Programming assignment 2: Linked lists. Trees. Friends.

The goal of this assignment is to build a useful tool for finding friends while exercising linked data structures in C.

Let’s assume that we want to be friends with people with whom we share interests. How does one find such potential friends?

You ask questions about people preferences such as “Do you like Coding?” or “Do you like Parties?”, and depending on the answers you distribute all people into groups of like-minded individuals. Then you can recommend potential friends who answered the questions similarly to each other. A high-level idea is depicted in Figure 1.

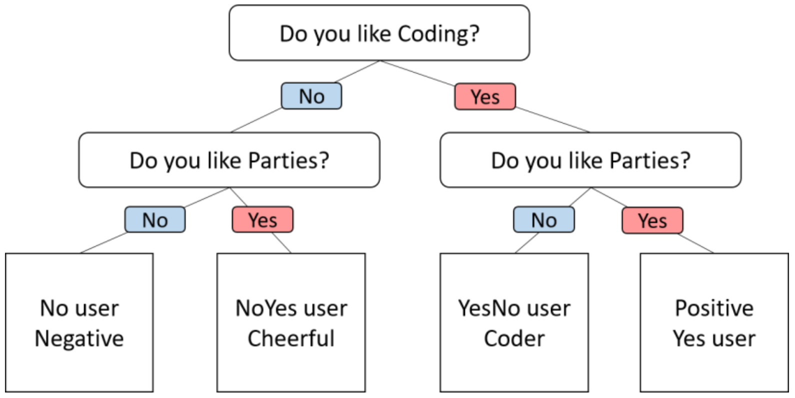


Figure 1. Example of distributing people into groups based on their interests.

We assume that *Cheerful* more likely will befriend *NoYes* user, than *Coder*. And that is what our program is going to do: recommend potential friends. The suggested list of interests is provided in file *interests.txt*, and can be later replaced with your own list of more serious topics.

Your task is to develop a *categorizer* program in C. This is a local prototype of the program, which will be extended into an online multi-user application in the last assignment.

Program specifications

The program runs for an indefinite amount of time and is terminated when the user types “q” as an answer to the next prompt.

Step 1. The program asks for a user name. The user name uniquely identifies each user and may contain any alphanumeric character and has length limitations: at least 8 and at most 128 characters. Alphanumeric characters are numbers 0-9, and letters of the alphabet A-Z or a-z. User name is case-sensitive – that is, the user named “Bob” is different from the user named “bob”.

The app validates the name entered from the command-line. In case of an invalid user name, it issues an appropriate message and goes back to the beginning of Step 1.

Next, it checks whether the user with this name already exists.

If this is a new user, the application proceeds to Step 2. Otherwise, it goes directly to Step 3.

Step 2. The program runs a test to determine to which group should it assign the user based on their interests. It asks a series of questions, and accepts the yes/no answers from the user. It treats user answers in a *case-insensitive* manner. The user may enter up to 3 characters, and the program only checks the first letter of each answer: “y”, “n” or “q”. If the user entered an invalid answer – longer than 3 letters or not one of {“yXX”, “nXX”, “qXX”} – the program issues an “invalid answer” message and repeats the question.

