Assignment 1

Abstract Data Types
Data Structures and Algorithms
Due Date: 13 April, 2022

1 Doubly Linked List

For the first assignment you'll be working with pointers and structures. The very first task is to create a Doubly Linked List.

1.1 Doubly Linked List Structure

Create a file called *my_dll.h*. Define a structure called *my_dll*. The structure should have the following data members:

1. **root**: Pointer to the first node
2. Anything else which might help you in your implementation

1.2 Node Structure

Create a file called *node.h*. Define a structure called *node*. The structure should have the following data members:

1. **data**: An integer
2. **prev**: Pointer to the previous node
3. **next**: Pointer to the next node

1.3 DLL Operations

There will be a strict penalty for those who fail to follow this format.
Define interface for functions to be defined on *my_dll* in the *my_dll.h* header file:

1. **insert(list, x)**: Insert a node containing data x at the end of the linked list (i.e. *list* which should be of type *my_dll*).
2. **insert_at(list, x, i)**: Insert a node containing data x at position i (zero-indexed) in the given list.
3. **delete(list, i)**: Delete the node at position i of the given *list*.
4. **find(list, x)**: Return the position of the first occurrence of x, if x is not present in the list return -1. Return type should be *int*.
5. **prune(list)**: Delete all nodes at odd indexes in the given linked list.
6. **print(list)**: Print all the elements of the given linked list starting from index 0.

7. **print_reverse(list)**: Print all the elements of the given linked list in reverse order.

8. **get_size(list)**: Return the current size of the linked list. Return type should be *int*.

You may accept the argument *list* by reference or by value. You may also create any other helper functions if you find it necessary.

1.4 Function Definitions

Write the routines for all the above defined functions in *my_dll.c*.
You are not allowed to use arrays anywhere in this question. We'll be monitoring the memory usage on large test cases so please be careful.

1.5 Driver code

Write a C program which illustrates the above operations by building a good command line interface, i.e. ask the user which operations they would like to do in a doubly linked list of integers.

1.6 Sample Test Case

Input	Output
insert 1 insert 2 insert_at 3 1	
print	1 3 2
insert 4 insert 5 insert 6	
print_reverse	6 5 4 2 3 1
prune	
print	1 2 5

2 Complex Number ADT

The goal is to define and implement an ADT for n-dimensional complex numbers. A n-dimensional complex number is of the form $a_1 + ia_2 + ja_3 + ka_4 +$ .... upto n terms.

2.1 File Structure

The code must consist of three files as follows:

1. *complex.h* - Which defines the ADT and the function prototypes. The ADT should be called *complex*.
2. *complex.c*- Which implements the functions defined in Sec 2.2.
3. *main.c* - Which must use the ADT and perform various operations on input and produce output as specified later.

2.2 Functions

The functions to be implemented are as follows. Please create the function prototypes EXACTLY as specified:

1. Addition **add(a, b)**: Given 2 complex numbers a and b, return a complex number c such that
$$c_t = a_t + b_t \forall t \in \{1, 2, 3, ...n\}$$
2. Subtract **sub(a, b)**: Given 2 complex numbers a and b, return a complex number c such that
$$c_t = a_t - b_t \forall t \in \{1, 2, 3, ...n\}$$
3. Mod **mod(a)**: Given a complex number a, return a float x such that
$$x = \sqrt{\sum_{t=1}^{n} {a_t}^2}$$
4. Dot product **dot(a, b)**: Given 2 complex numbers a and b, return a float x such that
$$x = \sum_{t=1}^{n} a_t b_t$$
5. Cosine similarity **cos(a, b)**: Given 2 complex numbers a and b, return a float x such that
$$x = DOT(a, b)/(MOD(a) * MOD(b))$$

2.3 Input

The first line contains a string TYPE and an integer N, denoting the type of the operation that you are supposed to make and the dimension of the complex numbers respectively. If the TYPE is MOD, then the next line contains N floating point numbers, denoting the complex number. Else, the next 2 lines contain N floating point numbers each, denoting the complex numbers on which you are supposed to make the operations.

2.4 Output

- If the TYPE is DOT or MOD, then print a single floating point number, rounded off to 2 decimal places.
- Else, print N floats, denoting the final complex number you get after doing the operations.

2.5 Sample Test Cases

Input	Output
ADD 4 1.5 2.1 3 4.3 9.2 7 6 5	10.7 9.1 9 9.3
MOD 4 2 0 1 2	3.00

NOTE: TAs will evaluate each code manually for correct implementation. We'll be grading the correctness of your code by using our own driver code, hence it is extremely important you follow the given structure.

3 Music Player ADT

The goal of this question is to define and implement an ADT for a Music Player.

The music player should be able to add, delete and swap songs in the queue along with the ability to play them.

3.1 File structure

The code must consists of three files as follows:

1. *song.h* - Defines the ADT and the function prototypes.
2. *song.c* - Implements the ADT and the functions.
3. *musicplayer.h* - Defines the ADT and the function prototypes.
4. *musicplayer.c* - Implements the ADT and the functions.
5. *main.c* - Uses the ADT and calls the different functions implemented.

3.2 Data Types

You can add any other information that you find useful for the data types as well

3.2.1 Song

All the songs should be stored in this with the following information:

1. **Name**: The name of the song
2. **Artist**: The name of the artist of the song
3. **Duration**: Duration of the song in seconds

3.2.2 MusicPlayer

The music player should have the following data present in it:

1. **Current song**: The current song that is playing.
2. **Queue**: Stores the queue in which the songs should be played from.

3.2.3 BONUS: User

All data types and functions related to this will be considered for bonus marks. If you lose marks in any other segment of the assignment, this bonus can make up for that. Maximum marks will still be capped at 100.

