Ceng213 - Data Structures
Programming Assignment 1
A Simple Music Streaming Service Implementation via Linked Lists

Fall 2021

1 Objectives

In this programming assignment, you are first expected to implement a *circular doubly linked list* data structure, in which each node will contain the data and two pointers to the previous and the next nodes. The linked list data structure will include a head pointer that points to the first node of the linked list. The details of the structure are explained further in the following sections. Then, you will use this specialized linked list structure to implement a simple *music streaming service* application.

Keywords: *C++, Data Structures, Linked List, Circular Doubly Linked List, Music Streaming Service*

2 Linked List Implementation (50 pts)

The linked list data structure used in this assignment is implemented as the class template `LinkedList` with the template argument `T`, which is used as the type of data stored in the nodes. The node of the linked list is implemented as the class template `Node` with the template argument `T`, which is the type of data stored in nodes. `Node` class is the basic building block of the `LinkedList` class. `LinkedList` class has a single `Node` pointer in its private data field (namely `head`) which points to the first node of the linked list, and an integer (namely `size`) which keeps the number of nodes in the list.

The `LinkedList` class has its definition and implementation in *LinkedList.h* file, and the `Node` class has its in *Node.h* file.

2.1 `Node`

`Node` class represents nodes that constitute linked lists. A `Node` instance keeps two pointers (namely `prev` and `next`) to its previous and next nodes in the list, and a data variable of type T (namely `data`) to hold the data. The class has two constructors and the overloaded output operator (`operator<<`). They are already implemented for you. You should not change anything in file *Node.h*.

2.2 `LinkedList`

`LinkedList` class implements a *circular doubly linked list* data structure. Previously, data members of `LinkedList` class have been briefly described. Their use will be elaborated in the context of utility functions discussed in the following subsections. You must provide implementations for the following public member functions that have been declared under indicated portions of *LinkedList.h* file.

2.2.1 `LinkedList();`

This is the default constructor. You should make necessary initializations in this function.

2.2.2 `LinkedList(const LinkedList<T> &obj);`

This is the copy constructor. You should make necessary initializations, create new nodes by copying the nodes in the given `obj`, and insert those new nodes into the linked list.

2.2.3 `~LinkedList();`

This is the destructor. You should deallocate all the memory that you were allocated before.

2.2.4 `int getSize() const;`

This function should return an integer that is the number of nodes in the linked list (i.e., the size of the linked list).

2.2.5 `bool isEmpty() const;`

This function should return `true` if the linked list is empty (i.e., there exists no nodes in the linked list). If the linked list is not empty, it should return `false`.

2.2.6 `bool contains(Node<T> *node) const;`

This function should return `true` if the linked list contains the given `node` (i.e., any `prev`/`next` in the nodes of the linked list matches with `node`). Otherwise, it should return `false`.

2.2.7 `Node<T> *getFirstNode() const;`

This function should return a pointer to the first node in the linked list. If the linked list is empty, it should return `NULL`.

2.2.8 `Node<T> *getLastNode() const;`

This function should return a pointer to the last node in the linked list. If the linked list is empty, it should return `NULL`.

2.2.9 `Node<T> *getNode(const T &data) const;`

You should search the linked list for the node that has the same data with the given `data` and return a pointer to that node. You can use `operator==` to compare two `T` objects. If there exists multiple such nodes in the linked list, return a pointer to the first occurrence. If there exists no such node in the linked list, you should return `NULL`.

2.2.10 `Node<T> *getNodeAtIndex(int index) const;`

You should search the linked list for the node at the given zero-based `index` (i.e., `index`=0 means the first node, `index`=1 means the second node, ..., `index`=`size`-1 means the last node) and return a pointer to that node. If there exists no such node in the linked list (i.e., index is out of bounds), you should return `NULL`.

2.2.11 `void insertAtTheFront(const T &data);`

You should create a new node with the given `data` and insert it at the front of the linked list as the first node. Don't forget to make necessary pointer, `head`, and `size` modifications.