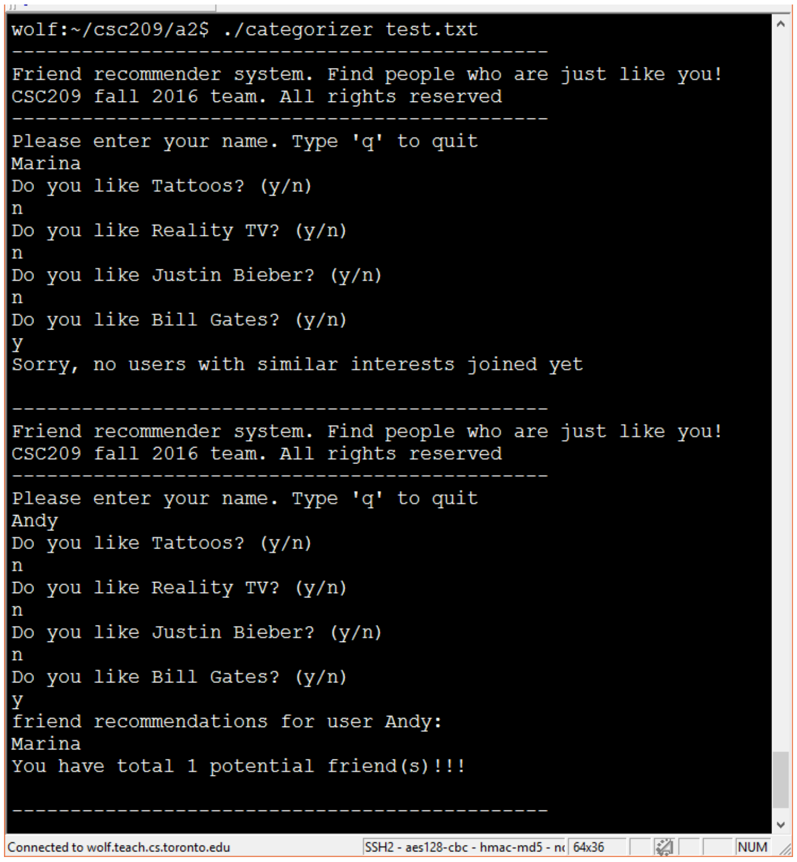
Once the new user has answered all the questions, it is added to the corresponding group of people who have given identical answers to these questions. The program proceeds to Step 4.

Step 3. If the user with this name already exists, the program does not run the test, but goes directly to step 4.

Step 4. We assume that at this point we know the group that the current user belongs to. The list of all users from this group other than the current user itself is printed to standard output as a recommended list of potential friends.

After printing the list of friends (if any), the program proceeds to the beginning of Step 1, collecting information about the next user. If - at any point - user types “q” in response to the next program prompt, the program terminates.

A screenshot of a sample program run is presented in Figure 2.



```
wolf:~/csc209/a2$ ./categorizer test.txt
---------------------------------------------
Friend recommender system. Find people who are just like you!
CSC209 fall 2016 team. All rights reserved
---------------------------------------------
Please enter your name. Type 'q' to quit
Marina
Do you like Tattoos? (y/n)
n
Do you like Reality TV? (y/n)
n
Do you like Justin Bieber? (y/n)
n
Do you like Bill Gates? (y/n)
y
Sorry, no users with similar interests joined yet

---------------------------------------------
Friend recommender system. Find people who are just like you!
CSC209 fall 2016 team. All rights reserved
---------------------------------------------
Please enter your name. Type 'q' to quit
Andy
Do you like Tattoos? (y/n)
n
Do you like Reality TV? (y/n)
n
Do you like Justin Bieber? (y/n)
n
Do you like Bill Gates? (y/n)
y
friend recommendations for user Andy:
Marina
You have total 1 potential friend(s)!!!

---------------------------------------------
```

Connected to wolf.teach.cs.toronto.edu
SSH2 - aes128-cbc - hmac-md5 - nc
64x36
NUM

Figure 2. Screenshot of a running program.

Implementation details

What follows is a set of step-by-step instructions for implementing the above functionality. We propose working in steps, adding new functionality only after you thoroughly tested the previous one. We also propose to use a short version of the interests file for testing correctness of each step, such as for example the first 4 lines of interests.txt.

This is an example of a larger program, where you are going to distribute the functionality among multiple source files and compile the entire program using *make* utility. The good starting point about make is here, and you will also have a special lab session dedicated to writing make files.

The content of the directory A2 is presented below:

categorizer.c	← contains main function for running the app
qtree.c	← contains functions for managing question tree
questions.c	← contains functions for reading questions from the file into an array of C strings
test1.c	← contains main function for testing the question reading part
test2.c	← contains main function for testing the question tree functionality
questions.h	← contains declarations and data types needed for the question reading part
qtree.h	← contains declarations and data types needed for the question tree

To make sharing code between multiple source files possible and to compile multiple files as a single compilation unit, you need to declare the functions and custom data types in your own header files.

However, it is possible that different parts of the application ask for the same header files to be included. To prevent compiler complaints about double declarations, you put *include guards* around the content of each header file, like this:

```
#ifndef HEADERFILE_H

#define HEADERFILE_H

        Your declarations here

        and at the end of the file is:

#endif
```

Once the header is included, it checks if a unique value (in this case HEADERFILE_H) is defined. Then if it's not defined, it defines it and continues to including the rest of the page. When the code is included again, the first `ifndef` fails, resulting in a blank file. That prevents double declarations.

