Week 05 Lab Exercise

Graphs and Social Networks

Objectives

- To explore an application of graphs
- To get some practice with graph problems
- To perform complexity analysis on graph algorithms
- To implement some basic features of social networks

Admin

Marks	5 (see the Assessment section for more details)
Demo	in the Week 5, 7 or 8 lab session
Submit	see the Submission section
Deadline to submit to give	12pm Monday of Week 7
Late penalty	0.2% per hour or part thereof deducted from the attained mark, submissions later than 5 days not accepted

Background

scenarios and systems to be modelled by graphs - for example, maps, social networks, and the web. In this lab, we will explore an application of graphs in a simple social network app called Friendbook.

Friendbook

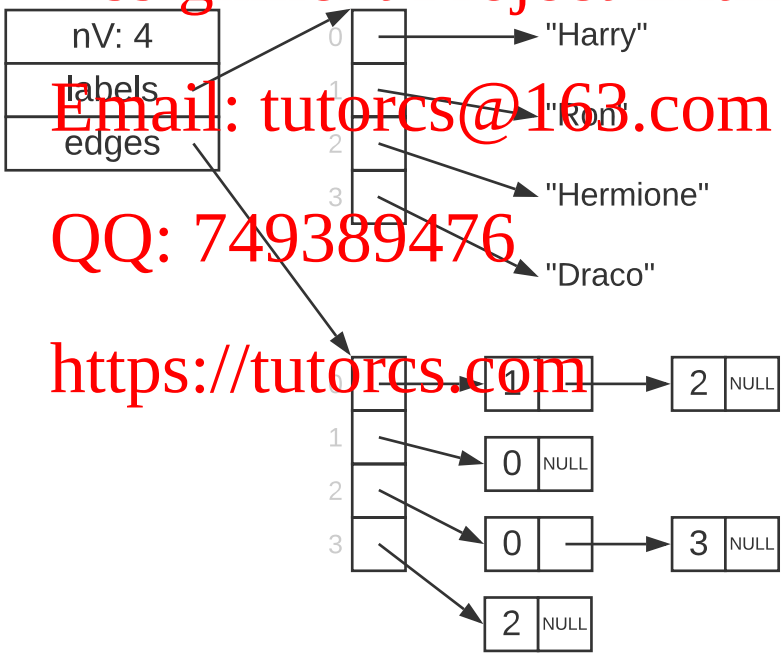
Friendbook is a very [illegible] network app with the following features:

- People can sign u[illegible]me. For simplicity, people are identified by their names, so two people cannot have the same name.
- People can friend other people (i.e., add them as friends). Friending goes both ways, so if you add someone as a friend, you become their friend as well.
- People can unfriend their friends (i.e., remove them from their friends list). This also goes both ways.
- People can see a count of how many friends they have.
- People can see a list of their friends.
- People can see a list of the mutual friends that they share with someone else.
- People can receive friend recommendations. Friendbook has two different methods of generating recommendations:
  1. The first method only recommends friends of friends, and ranks friend recommendations in order of the number of mutual friends, so people who you share more mutual friends with will be recommended first.
  2. The second method recommends friends of friends first, and then friends of friends of friends next, and then friends of friends of friends of friends, and so on. Anyone who can be reached by following friendship links can be recommended.

Names as Vertices

All of the graph implementations we have seen so far have used integer vertices numbered from 0 to $n-1$, where n is the number of vertices. This is convenient, as vertex numbers can double as indices into the adjacency matrix

It turns out we don't need to do that much more work. If we give each person an integer ID between 0 and $n-1$ and store a mapping between names and IDs, then we can co[illegible]e graph representations that we are familiar with. A simple way t[illegible]s mapping would be to store all the names in an array, and let th[illegible]erson be the index containing their name in the array. The first p[illegible]ay would have an ID of 0, the second person in the array would have an ID of 1, and so on. If we wanted to answer a question involving one or more people, we can scan this array to determine their ID, and then use this ID to query the matrix/list. For example, suppose this is our internal representation:



Now suppose we wanted to find out if Harry and Draco are friends. First, we need to find the vertex numbers associated with Harry and Draco, so we perform a linear scan of the array of names (called `labels`), and find that Harry is associated with a vertex number of 0, and Draco is associated with a vertex number of 3. The adjacency list for vertex 0 does not contain vertex 3, so we can conclude that Harry and Draco are **not** friends.