2.2.12 `void insertAtTheEnd(const T &data);`

You should create a new node with the given `data` and insert it at the end of the linked list as the last node. Don't forget to make necessary pointer, `head`, and `size` modifications.

2.2.13 `void insertAfterNode(const T &data, Node<T> *node);`

You should create a new node with the given `data` and insert it after the given `node` as its next node. Don't forget to make necessary pointer, `head`, and `size` modifications. If the given `node` is not in the linked list, do nothing.

2.2.14 `void insertAsEveryKthNode(const T &data, int k);`

You should create new nodes with the given `data` and insert them at the linked list as the every `k`th node. If the given `k` is smaller than 2, do nothing. Some example function calls are as follows:

- (1, 2, 3) → `insertAsEveryKthNode(4, 2)` → (1, 4, 2, 4, 3, 4)
- (1, 2, 3) → `insertAsEveryKthNode(4, 3)` → (1, 2, 4, 3)
- (1, 2, 3) → `insertAsEveryKthNode(2, 2)` → (1, 2, 2, 2, 3, 2)

2.2.15 `void removeNode(Node<T> *node);`

You should remove the given `node` from the linked list. Don't forget to make necessary pointer, `head`, and `size` modifications. If the given `node` is not in the linked list (i.e., the linked list does not contain the given `node`), do nothing.

2.2.16 `void removeNode(const T &data);`

You should remove the node that has the same data with the given `data` from the linked list. Don't forget to make necessary pointer, `head`, and `size` modifications. If there exists multiple such nodes in the linked list, remove all occurrences. If there exists no such node in the linked list, do nothing.

2.2.17 `void removeAllNodes();`

You should remove all nodes in the linked list so that the linked list becomes empty. Don't forget to make necessary pointer, `head`, and `size` modifications.

2.2.18 `void removeEveryKthNode(int k);`

You should remove every `k`th node from the linked list. If the given `k` is smaller than 2, do nothing. Some example function calls are as follows:

- (1, 2, 3, 4, 5) → `removeEveryKthNode(2)` → (1, 3, 5)
- (1, 2, 3, 4, 5) → `removeEveryKthNode(3)` → (1, 2, 4, 5)

2.2.19 `void swap(Node<T> *node1, Node<T> *node2);`

You should swap the two given nodes `node1` and `node2`. For this function, you are not allowed to just swap the data in the given nodes. Also, you are not allowed to create new nodes. Do the swapping by playing with the pointers in the nodes of the linked list. If either of the given nodes is not in the linked list, do nothing.

2.2.20 `void shuffle(int seed);`

You should shuffle the nodes of the linked list by swapping the nodes by applying the following algorithm:

```
for "i=0" to "i=size-1":
    swap the node at index i with the node at index (i*i+seed)%size
```

2.2.21 `LinkedList<T> &operator=(const LinkedList<T> &rhs);`

This is the overloaded assignment operator. You should remove all nodes in the linked list and then, you should create new nodes by copying the nodes in the given `rhs` and insert those new nodes into the linked list.

3 Music Streaming Service Implementation (50 pts)

The music streaming service in this assignment is implemented as the class `MusicStream`. `MusicStream` class has four `LinkedList` objects in its private data field (namely `profiles`, `artists`, `albums` and `songs`) with the types `Profile`, `Artist`, `Album` and `Song`, respectively. These four `LinkedList` objects keep the profiles, artists, albums and songs of the music streaming service. `Profile` class represents the users of the music streaming service. `Artist`, `Album` and `Song` classes represent the artists, albums and songs streamed on the music streaming service, respectively. `Playlist` class represents the playlists created by the users of the music streaming service. Playlists of a user are kept in the user's corresponding `Profile` object.

The `MusicStream`, `Profile`, `Playlist`, `Artist`, `Album` and `Song` classes has their definitions in *MusicStream.h*, *Profile.h*, *Playlist.h*, *Artist.h*, *Album.h* and *Song.h* files and their implementations in *MusicStream.cpp*, *Profile.cpp*, *Playlist.cpp*, *Artist.cpp*, *Album.cpp* and *Song.cpp* files, respectively.

