Ceng213 - Data Structures
Programming Assignment 1 : Linked Lists and a Simple Social Network

Fall 2020

1 Objectives

In this programming assignment, you are first expected to implement a *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 and a tail pointer that points to the first and the last nodes 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 *social network* application.

Keywords: *C++, Data Structures, Linked List, Doubly Linked List, Social Network*

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 the 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 the data stored in nodes. `Node` class is the basic building block of the `LinkedList` class. `LinkedList` class has two `Node` pointers in its private data field (namely `head` and `tail`) which point to the first and the last nodes of the linked 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` keeps two pointers (namely `prev` and `next`) to its previous and next nodes in the list, and the data variable of type T (namely `data`) to hold the data. The class has two constructors, and the overloaded output operator. They are already implemented for you. You should not change anything in file *Node.h*.

2.2 `LinkedList`

`LinkedList` class implements a doubly linked list data structure with the `head` and the `tail` pointers. 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 interface methods 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 T arr[], int arrSize);`

This constructor takes an array of `T` objects (`arr`) and the size of the `arr` array (`arrSize`) as parameters. You should make necessary initializations, create new nodes by copying the `T` objects in given `arr` array and insert new nodes into the linked list.

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

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

2.2.4 `~LinkedList();`

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

2.2.5 `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 `nullptr`.

2.2.6 `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 `nullptr`.

2.2.7 `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 no such node in the linked list, you should return `nullptr`.

2.2.8 `int getNumberOfNodes() const;`

This function should return an integer that is the number of nodes in the linked list.

2.2.9 `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 it is not empty, it should return `false`.

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

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

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

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

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

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

2.2.13 `void insertSorted(const T &data);`

You should create a new node with given `data` and insert it to the appropriate place of the linked list. For this function, you may assume that the linked list will be already sorted in ascending order with respect to the data values of its nodes and you should keep the linked list sorted after making the insertion. You may also assume that there will be no nodes with duplicate data in the linked list. You can use overloaded relational operators (i.e. `operator<`, `operator>`, `operator<=`, `operator>=`) to compare two `T` objects. Don't forget to make necessary pointer, and head-tail modifications.

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

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

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

You should delete the node that has the same data with the given `data` from the linked list. Don't forget to make necessary pointer, and head-tail modifications. If there exists no such node in the linked list, do nothing.

2.2.16 `void removeAllNodes();`

You should remove all nodes in the linked list so that the linked list becomes empty.

2.2.17 `T *toArray() const;`

You should return an array containing the data of all nodes of the linked list from first node's data to last node's data. If the linked list is empty, it should return `nullptr`.

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

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

3 Social Network Implementation (50 pts)

The social network in this assignment is implemented as the class `SocialNetwork`. `SocialNetwork` class has two `LinkedList` objects in its private data field (namely `profiles` and `posts`) with the types `Profile` and `Post`, respectively. These two `LinkedList` objects keep the profiles and posts of the social network. `Profile` class represents the users of the social network and `Post` class represents the messages shared by the users of the social network.

The `SocialNetwork`, `Profile` and `Post` classes has their definitions in *SocialNetwork.h*, *Profile.h* and *Post.h* files and their implementations in *SocialNetwork.cpp*, *Profile.cpp* and *Post.cpp* files, respectively.

3.1 `Profile`

`Profile` objects keep `firstname`, `lastname` and `email` variables of type `std::string` to hold the data related with the users of the social network. They also keep linked lists of pointers to the profiles of the user's friend (namely `friends`), pointers to the posts liked by the user (namely `likes`), and pointers to the posts by the user (namely `posts`). `Profile` and `Post` pointers in this three linked lists are pointers to the data variables of the nodes in `SocialNetwork` 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*.

3.1.1 `bool operator==(const Profile &rhs) const;`

This is the overloaded equality operator. You should compare `Profile` objects by `firstname`, `lastname`, and `email` variables. If all three variables are equal, this function should return `true`. Otherwise, it should return `false`.

3.1.2 `bool operator<(const Profile &rhs) const;`

This is the overloaded less than comparison operator. You should compare `Profile` objects by `firstname` and `lastname` variables. If `lastname` is lexicographically less than the `lastname` of `rhs`, return `true`. If they are same and `firstname` is lexcographically less than the `firstname` of `rhs`, return `true`. Otherwise, return `false`.

