UE21CS252A - Data Structures And Its Applications
Assignment 2 - Non-Linear Data Structures
Department of CSE - PES University

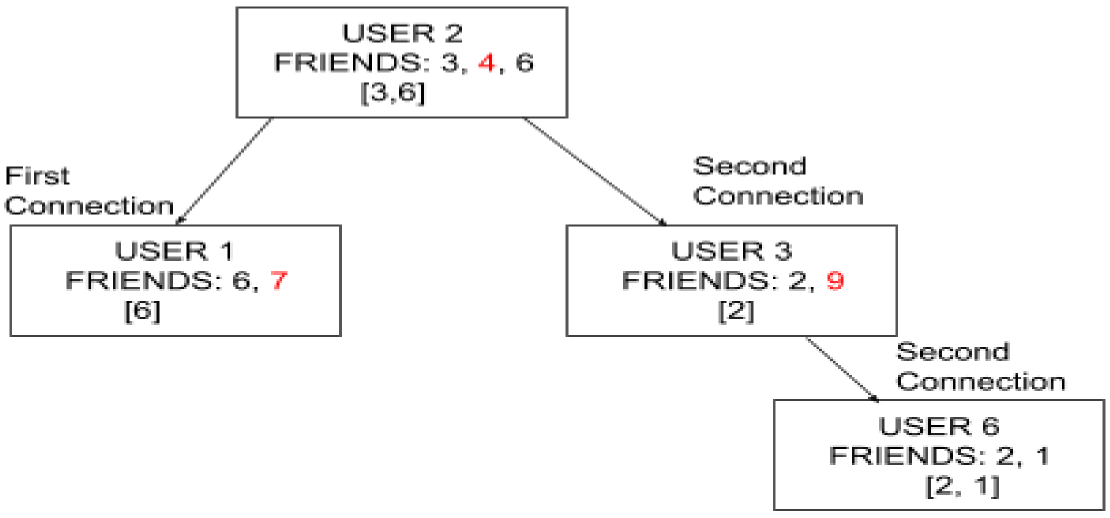
Date: 14/10/2022

Maximum Marks: 30

For this assignment, you'll be implementing a social network system, which consists of various users connected in a network of a non-linear structure. In this course, you've learnt that non-linear data structures are connected through pointers and are stored in non-contiguous locations. In the case of a network, a non-linear data structure is helpful, which helps you jump to various nodes without the need of knowing a certain fixed size of the data structure. Some of the important non-linear data structures you've learnt are trees and graphs. Let's learn more about this system to get a picture of what this network looks like.

Here are some of the logical units and terminologies of the system:

1. User: Essentially, a user is represented by a node in the data structure. This node contained information about the user, such as ID, name, number of friends, and a list of friends.
2. Connections: Users in a system are connected to one another based on the ID. The users do not have bidirectional connections. No connection forms a loop. Users are segregated in the system based on IDs, all users have first connections where they connect with users of ID less than their own and have second connections where they connect with users of ID greater than their own. This means that each user has at max two connections - first and second - or only one of those or none at all.
3. Friends: Each user has a list of friends or no friends at all. This list is a part of user information. Connections of the users may not necessarily be their friends. This means that a user has at most two connections but can have any number of friends. While the user can name any number of friends, only those friends who are a part of the network can be added and stored in the list. Here's a small diagram to help you understand the constraints of the system.



In the above diagram, you can understand the structure of the system based on the USER [ID] named in each node. Friends is a list that is initially given as input, the friend IDs marked as red are not present in the tree, hence the final list of friends stored (next line) contains only the friends that exist in the tree.

Choice of data structures for implementing the problem at hand:

For this system in question, the ideal data structure would be a binary search tree, as is evident by the constraints on the connection

Expected deliverables:

In a nutshell, the deliverables are **completing the implementations of given functions in 1 file.**

The structure of a node contains following members:

```
typedef struct node
{
    int id; //ID of user
    int numfren; //number of friends of user
    char name[MAX]; //name of user
    int* friends; //friends of user as an array
    struct node* right;  //user to the right
    struct node* left; //user to the left
} node;
```

The operations are as follows:

1 - INSERT NEW USER
2 - DELETE USER
3 - SEARCH
4 - PRINT FRIENDS OF
5 - PRINT INORDER
6 - EXIT

1. bst.c:
    a. Implementation file for all the basic operations for the system in question.
        I. **struct node* insertUser(struct node*root, int id, struct node* user);** Inserts users in the system based on the ID. **root** represents the root node where the system starts from, **user** represents the node to be inserted, and **id** represents the ID of the user to be inserted.
        If a user adds friends, none of which are present in the tree, then **-1** needs to be added in the friend list, and on finding a friend, -1 needs to be removed from the friends list.