Unfortunately, converting from a name to a vertex number adds quite a bit of overhead to our graph operations. Converting from a vertex number to a name is easy, as we can go straight to that index in the array ($O(1)$), but converting

We can improve the efficiency of the name to vertex number conversion by using a data structure that allows for efficient searching, such as an **ordered array**. Friendbook u to map names to vertex numbers and efficiently find the v ssociated with a particular name (see the `nameToId` functior e the Map ADT is implemented using an ordered array, this lo hieved in $O(\log n)$ time.

Setting Up

Create a directory for this lab, change into it, and run the following command:


```
$ unzip /web/cs2521/24T2/labs/week05/downloads/files.zip
```

If you're working at home, download `files.zip` by clicking on the above link and then unzip the downloaded file.

If you've done the above correctly, you should now have the following files:

`Makefile`	a set of dependencies used to control compilation
`Fb.c`	an incomplete implementation of the Friendbook ADT
`Fb.h`	the interface for the Friendbook ADT
`List.c`	a complete implementation of the List ADT
`List.h`	the interface for the List ADT
`Map.c`	a complete implementation of the Map ADT
`Map.h`	the interface for the Map ADT
`Queue.c`	a complete implementation of the Queue ADT
`Queue.h`	the interface for the Queue ADT
`runFb.c`	a program that provides a command-line interface to the Friendbook ADT

cases.

Once you've got the [illegible]t thing to do is to run the command

```
$ make
```

This will compile the [illegible] of the files, and produce the `./runFb` executable.

File Walkthrough

`runFb.c`

`runFb.c` provides a command-line interface to the Friendbook ADT. It creates a Friendbook instance, and then accepts commands to interact with it. Here is an example session with the program once it is working correctly:


```
$ ./runFb
Friendbook v1.0
Enter ? to see the list of commands.
> ?
Commands:
    + <name>                add a new person
    l                       list the names of all people
    f <name1> <name2>       friend two people
    u <name1> <name2>       unfriend two people
    s <name1> <name2>       get the friendship status of two
people
    n <name>                get the number of friends a person has
    m <name1> <name2>       list all mutual friends of two people
    r <name>                get friend recommendations for a
person based on mutual friends
    R <name>                get friend recommendations for a
person based on friendship closeness
    ?                       show this message
    q                       quit

> + Harry
Harry was successfully added to Friendbook!
```

```
Hermione was successfully added to Friendbook!
> f Harry Ron
Successfully fri[illegible]and Ron!
> f Ron Hermione
Successfully fri[illegible]d Hermione!
> s Harry Ron
Harry and Ron a[illegible]
> n Harry
Harry has 1 friend.
> n Ron
Ron has 2 friends.
> s Harry Hermione
Harry and Hermione are not friends.
> m Harry Hermione
Harry and Hermione's mutual friends:
    Ron
> r Harry
Harry's friend recommendations:
    Hermione                1 mutual friends
> u Harry Ron
Successfully unfriended Harry and Ron!
> s Harry Ron
Harry and Ron are not friends.
> q
$
```


`Fb.c`

`Fb.c` implements the Friendbook ADT. Most of the functions are complete, however, it would be helpful to read through these functions to get a good idea of how they manipulate and obtain information from the graph representation, how they create and return lists of names, and how they convert people's names to vertex numbers. You should also read the definition of `struct fb` and make sure you understand the purpose of each field.

`List.h`

see how to create a list and add names to the list, you should read some of the already-completed functions in the Friendbook ADT.

`Map.h`

`Map.h` defines the i[illegible] Map ADT, which is used to map people's names to IDs. An im[illegible] note is that the Map ADT is not strictly necessary - it is only [illegible]ency reasons. If we didn't have access to the Map ADT and wanted to know the ID of a particular person, we could simply scan the `names` array until we found the index containing that person's name, and their ID would be that index.

