CS130A Final Project : Data Structures for Solving Analytical Queries on Social Networks
Demo Day: Tuesday June 8, 2021

May 11, 2021

1 Introduction

Online Social networks connect people from around the globe and are being increasingly used to collaborate on ideas and opinions. Graphs are usually used to represent and model these networks, where nodes represent users and edges show the interconnections among these users (friendship relationship). In this assignment we will solve a search problem modelled on a social networking application. This networks contains a given number of users, with each user having a certain number of friends. An edge between Jane and Mary, means that Jane and Mary are friends (We are considering an undirected graph as in Facebook, and edge between Jane and Mary means that Jane is a friend of Mary and Mary is a friend of Jane).

Graphs can be implemented either using an adjacency list or an adjacency matrix representation. For social network graphs, we usually use the adjacency list representation. This is because the input graphs can have as many as 10^7 nodes, and in such scenarios the adjacency matrix representation needs an excessive amount of memory $O(n^2)$ and it can simply exceed memory limitations of the machine.

In addition to a graph, which represents information about friendship, we are also interested in *Profile Data* which maintaining profile information about each user in the graph. This information will be stored in a file. Each user will have a single *record* in the file. All records have the same format, and the same type of information, namely, name, age and occupation.

To support queries, we would like you to use a **Red-Black Tree** that uses name information. In particular, we would like to support two very different types of queries:

1. Exact match queries on a user's **unique** name, i.e, find information about user friends, e.g, their names, ages and occupations. In this case, we will need to search on the tree to find a pointer to the corresponding adjacent list and then figure out the friends age and occupation from the stored *Profile Data* file, thus allowing random access inside the file with O(log(n)) access time.

2. Range queries on the name attribute, for example retrieve the names and occupations of all people whose name is between *Amr* and *Natalie*.

Finally, you need to ensure that the records about the profiles of the users are persistent. Hence the file with user profiles must be stored on disk. For simplicity, we will not require the friendship graph itself or the Red-Black tree to be stored on disk, but any pointers to the Profile Data should point to the records in the file. More precisely, a pointer to a record in file is merely an integer indicating the record location in the file.

2 Required Functionality

You have to implement and maintain the following data structures:

- A *Profile Data*, which has a *name*, *age*, and *occupation* for each user. The Profile Data is stored in a file on disk.
- A *Red-Black* tree (in memory) on the *name* attribute of the Profile File. It needs to support insert, exact match search and range search queries (**deletion is not required**). Recall *name* is unique and use *strcmp* function to compare names.
- A *Friendship Graph* (in memory) where each graph node comprises of the name of a user (assume each name is unique), edges to a list of all of his/her friends. The graph should support insert and search for friends queries.

Possible Queries: Your program should be able to answer 2 types of queries:

1. *Friendship queries*, such as Find the occupation of all of Mike's friends. This queries should first traverse the red-black tree **then** move on to the graph to retrieve all friends of Mike, and finally go to the Data Profile and retrieve for each friend, their occupation.
2. *Range queries* on **name**, such Find the occupations of all users between *Fuheng* and *Sean*. In this case, the query should traverse the red-black tree to find all users between Fuheng and Sean, and for each one retieve from the Data Profile their occupation. In addition for the purposes of the demonstration and debugging, you should support *PrintAll*, which prints all users in the system with *all* their information, including name, age, occupation as well as a list of all their friends.

Possible Updates: Your program should be able to answer 3 types of update operations:

1. *Initialization:* Given an input file that includes a list of users, their attributes, i.e., their name, age, occupation, and a list of their friends' names, initialize the adjacency list, the red-black tree, and the Profile Data file
2. *Insert a single new user:* Insert a new user to the network e.g. Insert Mic 25 "Student at UCSB".
3. *Insert a Friendship:* Add a Friendship relationship between 2 users: AddFriend Mic Jane, **where Mic and Jane have to be already in the network**.

3 Implementation Details

3.1 Friendship Graph

The graph that models the social network graph should be implemented using a *generalization* of an adjacency list. An adjacency list is an array of a defined type (for instance GraphNode) that has a key, a linked list of friends and a pointer to the corresponding entry in the Profile Data in the file. Please see Figure 1 for an illustration of the relationships. Note you could also establish a relationship between the graph and the Profile Data file. Furthermore, note we are **not** asking you to support the *delete* operation.

