```
#ifndef _DEFS_H_
#define _DEFS_H_

// Admin login (name_ is the social network name)
#define ADMIN_LOGIN_SUCCESS "Hi, Administrator. Welcome to " << name_
#define ADMIN_LOGIN_FAIL "Failed to login as admin"

// Login
#define LOGIN_SUCCESS "Hi, " << activeFollower->GetName() << ". Welcome to " << name_ << endl
<< "Notifications: " << activeFollower->NumUnreadMessages() << " unread messages, " <<
activeFollower->NumFriendRequests() << " friend requests."
#define LOGIN_FAIL "User name or password incorrect"

// Logout
#define LOGOUT_SUCCESS "Goodbye"
#define LOGOUT_FAIL "Not logged in"

// Create Leader
#define CREATE_LEADER_SUCCESS "Leader successfully created"
#define CREATE_LEADER_FAIL "Not logged in as admin or user already exists"

// DeleteUser
#define DELETE_USER_SUCCESS "User deleted"
#define DELETE_USER_FAIL "Not logged in as admin or user does not exists"

// BroadcastMessage
#define BROADCAST_MESSAGE_SUCCESS "Messages sent to all followers"
#define BROADCAST_MESSAGE_FAIL "Not logged in or not leader"

// CreateFollower
#define CREATE_FOLLOWER_SUCCESS "Follower successfully created"
#define CREATE_FOLLOWER_FAIL "User already exists"

// ShowFriendRequests (On success put this line in a loop. the variable "i" starts from 0)
#define SHOW_FRIEND_REQUESTS_SUCCESS i + 1 << ") " << curRequest->name_ << ": " <<
curRequest->email_
#define SHOW_FRIEND_REQUESTS_FAIL "Not logged in"

// ShowFriendList (On success put this line in a loop. the variable "i" starts from 0)
#define SHOW_FRIEND_LIST_SUCCESS i + 1 << ") " << curFriend->name_ << ": " << curFriend->email_
#define SHOW_FRIEND_LIST_FAIL "Not logged in"

// SendFriendRequest
#define SEND_FRIEND_REQUEST_SUCCESS "Request Sent"
#define SEND_FRIEND_REQUEST_FAIL "Not logged in ,or user does not exist, or cannot befriend
self, or already friends"

// AcceptFriendRequest
#define ACCEPT_FRIEND_REQUEST_SUCCESS "Request accepted"
#define ACCEPT_FRIEND_REQUEST_FAIL "Not logged in or no such request"

// RemoveFriend
#define REMOVE_FRIEND_SUCCESS "Friend removed"
#define REMOVE_FRIEND_FAIL "Not logged in or no such friend"

// ShowMessageList (On success put this line in a loop. the variable "numMessage" starts from 1)
#define SHOW_MESSAGE_LIST_SUCCESS curMessage->Display(numMessage); // Use without cout
#define SHOW_MESSAGE_LIST_FAIL "Not logged in"

// ReadMessage
#define READ_MESSAGE_SUCCESS curMessage->Read(); // Use without cout
#define READ_MESSAGE_FAIL "Not logged in or invalid message number"

// SendMessage
```

```
#define SEND_MESSAGE_SUCCESS "Message sent"
#define SEND_MESSAGE_FAIL "Not logged in or no such friend"

// Follow
#define FOLLOW_SUCCESS "Added as follower"
#define FOLLOW_FAIL "Not logged in, or no such leader, or already following"


enum Result {FAILURE, SUCCESS};
enum Type { NONE, ADMIN, FOLLOWER, LEADER, MESSAGE };

#endif
```

```
#ifndef _SOCIALNETWORK_H
#define _SOCIALNETWORK_H

#include "defs.H"
#include "Follower.H"
#include "Leader.H"
#include "Lists.H"
#include "Message.H"
#include <string>
using namespace std;


class SocialNetwork {
public:
    SocialNetwork(string name, string password);
    ~SocialNetwork();

    void AdminLogin(string password);
    void Login(string email, string password);
    void Logout();

    // Admin actions
    void CreateLeader(string name, string email, string password);
    void DeleteUser(string email);

    // Leader actions
    void BroadcastMessage(string subject, string content);

    // Follower actions (also leader actions)
    void CreateFollower(string name, string email, string password);

    void ShowFriendRequests();
    void ShowFriendList();
    void SendFriendRequest(string friendEmail);
    void AcceptFriendRequest(string friendEmail);
    void RemoveFriend(string friendEmail);

    void ShowMessageList();
    void ReadMessage(int messageNum);
    void SendMessage(string email, string subject, string content);

    void Follow(string leaderEmail);

    // General actions
    void FindUser(string partialName);
private:
    string name_;
    string password_;
    List leaders_;
    List followers_;
    Type connectedtype_;
    void* currentuser_;
    // Private functions
    void RemoveFromLists(string email, Type type);
};

#endif
```

```
#ifndef _LISTS_H_
#define _LISTS_H_

#include "defs.H"

class Follower;
class Leader;
class Message;

struct Node {
    Follower* pFollower;
    Leader* pLeader;
    Message* pMessage;
    Node* next;
};

class List {
public:
    List(Type type);
    ~List();
    void goHead();
    Result add(void* data);
    Result deleteNode();
    Result getNext();
    void* getValue() const;
    int getSize() const;

private:
    Type type_;
    Node* head_;
    Node* iterator_;
    int size_;
};

#endif
```

```
#ifndef _MESSAGE_H_
#define _MESSAGE_H_