`Queue.h`

`Queue.h` defines the interface to the Queue ADT. The Queue ADT is currently not used, but should be used in the optional task if you decide to do it.

Task 1 - Friending :) 👬

Implement the `FbFriend()` function in `Fb.c`, which takes the names of two people, and friends them if they are not already friends. The function should return true if the people were not friends and were successfully friended, and false if the two people were already friends.

You should implement your function so that the adjacency lists are kept in ascending order. For example, in the example in the Background section, since Harry (0) is friends with Ron (1) and Hermione (2), 1 should appear before 2 in Harry's adjacency list.

You can assume that the two people exist and are not the same person.

HINT:

HINT:

Is inserting into an [illegible] similar to a problem that you have solved in a previous lab exe[illegible]

Once you think you've got the function working, test it by recompiling with `make` and running the `runFb` program. You can test whether your function works using the `+` (add person), `f` (friend) and `s` (friendship status) commands. For example:


```
$ ./runFb
Friendbook v1.0
Enter ? to see the list of commands.
> + Harry
Harry was successfully added to Friendbook!
> + Ron
Ron was successfully added to Friendbook!
> + Hermione
Hermione was successfully added to Friendbook!
> s Ron Hermione
Ron and Hermione are not friends.
> f Ron Hermione
Successfully friended Ron and Hermione!
> s Ron Hermione
Ron and Hermione are friends.
> s Harry Ron
Harry and Ron are not friends.
> f Harry Ron
Successfully friended Harry and Ron!
> s Harry Ron
Harry and Ron are friends.
> s Ron Hermione
Ron and Hermione are friends.
```

You can run the above test case with the following command:

```
$ ./runFb -e < tests/friend-1.txt
```

This command runs the program and causes it to read Friendbook commands from the `friend-1.txt`. The `-e` option (which stands for "echo") causes the program to print out the commands to the terminal as they are being run, which makes it easy to see which operations are being performed.

When you're certain that the function works correctly, determine its worst case time complexity and write in `analysis.txt` along with an explanation. The time complexity should be in terms of n, where n is the total number of people.

Task 2 - Counting Friends

Implement the `FbNumFriends()` function in `fb.c`, which takes the name of a person and returns the number of friends they have. You can assume that the person exists.

Once you think you've got the function working, test it by recompiling with `make` and running the `runFb` program. Here is an example test case:

```
$ ./runFb
Friendbook v1.0
Enter ? to see the list of commands.
> + Harry
Harry was successfully added to Friendbook!
> + Ron
Ron was successfully added to Friendbook!
> + Hermione
Hermione was successfully added to Friendbook!
> f Harry Ron
Successfully friended Harry and Ron!
```

```
> n Harry
Harry has 1 friend
> n Ron
Ron has 2 friend
> n Hermione
Hermione has 1 f
```

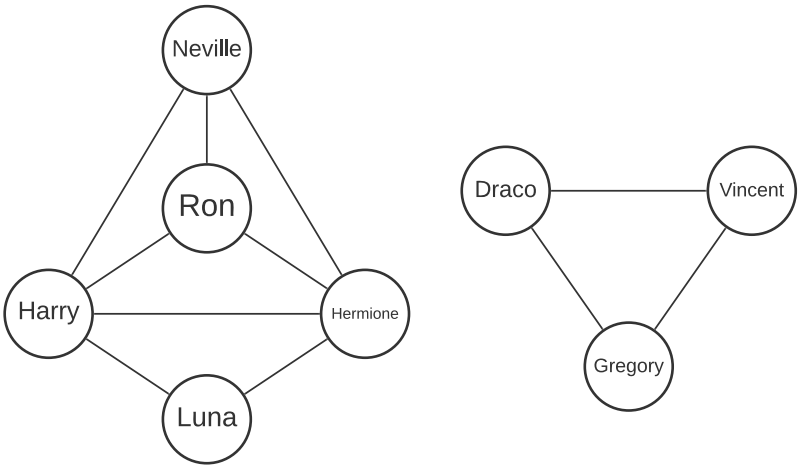
The Friendbook con[illegible] this example have been stored in the file `tests/num-friends-1.txt`, so you can also run this test case with the following command:


```
$ ./runFb -e < tests/num-friends-1.txt
```


When you're certain that the function works correctly, determine its worst case time complexity and write in `analysis.txt` along with an explanation. The time complexity should be in terms of n, where n is the total number of people.