After each task is completed, add the corresponding compilation unit to the make file, and test the overall functionality with suggested tests.

Task 1. List of interests

The first task is to read the list of interests from a file whose name is supplied as a single command-line argument. This time, the reading is from a file, not from standard input, and we use *fgets* for reading lines. Read man pages for *fgets*. The only catch about *fgets* is that it reads the line including its end-of-line characters. As the input file can be supplied in different formats, the line endings could be of type "\n", "\r\n", or any combination thereof. The best way to handle this is to replace the first encountered end-of-line character with the string termination character, like this:

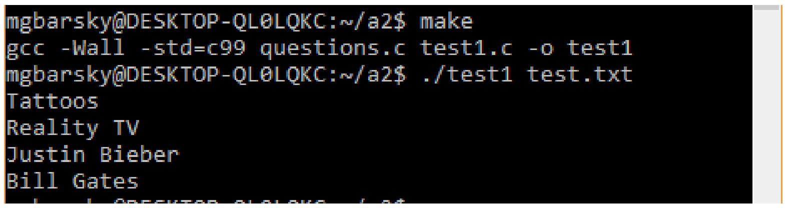
```
while (fgets (line, MAX_LINE, inputFP)!=NULL) {

      line [strcspn (line, "\r\n")] = '\0';   //remove end-of-line characters

}
```

`strcspn` simply returns the position of the first occurrence of any of '\n' or '\r', and we put '\0' into this position, creating a valid C string.

You do not know in advance how many lines you are going to read. The best way to handle a list of an unknown size is to use a *linked list*. As you read each line, you add it to the linked list. Both the nodes of the list and the string data inside each node should be **dynamically allocated on the heap**. You have to preserve the order of lines in your linked list, and therefore you should add new nodes at the **end** of the list.

The line reading and list creation functionality should be implemented in a separate C file *questions.c*, and the function declaration should be in a separate header file *questions.h*.

When you are done, write function *main* in a separate file *test1.c* in which you test your line reader functionality and print the content of a list to standard output in the following format:



```
mgbarsky@DESKTOP-QL0LQKC:~/a2$ make
gcc -Wall -std=c99 questions.c test1.c -o test1
mgbarsky@DESKTOP-QL0LQKC:~/a2$ ./test1 test.txt
Tattoos
Reality TV
Justin Bieber
Bill Gates
```

Figure 3. Sample output from test1.

An example of a simple starter *Makefile* to handle the Task 1 compilation unit is presented below. You will extend the Makefile with new compilation units as you complete new tasks.

```
CC = gcc
CFLAGS = -Wall -std=c99
all: questions
questions:
      $(CC) $(CFLAGS) questions.c test1.c -o test1
clean:
      rm test1
```

Task 2. Question tree

In this part, you need to implement a binary tree of questions, where each node contains a (dynamically allocated) string to store the question itself and two child nodes, which correspond to No/Yes answers. You may think of defining the following struct:

```
typedef struct QNode {
      char *question;
      struct QNode *  children[2];
} QNode;
```

Using this new type, you could connect multiple QNodes into a binary tree, by populating them with the interest strings from the list that you have created in Task 1. The tree is (sadly) exponential in the total number of questions, but this should not be a problem for the scale of this application (we are not using more than 15 questions, and thus the total number of nodes will not exceed ~70,000).

Each tree level contains the same question. Each node has exactly two children, corresponding to "yes" (`children[1]`) and "no" (`children[0]`) answers. Now, when recording user's answers, the program starts from the top of the tree and asks the question which is stored in the current QNode. It then moves to the right or to the left child, depending on the answer, and asks the next question. Thus the user moves through the tree until all questions have been asked and answered. An example of a tree for 3 questions is shown in Figure 4.