#include "defs.H"
#include "Lists.H"
#include <string>

using namespace std;

class Message {
public:
    Message(string source, string subject, string content);
    void Display(int num) const;
    void Read();
    bool isRead();
private:
    string source_;
    string subject_;
    string content_;
    bool   read_;
};

class MessageBox{
public:
    MessageBox();
    ~MessageBox();
    void Add(Message newMessage);
    int Size();
    int UnreadSize();
    void Print();
    Result ReadMessage(int messageNum);
private:
    List MessageBox_;
    int Total_Message_, Unread_Message_;
};
#endif
```

```
#ifndef _FOLLOWER_H_
#define _FOLLOWER_H_

#include "defs.H"
#include "Message.H"
#include "Lists.H"

using namespace std;
bool checkExists(List& list, string email);

class Follower {
public:
    Follower(string name, string email, string password);
    ~Follower();
    string GetName() const;
    string GetEmail() const;
    bool isPassword(string password) const;
    void showFriendRequests();
    void showFriendsList();
    Result addFriendRequest(Follower* pFollower);
    Result AcceptFriendRequest(string email);
    void RemoveFriendRequest(string email);
    Result RemoveFriend(string email);
    int NumFriendRequests() const;
    void showMessages();
    void addMessage(Message newMessage);
    Result ReadMessage(int messageNum);
    Result SendMessage(string email, string subject, string content);
    int NumUnreadMessages();
    Result AddFriend(Follower* pFollower);
protected:
    string name_;
    string email_;
    string password_;
    List friends_;
    List requests_;
    MessageBox Inbox_;
};

#endif
```

```
#ifndef _LEADER_H_
#define _LEADER_H_

#include "defs.H"
#include "Follower.H"
#include "Lists.H"

class Leader : public Follower {
public:
    Leader(string name, string email, string password);
    ~Leader();
    Result AddFollower(Follower* pFollower);
    Result RemoveFollower(string email);
    int GetNumOfFollower() const;
    void BroadcastMessageToAll(string subject, string content);
protected:
    List followlist_;
};

#endif
```

```cpp
#include<iostream>
#include "Lists.H"

//**************************************************************************************************
**********
//* function name:  List
//* Description  :  Constructor of list class
//* Parameters   :  type - type of list(Follower, Leader or Message)
//* Return Value :  None
//**************************************************************************************************
**********
List::List(Type type) : type_(type), head_(NULL), iterator_(NULL),size_(0) {};

//**************************************************************************************************
**********
//* function name:  ~List
//* Description  :  Destructor of list class
//* Parameters   :  None
//* Return Value :  None
//**************************************************************************************************
**********
List::~List() {
    while (head_ != NULL) {
        iterator_ = head_->next;
        delete head_;
        head_ = iterator_;
    }
}

//**************************************************************************************************
**********
//* function name:  goHead
//* Description  :  setting the iterator on the head of the list
//* Parameters   :  None
//* Return Value :  None
//**************************************************************************************************
**********
void List::goHead() {
    iterator_ = head_;
}

//**************************************************************************************************
**********
//* function name:  add
//* Description  :  adding a new data node to the head of the list
//* Parameters   :  data - a pointer to the added data
//* Return Value :  SUCCESS or FAILURE
//**************************************************************************************************
**********
Result List::add(void* data) {
    if (data == NULL) return FAILURE; // no data
    Node* tmp = head_;
    head_ = new Node;
    head_->pFollower = NULL;
    head_->pLeader = NULL;
    head_->pMessage = NULL;
    switch (type_)
    {
    case FOLLOWER:  head_->pFollower = (Follower*)data;
                    break;
    case LEADER:    head_->pLeader = (Leader*)data;
                    break;
    case MESSAGE:   head_->pMessage = (Message*)data;
                    break;
```

```
    }
    head_->next = tmp;
    size_++;
    iterator_ = head_;
    return SUCCESS;
}

//**************************************************************************************************
**********
//* function name:  deleteNode
//* Description  :  Delete the node the iterator points at. For illegal iterator return FAILURE
//* Parameters   :  None
//* Return Value :  SUCCESS or FAILURE