3.1 `Song`

`Song` objects keep `name` variable of type `std::string`, and `duration` and `songId` variables of type `int` to hold the data related with available songs in the music streaming service. All of the functions of `Song` class are already implemented for you. You should not change anything in files *Song.h* and *Song.cpp*. Also, there is a `public static Song` variable with name `ADVERTISEMENT_SONG` in `Song` class. You will be asked to use it as the default `Song` object to put in the lists of songs as advertisements. Further information will be given in following sections.

3.2 `Album`

`Album` objects keep `name` variable of type `std::string` and `albumId` variable of type `int` to hold the data related with available albums in the music streming service. They also keep linked lists of pointers to the songs of the albums (namely `songs`). `Song` pointers in the `songs` linked list are pointers to the `Song` objects stored in the linked lists in `MusicStream` class. Most of the functions of `Album` class are already implemented for you. In *Album.cpp* file, you need to provide implementations for following functions declared under *Album.h* header to complete the assignment. You should not change anything in file *Album.h*.

3.2.1 `void addSong(Song *song);`

Add (append) the given `song` to the list of songs of this album. For this function, you may assume that the given `song` does not already exist in the list of songs of this album.

3.2.2 `void dropSong(Song *song);`

Remove the given `song` from the list of songs of this album. For this function, you may assume that the given `song` already exists in the list of songs of this album and there are no multiple occurrences.

3.3 `Artist`

`Artist` objects keep `name` variable of type `std::string` and `artistId` variable of type `int` to hold the data related with available artists in the music streaming service. They also keep linked lists of pointers to the albums of the artists (namely `albums`). `Album` pointers in the `albums` linked list are pointers to the `Album` objects stored in the linked lists in `MusicStream` class. Most of the functions of `Artist` class are already implemented for you. In *Artist.cpp* file, you need to provide implementations for following functions declared under *Artist.h* header to complete the assignment. You should not change anything in file *Artist.h*.

3.3.1 `void addAlbum(Album *album);`

Add (append) the given `album` to the list of albums of this artist. For this function, you may assume that the given `album` does not already exist in the list of albums of this artist.

3.3.2 `void dropAlbum(Album *album);`

Remove the given `album` from the list of albums of this artist. For this function, you may assume that the given `album` already exists in the list of albums of this artist and there are no multiple occurrences.

3.4 `Playlist`

`Playlist` objects keep `name` variable of type `std::string`, `shared` variable of type `bool` and `playlistId` variable of type `int` to hold the data related with the playlists belonging to the users of the music streaming service. They also keep linked lists of pointers to the songs in the playlists (namely `songs`). `Song` pointers in the `songs` linked list are pointers to the `Song` objects stored in the linked lists in `MusicStream` class. Most of the functions of `Playlist` class are already implemented for you. In *Playlist.cpp* file, you need to provide implementations for following functions declared under *Playlist.h* header to complete the assignment. You should not change anything in file *Playlist.h*.

3.4.1 `void Playlist::addSong(Song *song);`

Add (append) the given `song` to the list of songs in this playlist. For this function, you may assume that the given `song` does not already exist in this playlist.

3.4.2 `void Playlist::dropSong(Song *song);`

Remove the given `song` from the list of songs in this playlist. For this function, you may assume that the given `song` already exists in this playlist and there are no multiple occurrences.

3.4.3 `void Playlist::shuffle(int seed);`

Shuffle the list of songs in this playlist. For shuffling, use `shuffle(int seed)` function of the `LinkedList` class with the given `seed`.