The following user data must be stored:

1. **Name**: Name of the user
2. **MusicPlayer**: The music player of the user
3. **LikedSongs**: A list of songs liked by the user in the data structure of your choice. This list can be arbritarily long.

3.3 Functions

For each of the ADTs, implement the following functions:

3.3.1 Songs

Function name	Parameters	Return type	Return values	Description
makeSong	*char*, char*, float*	*Song**	instance of a song data type	Create a new song

3.3.2 MusicPlayer

Function name	Parameters	Return type	Return values	Description
createMusicPlayer	-	*MusicPlayer**	Instance of a music player data type	Create a new music player
addSongToQueue	*MusicPlayer*, Song**	*int*	0 if the operation is successful, 1 otherwise	Add song to the end of the queue of the music player
removeSongFromQueue	*MusicPlayer*, int*	*int*	0 if the operation is successful, 1 otherwise	Remove song at the given index of the queue
playSong	*MusicPlayer**	*int*	0 if the operation is successful, 1 otherwise	Start playing the song at the front of the queue and remove it from the queue
getCurrentSong	*MusicPlayer**	*Song**	Pointer to the song currently playing	Get the current song playing

3.3.3 User

Function name	Parameters	Return type	Return values	Description
createUser	*char**	*User**	Instance of a new user	Create a new user
addSongToQueueUser	*User*, Song**	*int*	0 if the operation is successful, 1 otherwise	Add song to the queue of the user
removeSongFromQueueUser	*User*, int*	*int*	0 if the operation is successful, 1 otherwise	Remove song at the given index of the queue of the user
playSongUser	*User**	*int*	0 if the operation is successful, 1 otherwise. The operation can be unsuccessful if the queue is empty	Same as *playSong* description in *MusicPlayer*

getCurrentSongUser	*User**	*Song**	Pointer to the song that is playing. Return NULL if no song is playing.	Get the current song user is listening to
addLikedSong	*User*, Song**	*int*	0 if the operation is successful, 1 otherwise. The operation can be unsuccessful if the song has already been liked	Add song to the user's liked songs
removeLikedSong	*User*, Song**	*int*	0 if the operation is successful, 1 otherwise. The operation can be unsuccessful if the song has not been liked	Remove song from the user's liked songs
userCompatibility	*User*, User**	*int*	The answer	Tell number of songs liked by both users

3.4 Driver Code

Write a C program which illustrates the above operations by building a good command line interface, i.e. ask the user which operations he/she would like to do for the user.

3.5 Evaluation

There will be manual evaluation for this problem. However, your function signatures, return types and return values should match exactly as the ones described above. We will be testing your implementations on our own driver codes as well. The functions themselves should not print anything.

3.6 Sample I/O

Input	Output
make_user user1 make_song song1 artist1 3.5 make_song song2 artist1 3 make_song song3 artist2 5 like_song user1 song1	
get_liked_songs user1	song1
add_queue user1 song1 add_queue user1 song2 add_queue user1 song3 play_song user1 play_song user1	
current_song user1	song2
make_user user2 like_song user2 song1	

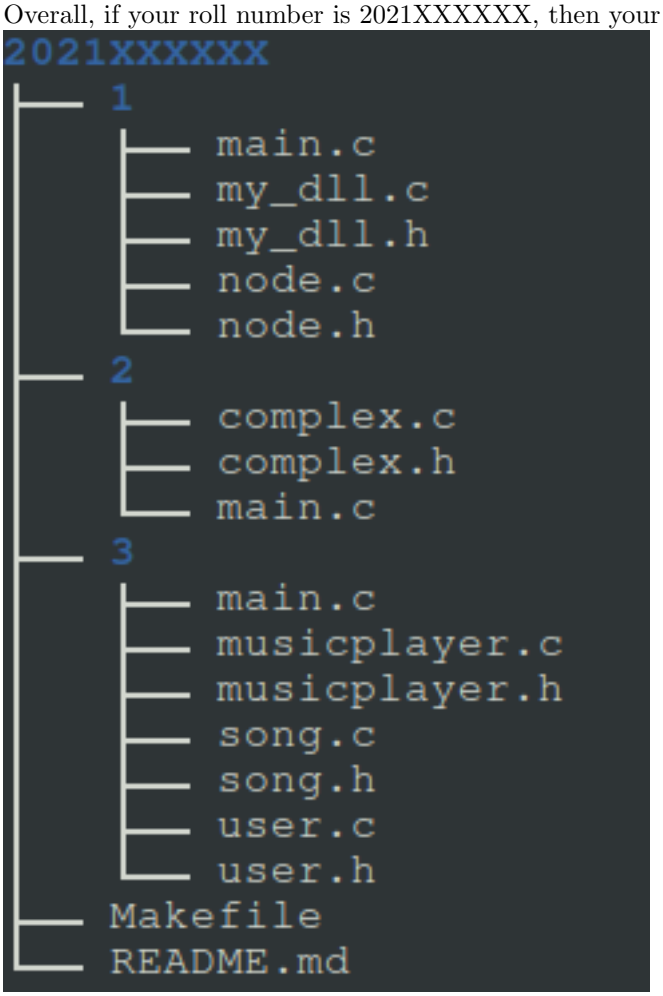
get_compatibility user1 user2	1

4 Submission format

Upload a zip file *roll_number .zip* which has a folder named as your roll number.

Inside the folder should be a *Makefile* to compile all 3 of your solutions, a *README.md* file containing assumptions and I/O choices taken by you while implementing the second and third problems and 3 folders named 1, 2 and 3 containing your codes for questions 1, 2 and 3 respectively.

Overall, if your roll number is 2021XXXXXX, then your file structure should look like:



You can include any other file that you require to this as well.