//**************************************************************************************************
**********
Result List::deleteNode() {
    if (iterator_ == NULL) return FAILURE; // illegal iterator_
    if (iterator_ == head_) { // case head_ of the list
        iterator_ = head_->next;
        delete head_;
        head_ = iterator_;
        size_--;
        return SUCCESS;
    }
    Node* tmp = head_;
    while (tmp->next != iterator_)
        tmp = tmp->next; // go to node before the requested one
    tmp->next = iterator_->next;
    delete iterator_;
    iterator_ = head_;
    size_--;
    return SUCCESS;
}

//**************************************************************************************************
**********
//* function name:  getNext
//* Description  :  Moves the iterator to the next item on the list. Return FAILURE if the
iterator is illegal
//* Parameters   :  None
//* Return Value :  SUCCESS or FAILURE
//**************************************************************************************************
**********
Result List::getNext() {
    if (iterator_ == NULL) return FAILURE;
    iterator_ = iterator_->next;
    return SUCCESS;
}

//**************************************************************************************************
**********
//* function name:  getValue
//* Description  :  Returns a pointer to the data in the node that the iterator is currently
pointing at
//* Parameters   :  None
//* Return Value :  a pointer to the data
//**************************************************************************************************
**********
void* List::getValue() const {
    if (iterator_ == NULL) return NULL;
    switch (type_){
    case FOLLOWER:  return iterator_->pFollower;
    case LEADER:    return iterator_->pLeader;
    case MESSAGE:   return iterator_->pMessage;
```

```
    }
    return NULL;
}
//*****************************************************************************************************
**********
//* function name:  getSize
//* Description  :  Returns the number of items in the list
//* Parameters   :  None
//* Return Value :  size_ (int)
//*****************************************************************************************************
**********
int List::getSize() const{
    return size_;
}
```

```cpp
#include <iostream>
#include "Message.H"

//**********************************************************************************************
**********
//* function name:  Message
//* Description  :  Constructor of Message class
//* Parameters   :  source-the source of the message
//*              :  subject-the subject of the message
//*              :  content-the content of the message
//* Return Value :  None
//**********************************************************************************************
**********
Message::Message(string source, string subject, string content) : source_(source),
subject_(subject), content_(content), read_(false) {}

//**********************************************************************************************
**********
//* function name:  Display
//* Description  :  prints the message
//* Parameters   :  num- number of messages
//* Return Value :  None
//**********************************************************************************************
**********
void Message::Display(int num) const
{
    cout << num << ") "<< (read_ ? "" : "(Unread) ") << "From: " << source_ << endl;
    cout << "Subject: " << subject_ << endl;
}

//**********************************************************************************************
**********
//* function name:  Read
//* Description  :  Read a Message
//* Parameters   :  None
//* Return Value :  None
//**********************************************************************************************
**********
void Message::Read()
{
    read_ = true;
    cout << "From: " << source_ << endl;
    cout << "Subject: " << subject_ << endl;
    cout << "Content: " << content_ << endl;
}

//**********************************************************************************************
**********
//* function name: isRead
//* Description  : checks if this is a read message
//* Parameters   : None
//* Return Value : Bool
//**********************************************************************************************
**********
bool Message::isRead()
{
    return read_;
}

//**********************************************************************************************
**********
//* function name: MessageBox
//* Description  : Constructor of MessageBox class  , sets values of 0 to all messages and
those who not were read
```

```
//* Parameters   : None
//* Return Value : None
//*****************************************************************************************