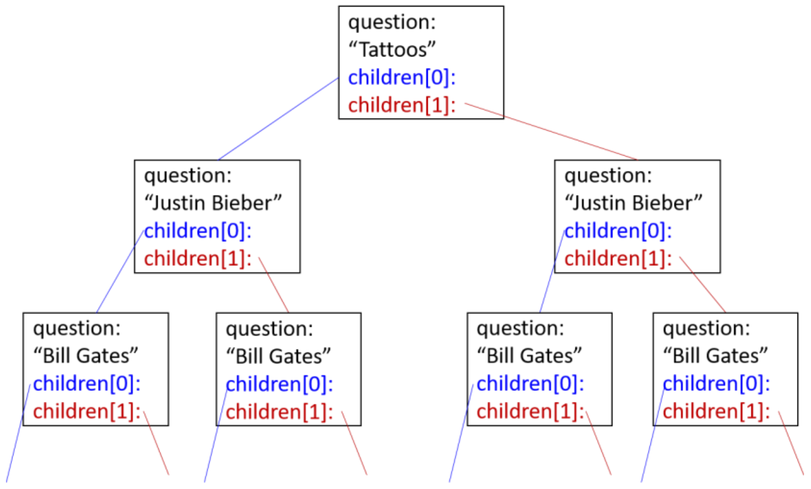


Figure 4. Sample tree of connected Question Nodes.

But wait, what happens when all the questions have been asked? In this case we reached the leaf node of the question tree. The leaf node corresponds to the last question, but its yes/no branches do not lead to the next QNode. Each child node at the last level should store lists of potential friends who had exactly the same answers moving through the same path of yes/no branches in the tree.

The number of friends for each path is not known in advance, so you need to store user names as linked lists of friends. You declare a struct Node, in which you store the user name and the link to the next Node.

```
typedef struct str_node {
      char *name;
      struct str_node *next;
} Node;
```

The only problem to solve is to make the Yes and No children of the last question node to point to Nodes instead of QNodes. We achieve this using *union*:

```
union Child {
      struct Node *fchild;
      struct QNode *qchild;
} Child;

typedef struct QNode {
      char *question;
      union Child children[2];
} QNode;
```

Now each node has a choice to point either to another QNode or to the Node. The leaf node of the tree will point to either the first Node in the linked list of friends, or to NULL if there are no users corresponding to this specific path in the tree.

When you can add different types of child nodes to each QNode, it becomes important to store the node type: REGULAR (internal) node or a LEAF node. Otherwise, given an arbitrary node, you would not know what type of children it contains. So let's create *enum* to store two node types. The final version of QNode is presented below.

```
typedef enum {
      REGULAR, LEAF
} NodeType;

typedef struct str_node {
      char *name;
      struct str_node *next;
} Node;

union Child {
      struct Node *fchild;
      struct QNode *qchild;
} Child;

typedef struct QNode {
      char *question;
      NodeType node_type;
      union Child children[2];
} QNode;
```

You have all the data types ready and you can link QNodes into a tree. The final design of the question tree is depicted in Figure 5.

