Data Structures for Social Media Platform

ELEC278

Prepared by Kaden Neild

20405715

Presented to

The Smith Faculty of Engineering and Applied Science

Queen's University

November 20th , 2024

.

Table of Contents

List of Tables

List of Figures

1. User Profile Management and Search Functionality

For the user profile management system, the decision was to make an AVL tree to store the all the users within the program. The purpose of this AVL tree is to provide many benefits in organization, time complexity and structure.

Table 1: AVL Tree Pros and Cons Table

AVL Tree Data Structure	
Pros	Cons
Compared to other trees, this data structure does not allow the height to go above one, which allows for faster searching.	Since one of the properties of an AVL tree is to stay balanced, it requires additional functions in the code to maintain the property.
Considering that it's self-balancing, the tree itself will always be logarithmic (Olog(n)) which allows the tree to be fast in searching (Dependent on the value in which the tree is organized), insertion and deletions.	Since it requires rotations, it was a little bit more difficult to implement and code and also debugging logic errors may be a little bit more difficult as well.
	Since it needs to self – balance, it requires a height integer for each node. Over-time this may consume a bit more memory depending on how large the tree is.

Here is the implementation of the AVL tree in the User Struct type. It stores a unique ID, name, email, a left and right pointer for the tree, the height of each node, an array to store the friends, integer for the number of friends.

Figure 1: User Management Structure

Going through each of the functions given, the time complexity varies on the data of searching required. Since the AVL tree is stored based alphabetically on the user's name, searching by name would be Olog(N) for time complexity. However, searching through for an email would make the AVL become a regular binary tree, and the time complexity would become worse being O(n). Similarly, changing the username or email would be Olog(N) because for both functions, searching through the tree with a name is much faster. Lastly, the delete user function allowed the AVL tree to re-balance itself after going through the different cases of each node so that the tree can maintain the Olog(N) property.

2. Friends System

For the next section of this assignment, it would be most practical to use a dynamic array to store the list of friends in each node of the AVL tree. This allows for the user to have as many friends as needed whilst not wasting memory in the process. The initialization of the friend system can be seen in Figure 1.

Table 2: Dynamic Array Pros and Cons Table

Dynamic Array Structure	
Pros	Cons
Considering the unknowing factor of deciding on how many friends a user can have; it becomes easy to manage the size and memory usage. If the friend array becomes full, it will multiply its size by 2.	In searching, using a dynamic array to iterate though can become O(n) which is not the best for time compared to other structures.
Since one of the requirements is to print the friends alphabetically, it makes sorting the array easier than different data structures like linked lists.	A dynamic array requires additional memory to keep track of its size which may increase the amount of memory usage dependent of the number of users.
Over-time with a very large number of users, having a dynamic array implemented will provide better efficiency since time complexities of resizing is O(1). Also searching an element with a given index is O(1) as well.	

Moving on, the functions that had to be implemented with this structure were adding a friend, deleting a friend, printing friends, mutual friends and printing mutual friends. Firstly, adding a friend was simple and not too complicated to implement, using iteration and string copy. If the array becomes full when inserting a user, the program will double the size of it. In addition, with the bubble sort algorithm, it can

re-sort itself after every user is added. Unfortunately, the worst-case for bubble sort would be O(n^2), which would slow down the program briefly. For deletion, since dynamic arrays indexes cannot be removed very easily, the program shifts the index so that that user can now be deleted. For printing mutual friends and friends, the program will just iterate over the array using the amount of friend's integer to know when to stop. Lastly, for the mutual friend's array, it was implemented to iterate through both friends lists using a double for loop and see if a value matches in both and stores it sequentially. It is important to mention that all these functions require O(n) time complexity except adding a friend at worst-case, which is not the best in terms of efficiency, however, it makes the program simpler and easier to maintain.

3. Messaging System

The messaging system for this assignment was implemented using two different data structures. For the database of the chats, a linked list was used whilst for storing the actual chat logs, this was implemented using a queue. The reason being was to keep a global database to store all the chats using the sender's and receiver's respective IDs to create the chat.

Table 3: Pro and Cons Table for Linked Lists

Linked List Structure	
Pros	Cons
No wasted memory for linked lists. This means that it always uses the amount it needs to display the relevant data. For example, the chats will only delete off the linked list when one of the user's is deleted.	Similar to an array, it uses O(n) time to iterate through each node of the list until it finds the desired value. For this assignment, it would iterate until it finds both of the user's IDs.
Insertions and deletions are usually efficient, using O(1) time complexity if the pointer to the node is already known.	Increased memory size, since you are storing an additional pointer for every node. Could use up significant memory dependent on the amount of users present.