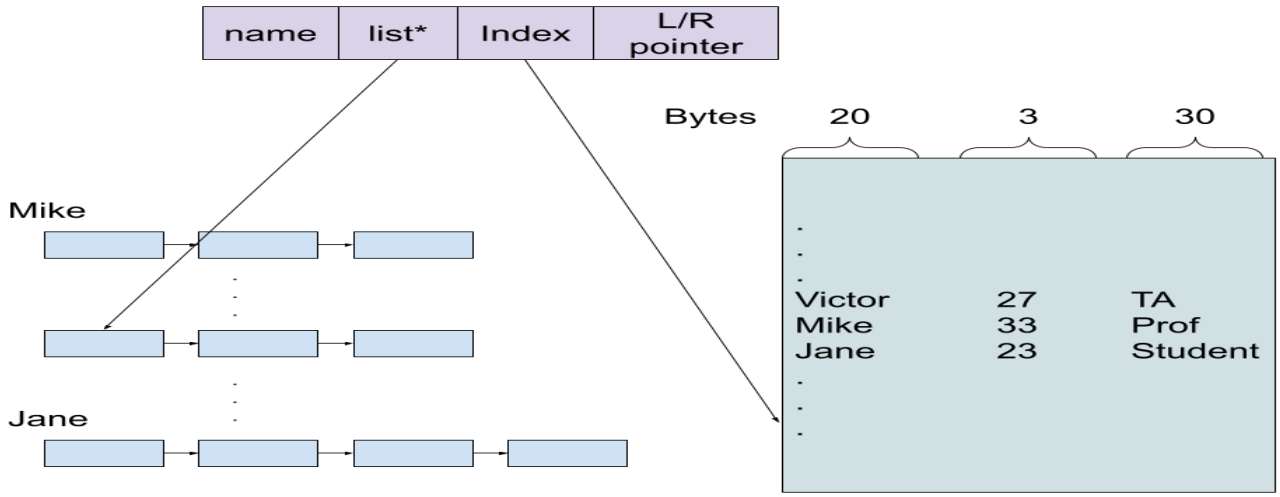

Figure 1: Friendship Graph architecture: The tree node contains a pointer to the list, an integer index of person's information on disk, and left/right pointers to its children.

3.2 Profile Data

The data records are stored in a file on disk. All records have the same size and format. Therefore, you can use the following structure for your file for each user:

```
char[20] name;
char[3] age;
char[30] occupation;
```

Note that since the size of each record is known, you can simply seek to any record you want in the file on the hard disk.

Hint: check *istreamseekg(streamoff offset, ios_base::seekdir dir)* function for seek, and also *setw* function can be helpful for padding.

A Red-Black Tree is defined on the **unique** *name* attribute, which means that the key is *name*. Hence, we should be able to search for one specific name as well as a range of names

(for instance Hilda to Yuval inclusive). See Figure 2 for an example of such a Red-Black tree. It has 12 nodes, corresponding to 12 people. Each tree node contains the *name* of a user and a **pointer** (index) into the Profile Data file (on disk) for this user's record.

For easiness in debugging, the output of *PrintAll* function should be similar to the following, each user's information on a separate line.

Mary,21,Developer,Jane,Alex,Ben

Alex,40,School Teacher,Mary,Mike

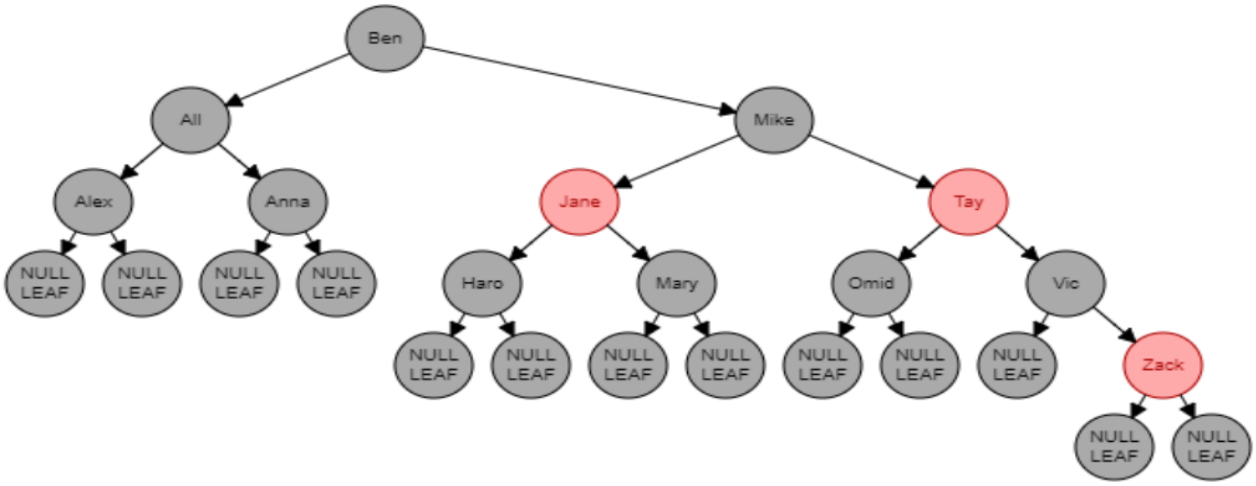

Figure 2: Red Black Tree sample structure

3.3 Handin

- For this project, you can work in groups of 2 students. Groups should demo their project on the course's final demo date, **June 8 2021**. Hence, there is no need to enforce a single input and output format.

Minimum Requirements Summary:

- Your program should be able to initialize the adjacency list, the red-black tree, and the Profile Data file from an input file that includes a list of users, their attributes, i.e., their name, age, occupation, and a list of their friends' names. The format will be simple, a list (separated by comma) of
  $name, age, occupation, friend1, friend2, friend3 \cdots$ each on a single line. Note that name and occupation potentially can have spaces.
- You will also use a similar file to initialize your program during the demo.
- Insert a new user to the network e.g. Insert Mic 25 "Student at UCSB".

- Add a Friendship relationship between 2 users: AddFriend Mic Jane, **where Mic and Jane have to be already in the network**.
- PrintAll
- List Names Age Occupation of friends of some user. ListFriendsInfo Omid, **where Omid has to be already in the network**
- List all users' information with names e.g. Alice $\leq$ Name $\leq$ Bob, ListInfo *lowerBound upperBound*, where lowerBound and upperBound are names that may or may not be in the system.
- There is no defined structure for queries; thus, you can define your own syntax. Your software is supposed to at least handle these queries but you can add more queries and add more data structures to make it faster. Your creativity will be greatly appreciated and highly encouraged.
- It is worth saying that we tried not to limit you in this project. This is the final project and you can use your creativity, programming skills and even graphical interface design as much as you want :)