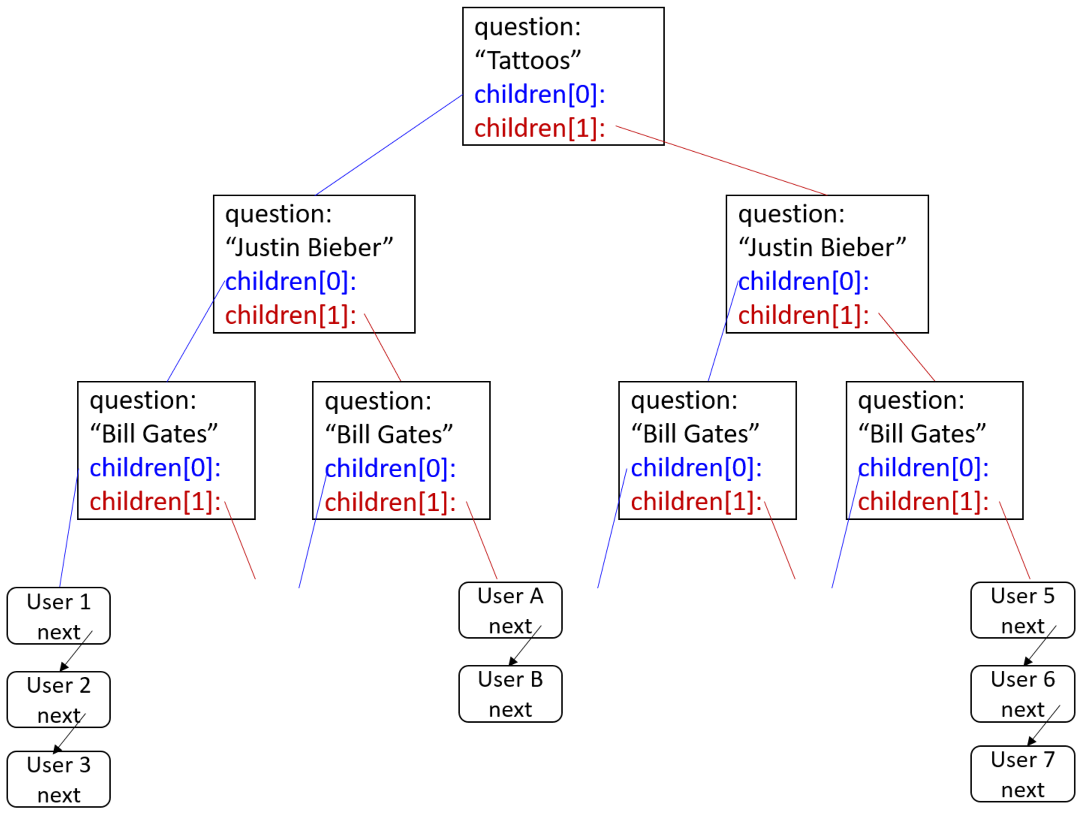


Figure 5. Question tree with QNodes and Nodes

Two sample functions are provided in file *q_tree.c*: *add_next_level* and *print_qtree*. The skeleton of the test for these two functions is also provided in file *test2.c*. Function *add_next_level* recursively adds next level of the tree based on the current node of the linked list of questions. You will not be able to test the functionality of tree creation, until you finish Task 1 and build a linked list of questions. Your function should return the head of the list, and you store it in variable *q_list*. First of all, test that you can create question tree with your list of questions, and then implement the additional functionality as following.

Implement tree traversal. The tree is traversed in a depth-first manner, and the easiest way to implement this is by using recursion. The recursion stops when you reach the node of type LEAF.

Now implement adding users to different tree leaves. This should be a function which takes as arguments current QNode and the answer in form of 0 or 1. The answer determines which branch of the tree to traverse to reach the next QNode – either left (answer no, child 0) or right (answer yes, child 1). The function returns next QNode reached through the corresponding branch. Once the function reaches a leaf node, the type of a child changes from QNode to Node, and we can attach the list of friends – a linked list of Nodes.

To test this functionality, pass the following command-line arguments to test2.c:

- The name of the input file with questions
- The user name

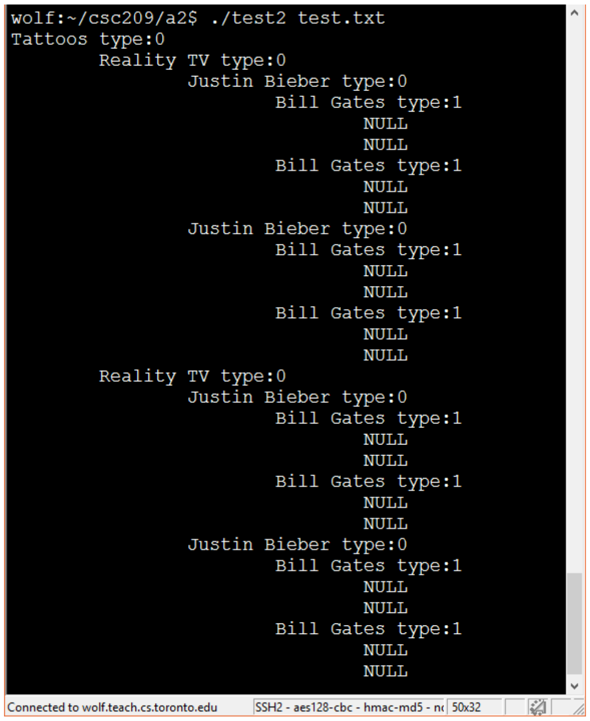
- The list of answers in form of 0 or 1. Each answer is provided as a separate command-line argument. The number of answers that you provide should correspond to the number of questions in the input file you are using. Check your argument count against question count. In case they do not match - issue the corresponding error message and exit.