Table 4: Pros and Cons Table for Queues

Queue Structure	
Pros	Cons
Obeys the property of FIFO. Helpful for the assignment because the messages must be printed out in FIFO.	Searching through a queue, would be O(n) time complexity.

Enqueue operations require only O(1) time complexity because of the iterations of the front and rear integers.	

Using both data structures, it was efficient to store the data referring to the restrictions. Here is the full structure of the queues and linked list implemented in my project.

```
// Messaging system
typedef struct {
    // add attributes
    int messageID[MAXCHATLOG]; // unique message id
    char *sender[MAXCHATLOG]; // the sender for every message
    char *message[MAXCHATLOG]; // the actual content of the message
    int front;
    int rear;
    int user1Id;
    int user2Id;
    struct Message *next;
} Message;
```

Figure 2: Full structure of the messaging system.

Whilst using the rules as guidelines to implement the messaging system, it was apparent to use three different queues that all have a specific purpose. For example, each message consists of a unique message ID, which just iterates every time a new message is created but is linked to the other queues because they all use the same front and rear values. The sender is the name of the person that sent the messages, the name is needed for the display chat functions as it requires to print out the name of the person who sent that specific message. The user1 and user2 IDs are implemented here for easy searching through the linked list to ensure that it is the proper chat because creating a message.

The functions that needed to be implemented are the creation of the message, print message and display chat. For the creation of the message, it would first check to make sure that the users are friends before attempting to create a message. It simply checks the chat database (Linked List) to see if there's already an existing node, if there is, then it adds the message using enqueue at that node. If not, it creates a new node with the two user's IDs. The display chat functionality will check to see if a chat exists already, if it's unable to find anything, it will just return. If the chat does exist, it will use the simple print queue function to iterate through the queue in FIFO order using the rear and front integers.

4. Posts, Like System and Feed Generation

Lastly, the final data structure implemented in the project was a max heap. The main reason why a max heap was selected is to store all the posts based on the number of likes and then the recency. This allows for it to become almost like a priority queue, and it prints out the post with the most likes first.

Table 5: Pros and Cons Table for a Heap

Heap Data Structure	
Pros	Cons
Heaps have a time complexity of O(logn) for inserting and deleting elements. Since the project could have an indefinite number of posts, this could be extremely beneficial.	A heap is not the best for searching because it would have to iterate through the tree in O(n) time complexity. Depending on the number of posts, this could delay it significantly.
Priority based complete binary tree which allows the posts to be processed based on priority of likes as well as being space efficient.	A heap is not the most stable data structure because the order of equal element could be lost when the heap is constructed or modified.

To construct my global heap, a separate of two different structures were used to implement it properly. Here is a picture of the full implementation of both the heap structure and the post structure.

```
typedef struct {
    // add attributes
    int postID;
    char* poster;
    char* content;
    // Changed to ID to avoid user changing their names
    int* idPeople;
    int likes;
} Post;

// Max heap
typedef struct{
    Post** array;
    int size;
    int capacity;
} PostDatabase;
```

Figure 3: Structures for the post system of the project.

The first structure, Post, is structure for every post that is created. It has a "postID" which iterates every time a new post is created. This was used for the recency if the likes were the same. It includes the poster, the content, an array to store the people's IDs who liked the posts and the number of likes. The second structure, PostDatabase, is the actual heap which holds an array of posts with a size and capacity as well.

The project required three different functions to implement such as creating a post, adding a like to a specific post, and displaying the feed of the user. Firstly, creating the post was implemented using a global heap to store every post that was being created. Every post got created with the right content, poster and initially started at zero likes. Every time a new post was created, it would use the heapify function to resort

it based on the number of likes. Moving on to the liking system, it would first check if the users are friends or if it's the user wants to like its own post. It would then use a for loop to iterate through my array in the post structure so see if the user already liked the post by checking if their ID is already in the system. If all the checks are good, it simple allocates memory to the dynamic array and adds the ID of the user that liked the post and iterates the number of likes. Finally, the display feed function needs to prioritize the number of likes, then recency whilst also only showing their friends and themselves. To achieve this, I implemented a temporary copy of the database and used that to print out the post for the priority. The main reason why the duplication was appropriate was to ensure that the main database was not being tampered with or some future bugs could appear. Then it was to just simple print out the heap and make sure that the posts do not surpass 20 or else the eldest would be deleted. This was implemented also in my heapify functionality where if two posts had the same number of likes, it would then sort that based on the post ID.