**********
MessageBox::MessageBox() : MessageBox_(MESSAGE), Total_Message_(0), Unread_Message_(0) {};

//*****************************************************************************************
**********
//* function name:  ~MessageBox()
//* Description  :  Destructor of MessageBox class
//* Parameters   :  None
//* Return Value :  None
//*****************************************************************************************
**********
MessageBox::~MessageBox() {
    MessageBox_.goHead();
    while (static_cast<Message*>(MessageBox_.getValue()) != NULL) {
        delete (static_cast<Message*>(MessageBox_.getValue()));
        MessageBox_.getNext();
    }
}

//*****************************************************************************************
**********
//* function name: Add
//* Description  : add a new massage the massagebox
//* Parameters   : newMessage - new message
//* Return Value : None
//*****************************************************************************************
**********
void MessageBox::Add(Message newMessage) {
    MessageBox_.add(new Message(newMessage));
    Total_Message_++;
    Unread_Message_++;
}

//*****************************************************************************************
**********
//* function name:  Size
//* Description  :  Return the number of total messages
//* Parameters   :  None
//* Return Value :  Total_Message_-the number of them
//*****************************************************************************************
**********
int MessageBox::Size(){
    return Total_Message_;
}

//*****************************************************************************************
**********
//* function name:  UnreadSize
//* Description  :  Return the number of total unread messages
//* Parameters   :  None
//* Return Value :  Unread_Message_-the number of them
//*****************************************************************************************
**********
int MessageBox::UnreadSize(){
    return Unread_Message_;
}

//*****************************************************************************************
**********
//* function name: Print
//* Description  : Display a summary of all messages
```

```
//* Parameters   : None
//* Return Value : None
//*****************************************************************************************
**********
void MessageBox::Print() {
    MessageBox_.goHead();
    int n = MessageBox_.getSize();
    for (int i = 0; i<n; i++){
        Message* tmp = static_cast<Message*>(MessageBox_.getValue());
        tmp->Display(i+1);
        MessageBox_.getNext();
    }
}

//*****************************************************************************************
**********
//* function name: ReadMessage
//* Description  : use it to read a message
//* Parameters   : read a message and also update the number of the unread mesaages
//* Return Value : Result- if we succeded return SUCCESS else FAILURE
//*****************************************************************************************
**********
Result MessageBox::ReadMessage(int messageNum) {
    int n = MessageBox_.getSize();
    if (messageNum <= 0 || messageNum > n) return FAILURE;
    MessageBox_.goHead();
    for (int i=0;i<messageNum-1;i++)
        MessageBox_.getNext(); // go to the desired message
    Message* tmp = static_cast<Message*>(MessageBox_.getValue());
    if (tmp->isRead() == false) Unread_Message_--;
    tmp->Read();
    return SUCCESS;
}
```

```cpp
#include <iostream>
#include "Follower.H"
#include <string>

using namespace std;

//**********************************************************************************************
**********
//* function name:  Follower
//* Description  :  Constructor of Follower class
//* Parameters   :  name - name of the follower
//*                 email - email of the follower(must be UNIQUE)
//*                 password - password of the user
//* Return Value :  None
//**********************************************************************************************
**********
Follower::Follower(string name, string email, string password) :
    name_(name), email_(email), password_(password), friends_(FOLLOWER), requests_(FOLLOWER) {};

//**********************************************************************************************
**********
//* function name:  ~Follower
//* Description  :  Destructor of Follower class
//* Parameters   :  None
//* Return Value :  None
//**********************************************************************************************
**********

Follower::~Follower() {};

//**********************************************************************************************
**********
//* function name:  GetName
//* Description  :  Returning the name of the follower
//* Parameters   :  None