3.2 `Post`

`Post` objects keep `message` variable of type `std::string` and `postId` variable of type `int` to hold the data related with posts of the users of the social network. Most of the functions of `Post` class are already implemented for you. In *Post.cpp* file, you need to provide implementations for following functions declared under *Post.h* header to complete the assignment. You should not change anything in file *Post.h*.

3.2.1 `bool operator==(const Post &rhs) const;`

This is the overloaded equality operator. You should compare `Post` objects by `message` and `postId` variables. If both variables are equal, this function should return `true`. Otherwise, it should return `false`.

3.2.2 `bool operator<(const Post &rhs) const;`

This is the overloaded less than comparison operator. You should compare `Post` objects by `postId` variable. If `postId` is less than the `postId` of `rhs`, return `true`. Otherwise, return `false`.

3.3 `SocialNetwork`

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

3.3.1 `void addProfile(const std::string &firstname, const std::string &lastname, const std::string &email);`

This function adds a new profile (i.e. registers a new user). It takes profile information (`firstname`, `lastname` and `email`) as parameter and inserts a new `Profile` object to the `profiles` linked list. You should use `insertSorted()` function for insertion. For this function, you may assume that the given `email` is not already registered.

3.3.2 `void addPost(const std::string &message, const std::string &email);`

This function adds a new post. It takes post information (`message`) and the owner profile's email (`email`) as parameters and inserts a new `Post` object to the `posts` linked list. It also marks/adds the new post as a post by user with given email by populating the corresponding profile object's `posts` list. You should use `insertAtTheTail()` function for both insertions. For this function, you may assume that the given email is already registered.

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

This function deletes an already registered user (i.e. profile). It takes email of a profile (`email`) as parameter. Deletion of a user includes some steps, which are deleting the user from its friends' list of friends, deleting the user's posts from other users' list of likes, deleting content of the user's `Profile` object, finally deleting the user's `Profile` object from the `SocialNetwork`. For this function, you may assume that the given email is already registered.

3.3.4 `void makeFriends(const std::string &email1, const std::string &email2);`

This function marks/adds two profiles (i.e. users) as friends with each other. It takes emails of two users (`email1` and `email2`) as parameters and makes them friends by populating their `Profile` objects' `friends` lists. For this function, you may assume that the given emails are different and they are already registered. You may also assume that they are not friends yet.

3.3.5 `void likePost(int postId, const std::string &email);`

This function marks a post as liked by a user. It takes id of a post (`postId`) and email of a profile (`email`) as parameters and marks that post as liked by that user by populating the corresponding `Profile` object's `likes` list. For this function, you may assume that the given email is already registered, and there exists a post with given id. You may also assume that the post is not liked by the user yet.

3.3.6 `void unlikePost(int postId, const std::string &email);`

This function makes a user unlike a post. It takes id of a post (`postId`) and email of a profile (`email`) as parameters and removes corresponding `Post` pointer from the corresponding `Profile` object's `likes` list. For this function, you may assume that the given email is already registered, and there exists a post with given id. If the post is not already liked by the user, do nothing.

3.3.7 `LinkedList<Profile *> getMutualFriends(const std::string &email1, const std::string &email2);`

This function returns a linked list of pointers to the corresponding `Profile` objects of the mutual friends of two users with given emails (`email1` and `email2`). List of mutual friends should have the same order as they exist in the first user's list of friends. For this function, you may assume that the given emails are different and they are already registered.

3.3.8 `LinkedList<Post *> getListOfMostRecentPosts(const std::string &email, int k=0);`

This function returns a linked list of pointers to the corresponding `Post` objects of the `k` most recent posts of the user with given email (`email`). List of posts should be sorted from latest post to earliest post. For this function, you may assume that the given email is already registered.

4 Driver Programs

To enable you to test your `LinkedList` and `SocialNetwork` implementations, two driver programs, *main_linkedlist.cpp* and *main_socialnetwork.cpp* are provided. Their expected outputs are also provided in *output_linkedlist.txt* and *output_socialnetwork.txt* files, respectively.

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 -std=c++11". 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.