DATA STRUCTURES AND ALGORITHMS

REPORT
FRIEND RECOMMENDATION SYSTEM

Data Structures Used

We have made use of various distinct data structures which have been taught in class but tweaked according to our needs for implementing the project. Data structures that have been used are:

1. Arrays
2. Dequeues
3. HashTables
4. Heaps

Arrays

The major role of arrays in our project has been to maintain the database of users. An array of pointers is used to store all users. These pointers point to structures which store all the relevant information about the users. This structure contains an integer variable storing the id of the user, a pointer to a structure that stores the parameters of the user, a boolean for checking the validity of the user, and two pointers to hash tables that store the friend lists.

HashTables

HashTables have been primarily to store the friend list of the users. The IDs of the friends are hashed by using the modulo operator with the current size of the hashtable. We have two hashtables in the UserNode: one which stores the friends of the user i.e all the users which are connected to this user by an out-edge and the other hash table stores the user IDs of those whom the concerned user is a friend of i.e. all the users which are connected to this user by an in-edge. The size of the hashtable is always maintained as a prime number. When the number of nodes in the hashtable exceeds the size of our hashtable we resize it again to a prime number.

Heaps

We have used a minheap to keep track of unused IDs. When the heap contains only one element i.e the root node, this indicates that all IDs after the value at the root node are unoccupied. When a user unregisters from the system, his ID is added to the minheap. Now when a new user registers, we assign his ID as the root node of the minheap which is the least positive integer that is unused. Now if the heap becomes empty after this operation we simply add the number next to the extracted number as it meant that there was only one node in the minheap and all IDs after that are free.

Dequeues

Dequeues has been implemented for performing Breadth-First Search i.e. BFS. While doing BFS we inject the out-vertices of the user in the dequeue and when all of them are done, we pop the users in the dequeues and inject their out-vertices.

Algorithms Used

Every function has a different algorithm associated with it. In our project, we have tried to minimize the time complexity taken by each algorithm as far as possible. Over here we will discuss the time complexities related to all the main functions.

Inserting a User

Inserting a user can be done in O(1) i.e. constant time complexity as we just allocate memory for the required things by directly accessing the ID no. in the Graph array. But after inserting a particular number of users the graph array may resize and this takes as time complexity of realloc which is either O(1) or O(n).

Inserting a Friend

Inserting a friend takes O(1) i.e. constant time complexity as we access the user in O(1) time, access the hash of the friend ID in O(1) time, and adding a node in the correct bucket of the table takes O(1) time as we have used separate chaining in the hashtable. After inserting a certain number of friends the hashtable may resize which takes time complexity of O(n).

Deleting a Friend

Deleting a friend is very similar to Inserting a friend and also takes O(1) time complexity as instead of adding a node, it deletes a node in the correct bucket of the correct table of the Graph array.

Deleting a User

Deleting a user takes O(k) time complexity where k is the average size of the friend list hashtable. For deleting a user we traverse through the in-vertices list of the user and delete the user from the out-vertices list of the users present in the in-vertices list of the user we are deleting. This means we are deleting the user as a friend from all those whom he is a friend of. Since deleting a friend takes O(1) time complexity we have to perform this for the size of the hashtable. A similar process will be repeated for the out-vertices list of the user to be deleted. We will traverse the out-vertices list of the user and delete him from the in-vertices list of the users present in that list. This means we are deleting the user from the in-vertices list of the users who are his friends.

Checking friendship status

Checking friendship status takes O(1) time complexity as we access the hashtable of the first user in O(1) time, access the hash value of the second user in O(1), and check whether the second user is present in the out vertices list of the first user in O(1) time.

Recommending friends to a new user

Recommending friends to a new user takes O(n) complexity where n is the size of the Graph Array. The arguments to the function are the Graph pointer and the ID of the new user who has registered. Then we check for the common number of parameters and store it in a struct(parametercount) which stores the Pointer to details and the count of common parameters. Then the array of structs (parameter count) is sorted w.r.t common count and then the top 10 (or less) are displayed.

Recommending friends to an old user

The time complexity of recommending friends to old use in the worst case is O(N) which happens when the user we are recommending has all users to be his friends. The time complexity in general is O(K + L) where K is the number of friends to be recommended and L is the number of direct friends of the user for whom we are recommending friends.

The basic idea of the algorithm is that we find users in a BFS manner which would yield us the recommendation list which proper preference, but not randomized…(partially randomized due to graph hash function)… Further randomization is achieved by book keeping the “preference level of each recommended friend” and then finding the segments where they have the same “preference level” and then randomizing that section.