Now test your question tree by running *test2* with different command-line parameters. By the end of each run, print the tree to the standard output using function *`print_qtree.`* You are not allowed to modify the format of the printed tree, so please consider using suggested data types and functions instead of writing your own.

Finally, implement the search for a user with a given user name in one of the lists attached to the bottom of the question tree. This function should perform a depth-first traversal of the entire tree. Once the leaf node is reached, function checks for a given user name in the lists stored in the left or right child of the current leaf node. If the user name is found, function returns the corresponding list of friends. Otherwise it proceeds to the next leaf node and checks again. If the entire tree has been traversed and the user name is not found, then the function returns NULL. You need to test this functionality, but you do not need to write a special test code. It will undergo the final test in Task 3.

All the above functionality should be implemented in a separate file *qtree.c* with its own header file *qtree.h*, where you put all your function declarations and custom data types.

The way you connect QNodes into the tree, and the way you implement the entire application is up to you. However, I repeat, your printing functionality has to produce an output in exactly the same format as *`print_qtree.`*



```
wolf:~/csc209/a2$ ./test2 test.txt
Tattoos type:0
        Reality TV type:0
                Justin Bieber type:0
                        Bill Gates type:1
                                NULL
                                NULL
                        Bill Gates type:1
                                NULL
                                NULL
                Justin Bieber type:0
                        Bill Gates type:1
                                NULL
                                NULL
                        Bill Gates type:1
                                NULL
                                NULL
        Reality TV type:0
                Justin Bieber type:0
                        Bill Gates type:1
                                NULL
                                NULL
                        Bill Gates type:1
                                NULL
                                NULL
                Justin Bieber type:0
                        Bill Gates type:1
                                NULL
                                NULL
                        Bill Gates type:1
                                NULL
                                NULL
```

Connected to wolf.teach.cs.toronto.edu
SSH2 - aes128-cbc - hmac-md5 - n
50x32

Figure 6. Required format for printing question trees.

Task 3. Categorizer

Finally, you need to write the application itself.

In a yet another source file *categorizer.c*, implement the main application loop.

When the application starts, it reads the questions from the input file whose name is provided as a command-line parameter, builds a linked list (Task 1), then builds the question tree (using provided function *add_next_level*), and starts an infinite while loop, which terminates only when the user answers “q” to any of the prompts. The program behaves according to specifications (read them again). It asks for user name, validates the name, tests if the name already exists, asks the questions about the interests of a new user, and outputs the list of users from the same interest group. After that, the application goes to the beginning of the loop and asks for the next user name.

The application should be compiled into executable *categorizer*, and you can test all user interface aspects by using a shortened version of the interests file – for example first 4 lines.

Test to confirm that the following works:

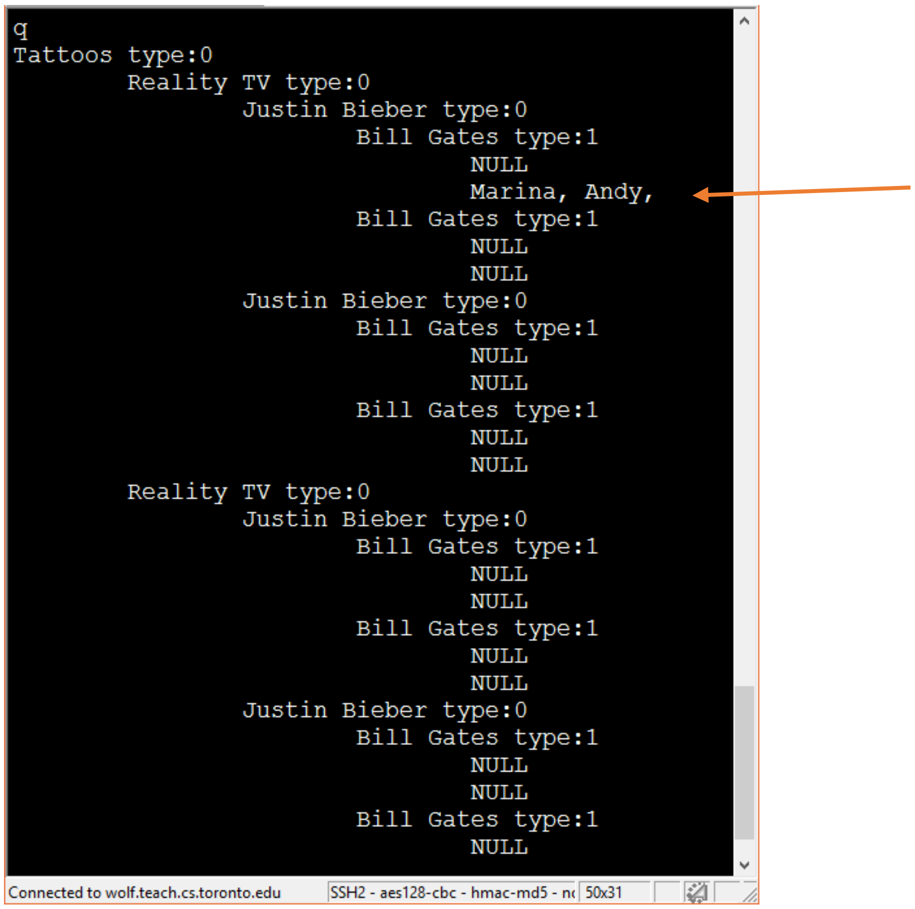
1. Invalid user name (non-alphanumeric characters, too short, too long) is rejected and the corresponding message is issued.
2. Invalid answer string (longer than 3 characters, none of “yXX”, “nXX”, “qXX”) is not accepted, the corresponding message is issued, and the question is repeated.
3. An existing user with the same name is found and the list of potential friends is printed *without* asking the questions.
4. Two users with the same interests end up in the same group.
5. Two users with at least one different answer end up in two different groups.

Run each test separately. Before running each test, run shell command `script (man script)`, and record the output of each test into a separate file. Name each test run as *output1.txt*, *output2.txt*, *output3.txt*, *output4.txt*, and *output5.txt*. For tests 3,4,5 make sure to print the trees with the friend lists in a format similar to the one shown in Figure 7 (using *print_qtree*).

Final task. Cleaning the memory

To learn how to program in C without producing dangerous memory leaks, you need to make sure that you free all dynamically allocated memory after you are done using it. For example, after you have added questions to the tree, you do not need the original question list anymore, and you can free the memory occupied by this list. An example of cleaning memory of a linked list is presented in the lecture. For cleaning memory of the question tree at the end of the program, you can use the depth-first traversal of the tree, and free each node only after its children have been freed. This can be easily implemented using recursion.

Finally, if you need help with implementing any of the above, please use help provided by your TA or the instructor during allocated hours. Please do not post large fragments of code on the discussion forum.



```
q
Tattoos type:0
        Reality TV type:0
                Justin Bieber type:0
                        Bill Gates type:1
                                NULL
                                Marina, Andy,
                        Bill Gates type:1
                                NULL
                                NULL
                Justin Bieber type:0
                        Bill Gates type:1
                                NULL
                                NULL
                        Bill Gates type:1
                                NULL
                                NULL
        Reality TV type:0
                Justin Bieber type:0
                        Bill Gates type:1
                                NULL
                                NULL
                        Bill Gates type:1
                                NULL
                                NULL
                Justin Bieber type:0
                        Bill Gates type:1
                                NULL
                                NULL
                        Bill Gates type:1
                                NULL
```

Connected to wolf.teach.cs.toronto.edu
SSH2 - aes128-cbc - hmac-md5 - n
50x31

Figure7. Sample question tree with friends.

What to submit:

Source C files:

- *questions.c*
- *qtree.c*
- *categorizer.c*
- *test1.c*
- *test2.c*

Header files:

- *questions.h*
- *qtree.h*

Makefile

Test results:

- *output1.txt, output2.txt, output3.txt, output4.txt, output5.txt*

Marking scheme:

Running *make* on CDF machines **compiles** your source code into three executables: *test1*, *test2*, and *categorizer*, without warnings and using c99 standard**.** Failure to compile results in a total mark of zero for this assignment.

The *test1* program produces the **correct output**: **20%**

The *test2* program produces the **correct output**: **30%**

All tests for the *categorizer* program run as expected and are recorded in output files: **40%**

There are **no memory leaks** – all memory has been freed before program terminates: **10%**.

For a total of 9 points towards the course grade.