Task 3 - Finding Mutual Friends

Implement the `FbMutualFriends()` function in `Fb.c`, which takes the names of two people, and returns a list of all their mutual friends. A person is a mutual friend of two people if that person is friends with both of those people. To illustrate this, here is an example:



Draco have no mutual friends.

You can assume that the two people exist and are not the same person. The mutual friends may be in the list in any order.

HINT:

To find out how to create a list and add names to it, see the comments in `List.h`, or read one of the existing functions in `Fb.c` that use the List ADT.

Once you think you've got the function working, test it by recompiling with `make` and running the `runFb` program. You can run the following command to test the scenario given above:


```
$ ./runFb -e < tests/mutual-friends-1.txt
```


When you're certain that the function works correctly, determine its worst case time complexity and write in `analysis.txt` along with an explanation. The time complexity should be in terms of n, where n is the total number of people.

Task 4 - Unfriending :(🧍💢🧍

Implement the `FbUnfriend()` function in `Fb.c`, which takes the names of two people, and unfriends them if they are friends. The function should return true if the people were friends and were successfully unfriended, and false if the two people were not friends (and so they could not be unfriended).

You can assume that the two people exist and are not the same person.

Once you think you've got the function working, test it by recompiling with `make` and running the `runFb` program. Here is an example test case:

```
Enter ? to see the list of commands.
> + Harry
Harry was successfully added to Friendbook!
> + Ron
Ron was successfully added to Friendbook!
> + Hermione
Hermione was successfully added to Friendbook!
> f Harry Ron
Successfully friended Harry and Ron!
> f Ron Hermione
Successfully friended Ron and Hermione!
> s Harry Ron
Harry and Ron are friends.
> s Ron Hermione
Ron and Hermione are friends.
> u Harry Ron
Successfully unfriended Harry and Ron!
> s Harry Ron
Harry and Ron are not friends.
> s Ron Hermione
Ron and Hermione are friends.
> n Harry
Harry has 0 friends.
```


The Friendbook commands used in this example have been stored in the file `tests/unfriend-1.txt`, so you can also run this test case with the following command:

```
$ ./runFb -e < tests/unfriend-1.txt
```

No time complexity analysis is required for this task.

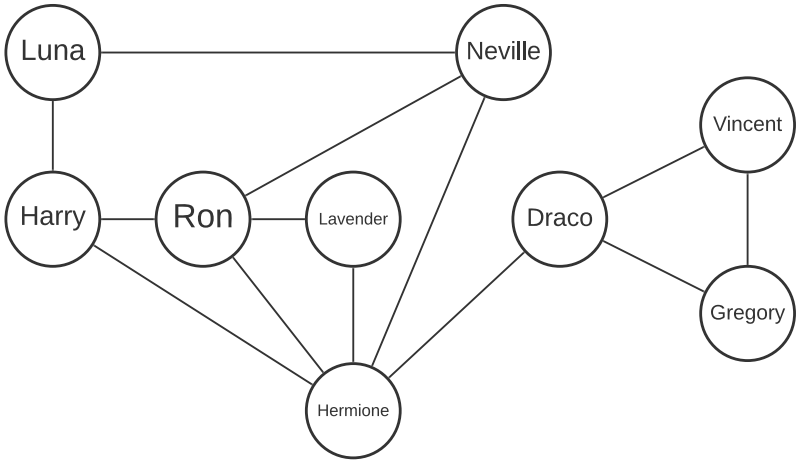
Implement the `FbFriendRecs1()` function in `Fb.c`, which takes the name of a person and finds friend recommendations for them. The function should store the recommendations in the given `recs` array and return the number of recommendations stored in the array.

