph on 5-letter words.

After this graph has been constructed, determining a shortest ladder is simply a matter of finding a shortest path between a pair of given words. You will learn how to do this later and this will be part of your next assignment. For this project, you only need to write a program that reads words from the file `words.dat`, constructs the ladders graph, and then processes input that consists of one or more lines of the form:

```
P w
```

where `w` is a string. In response to each line of input, your program should print the neighbors of the word `w` in the graph. If `w` is not a word in the graph, your program should produce an error message.

9. Some advice.

This project requires that you integrate several concepts that we have explored in class and it is my expectation that it will take you all of 3 weeks to complete it. So please start early and do not hesitate to ask me questions as you make progress.

Here is my advice on how to break up your work on the project into stages.

Stage 1 Implement the graph class without templates and without dynamic resizing and with nodes hardwired to contain strings. Write the "driver" program described in Section 7 and test your program with small graphs.

Stage 2 Implement dynamic resizing. Write the program that constructs the ladders graph and use this to test your new implementation. If correctly built, this graph contains 5757 nodes and 14135 edges.

Stage 3 Implement the `graph` class and the helper classes as template classes. Run your implementation through the tests you performed in Stage 1 and Stage 2 again.

You should break up work within each stage into pieces and make sure that you implement a piece and test it before moving on to the next piece. You will maximize your grade on this project by using this approach. This is because, it is better to turn in a working program that does a few of the tasks well, rather than a large chunk of code that attempts to do everything, but does nothing.

10. Overall organization.

The `graph` class should reside in files `graph.h` and `graph.cxx`. Similarly, you should have files `node.h` and `node.cxx` for the `node` and files `edge.h` and `edge.cxx` for the `edge` class. The "driver" program described in Section 7 should go into a file called `driver.cxx`. The program that builds the builds the ladders graph and outputs information about it should be in a file called `ladders.cxx`.

In addition to these files that contain your code, you should also have a `README` file that clearly tells us about any known errors your programs have and also lists all required features not implemented.

All of these files should be in a directory that you will have to submit by 5 pm on 10/7/02.