3.5 `Profile`

`Profile` objects keep `email` and `username` variables of type `std::string`, and `plan` variable of type `SubscriptionPlan` to hold the data related with the users of the music streaming service. `SubscriptionPlan` is an enumerated type defined in *Profile.h* file with values `free_of_charge`, which means the user does not pay for the service and will listen to advertisements between songs, and `premium`, which means the user pays for the service and will not listen to any advertisement songs. A `Profile` object also keeps linked lists of (1) pointers to the profiles that the user follows (namely `following`), (2) pointers to the profiles that follows the user (namely `followers`), and (3) the playlists belonging to the user (namely `playlists`). `Profile` pointers in the `following` and `followers` linked lists are pointers to the `Profile` objects stored in the linked lists in `MusicStream` class. Most of the functions of `Profile` class are already implemented for you. In *Profile.cpp* file, you need to provide implementations for following functions declared under *Profile.h* header to complete the assignment. You should not change anything in file *Profile.h*.

3.5.1 `void followProfile(Profile *profile);`

This function makes this user (i.e., profile) follow the given `profile`. For this function, you may assume that this user is not already following the given `profile`.

3.5.2 `void unfollowProfile(Profile *profile);`

This function makes this user unfollow the given `profile`. For this function, you may assume that this user is already following the given `profile`.

3.5.3 `void createPlaylist(const std::string &playlistName);`

This function creates a new playlist with `playlistName` for this user.

3.5.4 `void deletePlaylist(int playlistId);`

This function deletes the playlist with `playlistId` of this user. For this function, you may assume that this user has a playlist with `playlistId`.

3.5.5 `void addSongToPlaylist(Song *song, int playlistId);`

This function adds the given `song` to the playlist with `playlistId` of this user. For this function, you may assume that this user has a playlist with `playlistId` and the given `song` is not already in the playlist.

3.5.6 `void deleteSongFromPlaylist(Song *song, int playlistId);`

This function removes the given `song` from the playlist with `playlistId` of this user. For this function, you may assume that this user has a playlist with `playlistId` and the given `song` is already in the playlist.

3.5.7 `Playlist *getPlaylist(int playlistId);`

This function gets (i.e., returns a pointer to) the playlist with `playlistId` of this user. For this function, you may assume that this user has a playlist with `playlistId`.

3.5.8 `LinkedList<Playlist *> getSharedPlaylists();`

This function gets a linked list of (i.e., returns a linked list of pointers to) the playlists shared by the users who are followed by this user.

3.5.9 `void shufflePlaylist(int playlistId, int seed);`

This function shuffles the songs in the playlist with `playlistId` of this user. For shuffling, use `shuffle(int seed)` function of the `LinkedList` class with the given `seed`. For this function, you may assume that this user has a playlist with `playlistId`.

3.5.10 `void sharePlaylist(int playlistId);`

This function marks the playlist with `playlistId` of this user as shared. For this function, you may assume that this user has a playlist with `playlistId`.

3.5.11 `void unsharePlaylist(int playlistId);`

This function marks the playlist with `playlistId` of this user as unshared. For this function, you may assume that this user has a playlist with `playlistId`.

3.6 MusicStream

In `MusicStream` class, all member functions should utilize `profiles`, `artists`, `albums` and `songs` member variables to operate as described in the following subsections. In *MusicStream.cpp* file, you need to provide implementations for following functions declared under *MusicStream.h* header to complete the assignment.

3.6.1 `void addProfile(const std::string &email, const std::string &username, SubscriptionPlan plan);`

This function adds a new profile to the music streaming service (i.e., registers a new user). It takes the profile information (`email`, `username` and `plan`) as parameter and inserts (appends) a new `Profile` object to the `profiles` linked list. For this function, you may assume that no profile with the given `email` is already registered.

3.6.2 `void deleteProfile(const std::string &email);`

This function deletes a profile from the music streaming service (i.e., deletes an already registered user). It takes email of the profile to delete (`email`) as parameter. Deletion of a user includes some steps: (1) deleting the user from its followers' list of followings, (2) deleting the user from its followings' list of followers, (3) deleting the content of the user's `Profile` object, and finally deleting the user's `Profile` object from the `MusicStream`. For this function, you may assume that a profile with the given `email` is already registered.

