CS 311 - HW 9 - 100 points

Directed Graph

Write a **directed graph** application program to implement and work with directed graphs.

Steps:

1. For this HW, you need the implementation of llist and slist from HW5 and HW6. Make sure that your slist works correctly. Copy them into your HW9 folder. Change the llist typdef to char to store vertex names.
2. Implement the given dgraph class: complete the attached **dgraph.h** and **dgraph.cpp**
3. Refer to Lecture 17 for more information.
4. **Class dgraph contains the following data members:**
   - **Gtable[20]**: an array of 20 structs of type Gvertex
     - Each of these structs represent one vertex of directed graph.
     - The directed graph cannot have more than 20 vertices in this implementation
   - **Struct Gvertex: (this is declared before the class in the header file)**
     - a vertex name (char)
     - the out degree (int) ---> number of edges going out of vertex
     - a linked list object for adjacent vertices **(HW6 slist but with char element type)**
     - the mark/visit number (int) ---> This element will be used in HW10

   And the following methods/member functions:
   - **dgraph Constructor** - initializes the table entries
     - [Make sure all names are initialized to be ‘ ‘ and visit number is 0]
   - **dgraph Destructor** - destroys the table
     - [Does this call the list destructor automatically? If not, you have to destroy the lists. Test and see. **]**
   - **displayGraph()** - displays the table content in a very readable format But make sure you do not display unused slots
   - **fillTable()** - reads the input file **table.txt** to fill the table Open and close the input file table.txt in here
   - **int findOutDegree(char)** - returns the out degree of the vertex whose name is given as an argument.
     - **Must map the vertex name to the slot number.** A=0, B=1, etc.
     - May throw an exception if not found.
   - **slist findAdjacency(char)** - returns the linked list of adjacent vertices of the vertex whose name is given as an argument.
     - **Must map the vertex name to the slot number**. A=0, B=1, etc.
     - May throw an exception if not found.
     - [This one calls your HW6 copy constructor automatically because a list is being returned. Thus your HW6 copy construcor needs to be working.]

Note that the mark/visit number is not being used yet by the client. It will be used in the next HW.

Note that the linked list of adjacent vertices is of type slist and thus, you can use any of the slist member functions on it.

table.txt should have the following format with vertices in the alphabetical order.
Each line is
name out-degree a-list-of-its-adjacent-vertices-separated-by-blanks
e.g.
A 2 B F
I have provided you the input file based on the notes.
You must use it to test your program thorughly. Enter your test results in your Test1.txt

5. Note that your grade will be 0 if you submit some other implementation of directed graph class.
6. Note that you will need this implementation of directed graph class for HW10 as well. Your HW10 will not be graded if your submission to HW9 does not work.
7. Complete the attached program **HW9client.cpp** as follows:
   - **fillTable()**
   - **displayGraph()**
   - **LOOP until the user wants to stop:**
     - **the user will specify which vertex**
     - **findOutDegree(char)**
     - **findAdjacency(char)**
     - **display the returned results (hint: use HW6 function)**
     - **And catche exceptions to display error messages but do not exit.**

8. Note that in this program, exceptions should **NOT** abort the program.
   - Exception handling is necessary.

9. Test your program. The results must match the results given at the end of this file.

Submission

Submit a zip file containing the following files.

1. Llist.h, llist.cpp, slist.h, slist.cpp -- copy them from HW5 and HW6
   - Change the llist typdef to char to store vertex names.
2. dgraph.h -- class header file

3. dgraph.cpp (50 points) -- class implementation file
4. HW9client.cpp (50 points) -- the implemented application
5. test1.txt -- must match the results given at the end of this file

test1.txt compilation and the results of thorough test cases (script) using my table.txt

Important note1: You will miss up to 10 points if you don't comment your programs.

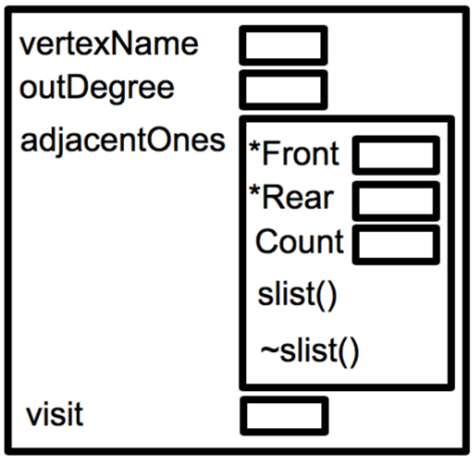
Important Note2: Always make sure the files you submit can be compiled on **empress.csusm.edu** with no error. We will compile and test your files on empress.

