Homework #4

Network of Friends

Due 11:59PM 11/28/2012

Design Document

You will be required to submit a design document for this assignment.

Assignment Description

Your job is to add user accounts for your game to keep track of each player's statistics, such as high score, as well as a network of friends for each user.

The user account information and network for friends is stored in a database file. The database file contains information about individual users and which users are friends with each other. Friendship is a **bi-directional** relationship. This means that if user **A** is a friend of user **B**, then user **B** must also be a friend of user **A**. The database file must conform to the database file format specification. The default database filename is**"db.txt"** in the same directory as your program.

You must use a C++ class to represent a single user. You may use any other C++ classes as you see fit.

When your program starts, it must read the default database file to get all the information and store them in in-memory data structures.

Your program must not expect the user to enter data in the right format. Your program must be able to handle user input that looks like complete garbage! If the user rolls his/her hands on the keyboard or sits on the keyboard, your program should still prints out reasonable error messages. For example, if you prompt the user for a user ID and the user enters "xyz abc *@#", you should let the user knows that you don't understand the input and you are expecting a valid user ID.

You are **NOT** allowed to use any of the containers or algorithms from the C++ Standard Template Library (STL) except for the vector and the queue classes.

Database File Format

The database is an ASCII text file with multiple human-readable lines of text. The first line contains the number of users in the database. We will refer to this number as **N**. We will refer to the users as users **0** through **N-1**. Every user record is **5** lines long and contains a user ID, username, birth year, state of residence, and a list of user IDs of friends. The total number of lines in a database file must be exactly **5 × N + 1**. The required format of the file is as follows:

Line#	Description
1	**N** (a number representing how many users are in the database)
2	**id_0** (user ID of user 0)
3	<TAB>**username_0** (unique username of user 0)
4	<TAB>**year_0** (year at USC of user 0)
5	<TAB>**score_0** (high score of user 0)

6	<TAB>**ids of friends** (separated by <TAB> characters)
7	**id_1** (user ID of user 1)
8	<TAB>**username_1** (unique username of user 1)
...	...
5N-3	**id_(N-1)** (user ID of user N-1)
5N-2	<TAB>**username_(N-1)** (unique username of user N-1)
5N-1	<TAB>**birthyear_(N-1)** (year at USC of user N-1)
5N	<TAB>**score_(N-1)** (high score of user N-1)
5N+1	<TAB>**ids of friends** (separated by <TAB> characters)

To make sure that the database file is not corrupted, when your program read from the database file, you must **validate** the data. The requirements are:

- The first line must contain a single positive integer **N** with $1 \leq N < 1{,}000{,}000{,}000$. There must be no non-numeric characters in this line.
- A **user ID** must be a positive integer ≥ 1 and $< 1{,}000{,}000{,}000$. There must be no non-numeric characters in this line. A user ID must be unique in a database.
- A **username** must be a string containing only alphanumeric and the underscore characters. There must be no other type of characters in this line, except for the leading <TAB> character. A username must be unique in a database. A username is case-sensitive. Therefore, "bob" and "Bob" are considered different usernames.
- A **year** must be a positive integer ≥ 1 and ≤ 5. There must be no non-numeric characters in this line, except for the leading <TAB> character.
- A **score** must be a an integer. There must be no other type of characters in this line, except for the leading <TAB> character.
- A list of **friend user IDs** are a list of numbers, separated by one or more space characters or <TAB> characters. The first character of this line must be a <TAB> character. There must be no other type of characters in this line. The IDs must be valid user IDs in the database and may refer to users whose data appear later in the database. If a user has no friends, this line should contain only a single <TAB> character.

All fields are required for all users. If there are any "dangling references" to friends that do not exist in the database, the database is considered corrupted.

Data Structure to Keep Track of a Network of Friends

The basic data structure that can be used to keep track of a network of friends is called an **Adjacency List**. In an adjacency list data structure, you would use a vector of users. For example, you can have:

```
vector<User> userList;
```