3.6.3 `void addArtist(const std::string &artistName);`

This function adds a new artist to the music streaming service. It takes the artist information (`artistName`) as parameter and inserts (appends) a new `Artist` object to the `artists` linked list.

3.6.4 `void addAlbum(const std::string &albumName, int artistId);`

This function adds a new album to the music streaming service. It takes the album information (`artistName`) and the id of the artist that this album belongs to (`artistId`) as parameter and inserts (appends) a new `Album` object to the `albums` linked list. Also remember that the `Artist` object of an artist stores the list of pointers to the `Album` objects of the artist's albums. For this function, you may assume that an artist with the given `artistId` already exists in the service.

3.6.5 `void addSong(const std::string &songName, int songDuration, int albumId);`

This function adds a new song to the music streaming service. It takes the song information (`songName` and `songDuration`) and the id of the album that this song belongs to (`songId`) as parameter and inserts (appends) a new `Song` object to the `songs` linked list. Also remember that the `Album` object of an album stores the list of pointers to the `Song` objects of the album's songs. For this function, you may assume that an album with the given `albumId` already exists in the service.

3.6.6 `void followProfile(const std::string &email1, const std::string &email2);`

This function takes the emails of two users (`email1` and `email2`) as parameter and makes the user with `email1` (i.e., first user) follow the user with `email2` (i.e., second user) by populating their corresponding `Profile` objects' `following` and `followers` lists. For this function, you may assume that the given emails are different and profiles with the given emails are already registered. You may also assume that the first user is not already following the second user.

3.6.7 `void unfollowProfile(const std::string &email1, const std::string &email2);`

This function takes the emails of two users (`email1` and `email2`) as parameter and makes the user with `email1` (i.e., first user) unfollow the user with `email2` (i.e., second user) by populating their corresponding `Profile` objects' `following` and `followers` lists. For this function, you may assume that the given emails are different and profiles with the given emails are already registered. You may also assume that the first user is already following the second user.

3.6.8 `void createPlaylist(const std::string &email, const std::string &playlistName);`

This function creates a new playlist with `playlistName` for the user with `email`. For this function, you may assume that a profile with the given `email` is already registered.

3.6.9 `void deletePlaylist(const std::string &email, int playlistId);`

This function deletes the playlist with `playlistId` of the user with `email`. For this function, you may assume that a profile with the given `email` is already registered and it has a playlist with `playlistId`.

3.6.10 `void addSongToPlaylist(const std::string &email, int songId, int playlistId);`

This function adds (not creates) the song with `songId` to the playlist with `playlistId` of the user with `email`. For this function, you may assume that a profile with the given `email` is already registered, it has a playlist with `playlistId` and the song with `songId` is not already in the playlist.

3.6.11 `void deleteSongFromPlaylist(const std::string &email, int songId, int playlistId);`

This function removes the song with `songId` from the playlist with `playlistId` of the user with `email`. For this function, you may assume that a profile with the given `email` is already registered, it has a playlist with `playlistId` and the song with `songId` is already in the playlist.

3.6.12 `LinkedList<Song *> playPlaylist(const std::string &email, Playlist *playlist);`

This function returns a linked list of pointers to the songs in the given `playlist`. If the user with `email` is subscribed to the `premium` plan, you have to return the linked list of songs as it is. However, if the user is not subscribed to the premium plan (i.e., has the `free_of_charge` plan), you have to return the linked list such that the default advertisement song (`Song::ADVERTISEMENT_SONG`) exists after every song. For this function, you may assume that a profile with the given `email` is already registered and the given `playlist` belongs to that profile or shared with that profile.

3.6.13 `Playlist *getPlaylist(const std::string &email, int playlistId);`

This function gets (i.e., returns a pointer to) the playlist with `playlistId` of the user with `email`. For this function, you may assume that a profile with the given `email` is already registered and it has a playlist with `playlistId`.