//* Return Value :  Name of the follower
//**********************************************************************************************
**********
string Follower::GetName() const {
    return name_;
}

//**********************************************************************************************
**********
//* function name:  GetEmail
//* Description  :  Returning the email of the follower
//* Parameters   :  None
//* Return Value :  email of the follower
//**********************************************************************************************
**********
string Follower::GetEmail() const {
    return email_;
}

//**********************************************************************************************
**********
//* function name:  isPassword
//* Description  :  Checking if the password entered by the user is the current one
//* Parameters   :  None
//* Return Value :  true of false
//**********************************************************************************************
**********
bool Follower::isPassword(string password) const {
    if (password_.compare(password) != 0) return false;
```

```
    return true;
}

//**************************************************************************************************
**********
//* function name:  showFriendRequests
//* Description  :  Going through the friend requests list and displaying it
//* Parameters   :  None
//* Return Value :  None
//**************************************************************************************************
**********
void Follower::showFriendRequests() {
    requests_.goHead();
    int n = requests_.getSize();
    for (int i = 0; i < n; i++) {
        Follower* curRequest = static_cast<Follower*>(requests_.getValue());
        cout << SHOW_FRIEND_REQUESTS_SUCCESS << endl;
        requests_.getNext();
    }
}

//**************************************************************************************************
**********
//* function name:  showFriendsList
//* Description  :  Going through the friends list and displaying it
//* Parameters   :  None
//* Return Value :  None
//**************************************************************************************************
**********
void Follower::showFriendsList() {
    friends_.goHead();
    int n=friends_.getSize();
    for (int i = 0; i < n; i++) {
        Follower* curFriend = static_cast<Follower*>(friends_.getValue());
        cout << SHOW_FRIEND_LIST_SUCCESS << endl;
        friends_.getNext();
    }
}

//**************************************************************************************************
**********
//* function name:  addFriendRequest
//* Description  :  Adding a friend request from a fellow follower. Return FAILURE if already
friends or if
//*                 already sent request
//* Parameters   :  pFollower - a pointer to the friendship requesting follower
//* Return Value :  SUCCESS or FAILURE
//**************************************************************************************************
**********
Result Follower::addFriendRequest(Follower* pFollower) {
    if ((checkExists(friends_, pFollower->GetEmail()) == true) || // already friends
        (checkExists(requests_, pFollower->GetEmail()) == true) // already sent request
        )
        return FAILURE;
    requests_.add(pFollower);
    return SUCCESS;
}

//**************************************************************************************************
**********
//* function name:  AcceptFriendRequest
//* Description  :  Accepting a friend request. Return FAILURE if already friends or no friend
request
//* Parameters   :  email - the email of the user accepting the friend request
```

```
//* Return Value :  SUCCESS or FAILURE
//**********************************************************************************************
**********
Result Follower::AcceptFriendRequest(string email) {
    if (checkExists(friends_, email) == true)
        return FAILURE; // already friends
    if (checkExists(requests_, email) == true){
        Follower* tmp = static_cast<Follower*>(requests_.getValue());
        requests_.deleteNode(); // iterator will be on the request
        friends_.add(tmp);
        return SUCCESS;
    }
    return FAILURE; // no such request
}

//**********************************************************************************************
**********
//* function name:  RemoveFriendRequest
//* Description  :  Removing a friend request of a given email if exists
//* Parameters   :  email - email of the follower to be removed
//* Return Value :  None
//**********************************************************************************************
**********
void Follower::RemoveFriendRequest(string email) {
    if (checkExists(requests_, email))