        II. **struct node*refineUser(struct node*user, struct node *users);** This function is called before insertUser function (check main function). Here, if the **user** (to be inserted) has an ID that is already present in the tree, you increment the ID by 1 and check again, until the next nearest available ID is found and inserted. This is only for IDs that are repeated. The other requirement in this function is to update its friends list. If all the mentioned friends do not exist in the tree, the friend list for that user will contain -1 and number of friends will be 0. After updating friends, it will also insert itself in its friends (bidirectional friends)

For example,

Node1: ID - 12, friends - [8, 9]

Node2: ID - 3, friends - [9, 2]

Node3: ID - 6, friends - [12, 3, 1] **(newly inserted)**

Updated:

Node1: ID - 12, friends - [8, 9, 6]

Node2: ID - 3, friends - [9, 2, 6]

Node3: ID - 6, friends - [12, 3]

Friends of newly inserted node id=6, which are 12 and 3, also have 6 in their friends list now.

III. **struct node* search(int key, struct node *users);** Searches for a user in the tree and returns **NULL if not present** and with the node itself if present.

IV. **void friends(int id, struct node *users);** Print friends of user with ID=**id**. Print -1 if the number of friends of this user is 0.

V. **struct node *minValueNode(struct node *node);** Helper function to deleteNode which helps you find the next minimum node when deleting a node with two connections (finding inorder successor in BST)

VI. **struct node*deleteFriends(int key, struct node*users);** For each friend of the user with ID=key, delete itself from its friend's friend list.

For example, Node: ID - 2, Friends - [1, 3]

Node: ID - 1, Friends - [2]

Delete(1)

Updated: Node: ID - 2, Friends - [3]

Node: ID - 1 [DELETED]

VII. **struct node *deleteNode(struct node *root, int key);** Delete the user with ID = **key**, where **root** is the pointer to the root node of the tree. Returns the root of the updated tree. Root returns NULL if the tree is empty.

VIII. **void printInOrder(node* myusers);** Prints the IDs of all users in ascending order (inorder). **myusers** points to the root.

Advisory, but not mandatory: It would be better if you use helper functions rather than writing everything in the same function to avoid cluttering everything and enhancing the code readability.

Instructions:

1. **Do not take any inputs from STDIN** and **do not print anything to the STDOUT**. The operations implemented as a part of the libraries should handle all the cases well.
2. **Remove all the printf and scanf statements other than the one asked for.**
3. **Do not change any code already given**, especially the function prototypes and struct declarations.
4. We recommend you test your implementation on any Unix/Linux(Ubuntu)/macOS operating systems with either gcc or clang compilers of any versions. C compilers on Windows tend to bypass segmentation fault errors which plays a major role when dealing with linked lists and dynamic memory allocations.
5. Free the dynamically allocated memory wherever necessary. Avoid dangling pointers and memory leaks which may lead to segfaults.
   Link to the files: DSA-A2
   To execute files: gcc socialnet.c
   Test inputs: ./a.out < ip.txt
   To print output to a file: ./a.out < ip.txt > output.txt

10. For any queries that need to be resolved, please post them on the google sheets and the TAs will post an answer beside it as soon as possible. Please make sure that your query has not already been answered before in the sheet before posting it.
    The link: DSA21_A2_Queries
    There are 2 sheets. One for the RR campus and one for EC Campus. **Post your queries in your respective sheets** based on your campuses.
11. Make sure your code runs properly for the sample inputs provided. However, that does not mean it will fetch you full marks for this assignment. We will **run your code against more inputs and run a plagiarism check against all submissions**, both of which may affect the marks gained. We have provided you 2 sample input files, with sample output files containing expected output. Please make sure, your code gives the expected results, before submitting.
12. Submission links:
    a. RR Campus.
    b. EC Campus.
13. **Submit only 1 file** (without any compilation errors): socialnet.c after naming the file as **SRN.c** where SRN is in the format “PES1UG21CSXXX” or “PES2UG21CSXXX”.
    Eg: PES1UG21CS519.c or PES2UG21CS519.c
14. Only **valid** submission is: - SRN.c

Expected learning outcomes:

1. Cope with abstractions depicting real-world entities.
2. Choosing appropriate data-structures for the right design.
3. Writing reusable and modular code.
4. Strengthen memory management in C and avoid dangling pointers and memory leaks while using dynamic data-structures like linked lists.
5. Getting familiar with the various operations of a BST.
6. Rearranging the nodes in BST.

– end –