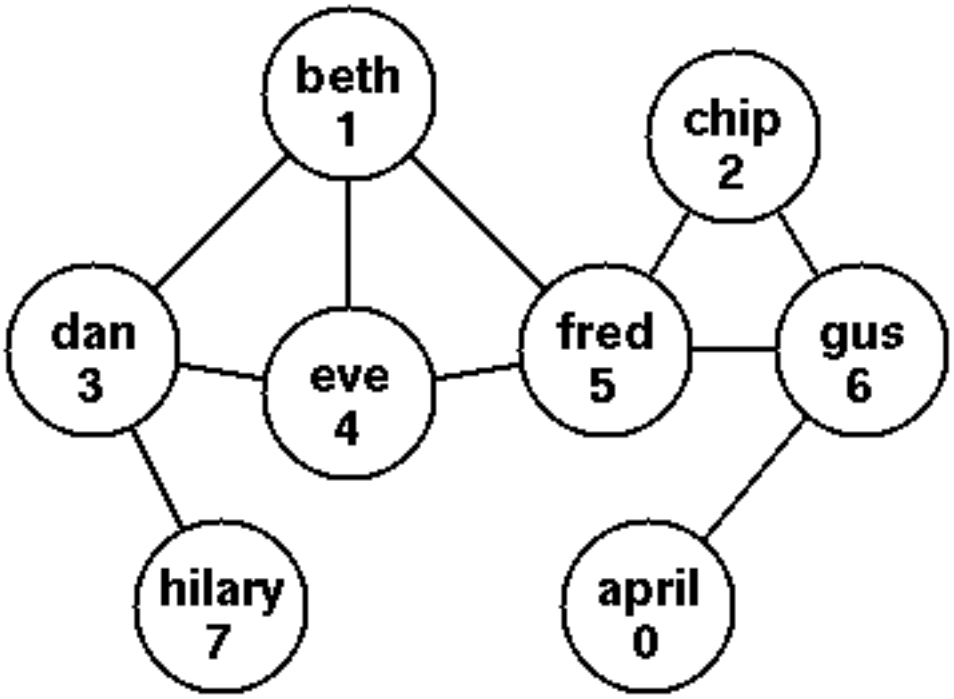
To refer to a particular user, you would use the index. If there are **N** users, they would be referred as user **0** through **N-1**. For example, to refer to user **i**, you can use:

```
userList[i]
```

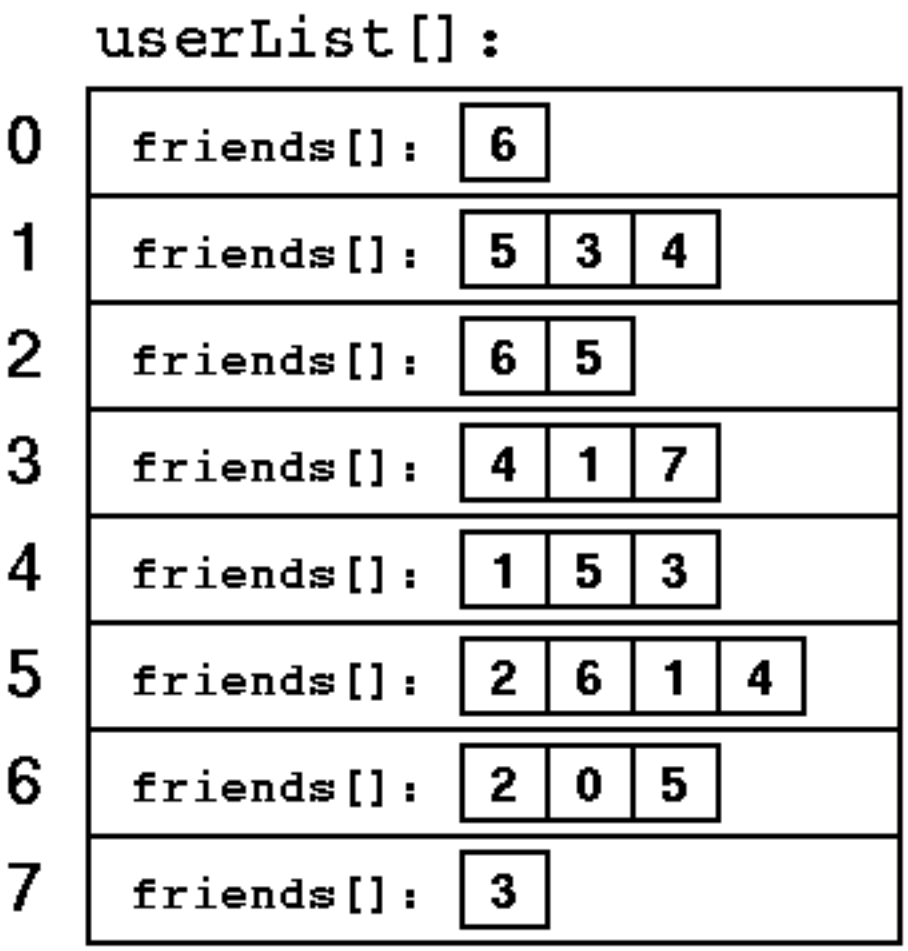
Please note that array indices and user IDs are completely different things.

Inside the data structure that represents a user is a vector of integers named **friends[]**. For user **i**, these integers are indices that correspond to friends of user **i**. Please note that friendship is a bi-directional relationship. This means that if **j** is in **i's friends[]** vector, then **i** must be in **j's friends[]** vector.

Let's look at an example. The figure below is a **graph representation** of a network of friends. Each circle (known as a **node** in the graph) represents a user. The number inside a circle is the index into **userList[]**. Friendship between users **i** and **j** is represented by an **edge** connecting nodes **i** and **j**. Some information about the users is omitted in the graph.