3.6.14 `LinkedList<Playlist *> getSharedPlaylists(const std::string &email);`

This function gets a linked list of (i.e., returns a linked list of pointers to) the playlists shared by the users who are followed by the user with `email`. For this function, you may assume that a profile with the given `email` is already registered.

3.6.15 `void shufflePlaylist(const std::string &email, int playlistId, int seed);`

This function shuffles the songs in the playlist with `playlistId` of the user with `email`. For shuffling, use `shuffle(int seed)` function of the `LinkedList` class with the given `seed`. For this function, you may assume that a profile with the given `email` is already registered and it has a playlist with `playlistId`.

3.6.16 `void sharePlaylist(const std::string &email, int playlistId);`

This function marks the playlist with `playlistId` of the user with `email` as shared. For this function, you may assume that a profile with the given `email` is already registered and it has a playlist with `playlistId`.

3.6.17 `void unsharePlaylist(const std::string &email, int playlistId);`

This function marks the playlist with `playlistId` of the user with `email` as unshared. For this function, you may assume that a profile with the given `email` is already registered and it has a playlist with `playlistId`.

3.6.18 `void subscribePremium(const std::string &email);`

This function changes the plan of the user with `email` to `premium`. For this function, you may assume that a profile with the given `email` is already registered.

3.6.19 `void unsubscribePremium(const std::string &email);`

This function changes the plan of the user with `email` to `free_of_charge`. For this function, you may assume that a profile with the given `email` is already registered.

4 Driver Programs

To enable you to test your `LinkedList` and `MusicStream` implementations, two driver programs, *main_linkedlist.cpp* and *main_musicstream.cpp* are provided.

5 Regulations

1. **Programming Language:** You will use C++.
2. **S**tandard Template Library is **not** allowed.
3. **E**xternal libraries other than those already included are **not** allowed.
4. **T**hose who do the operations (insert, remove, get) without utilizing the linked list will receive **0 grade**.
5. **T**hose who modify already implemented functions and those who insert other data variables or public functions and those who change the prototype of given functions will receive **0 grade**.
6. **T**hose who use STL vector or compile-time arrays or variable-size arrays (not existing in ANSI C++) will receive **0 grade**. Options used for g++ are "-ansi -Wall -pedantic-errors -O0". They are already included in the provided Makefile.
7. **Y**ou can add private member functions whenever it is explicitly allowed.
8. **Late Submission Policy:** Each student receives 5 late days for the entire semester. You may use late days on programming assignments, and each allows you to submit up to 24 hours late without penalty. For example, if an assignment is due on Thursday at 11:30pm, you could use 2 late days to submit on Saturday by 11:30pm with no penalty. Once a student has used up all their late days, each successive day that an assignment is late will result in a loss of 5% on that assignment.

   No assignment may be submitted more than 3 days (72 hours) late without permission from the course instructor. In other words, this means there is a practical upper limit of 3 late days usable per assignment. If unusual circumstances truly beyond your control prevent you from submitting an assignment, you should discuss this with the course staff as soon as possible. If you contact us well in advance of the deadline, we may be able to show more flexibility in some cases.

9. **Cheating:** We have zero tolerance policy for cheating. In case of cheating, all parts involved (source(s) and receiver(s)) get zero. People involved in cheating will be punished according to the university regulations. Remember that students of this course are bounded to code of honor and its violation is subject to severe punishment.

10. **Newsgroup:** You must follow the Forum (`odtuclass.metu.edu.tr`) for discussions and possible updates on a daily basis.

6 Submission

- Submission will be done via CengClass (`cengclass.ceng.metu.edu.tr`).
- Don't write a main function in any of your source files.
- A test environment will be ready in CengClass.
  - You can submit your source files to CengClass and test your work with a subset of evaluation inputs and outputs.
  - Additional test cases will be used for evaluation of your final grade. So, your actual grades may be different than the ones you get in CengClass.
  - Only the last submission before the deadline will be graded.