More details on implementation

In dgraph class, we define the following struct to store a vertex.

```
struct Gvertex
{
  char vertexName;      //the vertex Name
  int outDegree;                //the Out degree
  slist adjacentOnes;   //the adjacent vertices in an slist
  int visit;           // This will hold the visit number
};
```

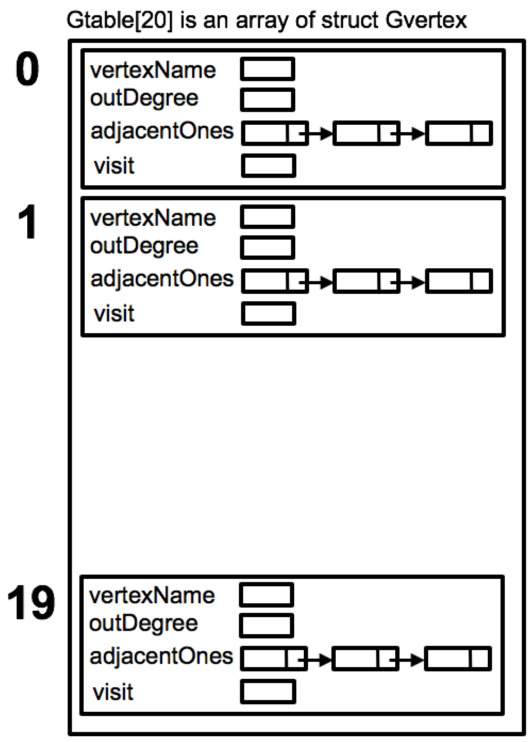
Gvertex is a struct



Then we define an array of struct to store the graph.

```
Gvertex Gtable[20];  // a table representing a dgraph
```

Note: Refer to arrayOfStruct/structDemo.cpp as an example to work with array of struct.



We also define a data member, countUsed, to keep track of the number of used slots in the array, which represent the number of vetrices in the graph.

Working with input file:

In the client, you have to read the contents of table.txt and store them in Gtable array. Note that the number of adjacent vertices for each vertex might be different. Thus, you need the data member outDegree to figure out how many times run a for loop to read an adjacent vertex from file and add it to linked list of corresponding vertex. You may use the following code to work with file. Also refer to inputFile/fileioDemo.cpp for a complete example.

```
//For each line of the input file:
//Read a vertex name.
//If you could read a name you have not reached end of file yet.
while (fin >> Gtable[countUsed].vertexName) //If could read name
```

```
{
   //Now read the next item, which is the outDegree.
   fin >> Gtable[countUsed].outDegree;
   Then for the OutDegree times (in a for-loop):
      fin >> X; // Read an adjacent vertex name, add to list
      (Gtable[countUsed].adjacentOnes).addRear(x);
              // addRear is an slist function from slist class
   countUsed++;
 }
```

Result

```
./hw9.out
A 2 0 [B F ]
B 2 0 [C G ]
C 1 0 [H ]
D 0 0 [empty]
E 0 0 [empty]
F 2 0 [I E ]
G 0 0 [empty]
H 2 0 [G I ]
I 3 0 [A G E ]
Enter a vertex name or character Z to exit: A
2 edges go out to
[B F ]
....in the llist destructor
Enter a vertex name or character Z to exit: B
2 edges go out to
[C G ]
....in the llist destructor
Enter a vertex name or character Z to exit: C
1 edges go out to
[H ]
....in the llist destructor
Enter a vertex name or character Z to exit: D
0 edges go out to
```

```
[empty]
....in the llist destructor
Enter a vertex name or character Z to exit: E
0 edges go out to
[empty]
....in the llist destructor
Enter a vertex name or character Z to exit: F
2 edges go out to
[I E ]
....in the llist destructor
Enter a vertex name or character Z to exit: G
0 edges go out to
[empty]
....in the llist destructor
Enter a vertex name or character Z to exit: H
2 edges go out to
[G I ]
....in the llist destructor
Enter a vertex name or character Z to exit: I
3 edges go out to
[A G E ]
....in the llist destructor
Enter a vertex name or character Z to exit: J
No such vertex exists.
Enter a vertex name or character Z to exit: Z
```