The function should find people who are friends of friends of the person. In other words, it should only recommend people who share at least one mutual friend with the person. Obviously, it should not recommend someone who is already the person's friend.

Each recommendation consists of the name of the person being recommended and the number of mutual friends they share with the given person.

The recommendations should be sorted in descending order on the number of mutual friends shared, since someone with more mutual friends is more likely to be known by the person, and is therefore more likely to be added as a friend. If two people share the same number of mutual friends, they may be sorted in any order.

For example, consider the following scenario:



If `FbFriendRecs1()` is called with the name "Harry", the following output should be produced:

```
Harry's friend recommendations:
        Neville                 3 mutual friends
```

Explanation: Neville should be recommended first as he shares three mutual friends with Harry: Luna, Ron and Hermione. Lavender should be recommended next as she shares two mutual friends with Harry: Ron and Hermione. Draco should be recommended last as he shares just one mutual friend with Harry: Hermione. There is no typo on the last line - it is left as "friends" for simplicity.

Once you think you've got the function working, test it by recompiling with `make` and running the `runFb` program. You can run the following command to test the scenario given above:


```
$ ./runFb -e < tests/friend-recs-1.txt
```


When you're certain that the function works correctly, determine its worst case time complexity and write in `analysis.txt` along with an explanation. The time complexity should be in terms of n, where n is the total number of people.

Optional Task

> **NOTE:**
>
> This task is **optional**. It is not worth any marks.

Implement the `FbFriendRecs2()` function in `Fb.c`, which takes the name of a person and finds friend recommendations for them. The function should return a list containing the names of all the people being recommended, with the names being ordered as described below.

Unlike `FbFriendRecs1`, this function should recommend all people who are reachable from the given person via friendship links (not just people who share a mutual friend), and should recommend people who are "closer" to the person

recommend someone who is already the person's friend. If multiple people are the same "distance" from the person, they can be recommended in any order.

Limit the number of recommendations to 20 to avoid generating too many recommendations.

For example, consider the same scenario as in Part 2:

If `FbFriendRecs2()` was called with the name "Luna", the following is one possible valid output:


```
Luna's friend recommendations:
        Ron
        Hermione
        Draco
        Lavender
        Vincent
        Gregory
```

Explanation: Ron and Hermione are the closest people to Luna who are not also her friends, so they are recommended first. The example output recommends Ron first and then Hermione, but it would be equally valid to recommend Hermione first and then Ron. Draco and Lavender are the next furthest away, so they are recommended next. It would be valid to recommend Lavender before Draco. Vincent and Gregory are the next furthest away, so they are printed next. Once again, it would be valid to recommend Gregory before Vincent.

HINT:

You will need to us rsal algorithm to complete this task. But which one? You ca aph traversal algorithms here, and then follow the pseudo osen algorithm.

Submission

You need to submit two files: `Fb.c` and `analysis.txt`. **You must submit all of these files, even if you did not complete all of the tasks.** You can submit via the command line using the `give` command:


```
$ give cs2521 lab05 Fb.c analysis.txt
```


You can also submit via give's web interface. You can submit multiple times. Only your last submission will be marked. You can check the files you have submitted here.

WARNING:

After you submit, you **must** check that your submission was successful by going to your submissions page. Check that the timestamp is correct. If your submission does not appear under Last Submission or the timestamp is not correct, then resubmit.

Most of the marks for this lab will come from automarking. To receive the rest of the marks, you must show your work to your tutor during your Week 5, 7 or 8 lab session. You will be marked on the following criteria:

Aspect	Marks	
Code correctness	4	Tests for each task will be automarked. Automarking will be run after submissions have closed. After automarking is run you will be able to view your results here.
Complexity analysis	1	This mark is based on how accurate you were with your time complexity analysis and the quality of your explanations in `analysis.txt`. Analysis of each function is worth 0.25 marks.