An **adjacent list** representation of the above graph is depicted in the figure below.



Breadth First Search

The **Breadth First Search** (or **BFS**) algorithm can be used to **rank** nodes relative to a **start node**. Let's call this start node **S**. The **rank** of a node **X**, relative to node **S** is that smallest number of edges one can traverse to travel from node **S** to node **X**.

To facilitate the running of the BFS algorithm, a **queue** will be used. A **queue** is a **first-in-first-out** (or **FIFO**) list data structure where a data item can be appended to the **back** of the list by using the **push_back** method. A data item can be removed (or **popped**) from the **front** of the list by using the **pop_front** method.

Given that you use the Adjacent List data structure described above, below is the **pseudo-code** for a queue-based BFS algorithm (please note that since this is **pseudo-code**, you cannot run the code as-is; you need to

```
( 1)  initialize all nodes to be "unvisited" (set rank of -1 to all nodes)
( 2)  mark node S as visited, rank[S] = 0, prev[S] = null
( 3)  append S to search queue
( 4)  while (the search queue is not empty)
( 5)      pop the first node from the search queue, call this u
( 6)      for each node v adjacent to u
( 7)          if (v has not been visited)
( 8)              mark v as visited, rank[v] = rank[u] + 1
( 9)              add v to the search queue
(10)              prev[v] = u
(11)          end if
(12)      end for
```

```
(13)  end while
```

Please note that when the above algorithm terminates, you are guaranteed to have assigned a rank to every node that can be reached from node **S**. Nodes that are not reachable from node **S** will have a rank value of **-1**.

User Interface

There are basically two user interface states; (1) a user is **logged on** or (2) no user is logged on. When no user is logged on, your top-level screen should look like the following:

```
Number of users: 8 Number of relations: 10
Please enter one of the following commands:
(+) Add user account
(-) Delete user account
(L) Log on to user account
(H) List all users sorted by high scores
(q) Quit
```

If a user logs on, your top-level screen should look like the following:

```
Number of users: 8 Number of relations: 10
User logged on:
user name: beth
year: Freshman
high score: 100020
friends: dan, eve, fred

Please enter one of the following commands:
(a) List all usernames
(f) Add a friend for beth
(d) De-friend one of beth's friends
(s) Sort all beth's friends by score
(H) List all users sorted by high scores
(B) Shortest-path to beth from any user
(P) Play game
```

The shortest-path from user **X** to **S** can be obtained by following the **prev** field starting from **X** after the BFS algorithm has terminated. If the rank of node **X** is **-1**, then there is no way to reach node **S** from node **X**. For example, if beth is node **S**, for april, you should print the following:

```
shortest path from april to beth is: april, gus, fred, beth
```

Summary of Requirements

- You must have a fully-written Design Document.
- You must comment your code (all classes, all functions and anywhere else that needs it).
- Your application must have functional a menu-drive to allow the user to select which action(s) they would like to perform that meets **all the requirements** specified above.
- You must use a C++ class to represent a user.
- You must validate data read from a database file and when a user is added according to the requirements specified above.
    - You must not allow duplicate user IDs.
    - You must not allow duplicate usernames.
    - If validation failed when a user is added, you should prompt the user again for a correct value (and you should give a reasonable error message to let the user know why the previous input was rejected).
    - Before you load a database file, you should clear out all the users information from your in-memory data structures. If validation failed when a database file is being loaded, you should clear out all the users information that you have processed from your in-memory data structures.
- You must not allow a friendship to be formed to a non-existing user.

- You must provide the user a way to cleanly terminate your application.
- Shortest-path to the logged on user from another user.

Grading Breakdown (40 Points)

- Design writeup for entire assignment (5 points)
- Code comments (3 points)
- Functional menu-driven user interface for all application options (2 points)
- Proper implementation of loading from a database file (including the default database file) (2 points)
- Proper implementation exporting to a database file (2 points)
- Proper implementation of user log on, list all user names, list all information (2 points)
- Proper implementation of add a user (including no improper values accepted) (2 points)
- Proper implementation of delete a user (including deleting all dangling friend references) (3 points)
- Proper implementation of add a friend (2 points)
- Proper implementation of de-friend (2 points)
- Proper implementation of sorting user's friends by score (3 points)
- Proper implementation of sorting all users by score (3 points)
- Proper implementation of shortest-path from a user (9 points)

IMPORTANT: Some operations depend on the proper implementation of other operations. When such dependency exists, **you may not get any points** for a part if the part it depends on is broken. The number of points you get will be decided at the TA/graders' discretion.

IMPORTANT: Before you submit your code, **MAKE SURE** that your code compiles and runs properly. Please also note that you must submit your code using the Blackboard System and verify your submission afterwards to ensure that what you have submitted is a valid submission. E-mailed submissions will **not be accepted**.