        requests_.deleteNode(); // iterator will be on the request
}

//**********************************************************************************************
**********
//* function name:  RemoveFriend
//* Description  :  Removing a friend :'(
//* Parameters   :  email - email of the friend to be removed
//* Return Value :  SUCCESS or FAILURE
//**********************************************************************************************
**********
Result Follower::RemoveFriend(string email) {
    if (checkExists(friends_, email)) {
        return friends_.deleteNode(); // iterator will be on the request
    }
    return FAILURE; // not friends
}

//**********************************************************************************************
**********
//* function name:  NumOfFriendRequests
//* Description  :  Getting the amount of friend requests
//* Parameters   :  None
//* Return Value :  Number of friends request (int)
//**********************************************************************************************
**********
int Follower::NumFriendRequests() const {
    return requests_.getSize();
}

//**********************************************************************************************
**********
//* function name:  showMessages
//* Description  :  Printing the inbox messages
//* Parameters   :  None
//* Return Value :  None
//**********************************************************************************************
**********
void Follower::showMessages() {
```

```
    Inbox_.Print();
}

//*****************************************************************************************************
**********
//* function name:  addMessage
//* Description  :  adding a new message
//* Parameters   :  message - the new message to the inbox
//* Return Value :  None
//*****************************************************************************************************
**********
void Follower::addMessage(Message newMessage) {
    Inbox_.Add(newMessage);
}

//*****************************************************************************************************
**********
//* function name:  ReadMessage
//* Description  :  Reading a message of a given id. Returning FAILURE of no message to be
fetched
//* Parameters   :  serial - message id
//* Return Value :  SUCCESS or FAILURE
//*****************************************************************************************************
**********
Result Follower::ReadMessage(int messageNum) {
    return Inbox_.ReadMessage(messageNum);
}

//*****************************************************************************************************
**********
//* function name:  SendMessage
//* Description  :  Sending a message with <subject> and <content> to a user with <email>
//* Parameters   :  email - the destination user email
//*                 subject - message's subject
//*                 content - message's content
//* Return Value :  SUCCESS or FAILURE
//*****************************************************************************************************
**********
Result Follower::SendMessage(string email, string subject, string content) {
    if (checkExists(friends_, email)) {
        Message newMessage((this->email_), subject, content);
        (static_cast<Follower*>(friends_.getValue()))->addMessage(newMessage);
        return SUCCESS;
    }
    return FAILURE;
}

//*****************************************************************************************************
**********
//* function name:  NumUnreadMessages
//* Description  :  returning the number of unread message the follower has
//* Parameters   :  None
//* Return Value :  Number of unread messages (int)
//*****************************************************************************************************
**********
int Follower::NumUnreadMessages() {
    return Inbox_.UnreadSize();
}

//*****************************************************************************************************
**********
//* function name:  AddFriend
//* Description  :  Adding pfollower to the friendlist directly without a request(helper func)
//* Parameters   :  pFollower - a pointer to the follower to add to the friendlist
```

```
//* Return Value :  SUCCESS or FAILURE
//*****************************************************************************************
**********
Result Follower::AddFriend(Follower* pFollower) {
    if (checkExists(friends_, (pFollower->GetEmail())) == true) return FAILURE; // already
    friends
    if (checkExists(requests_, (pFollower->GetEmail())) == true) requests_.deleteNode();
    friends_.add(pFollower);
    return SUCCESS;
}

//*****************************************************************************************
**********
//* function name:  checkExists
//* Description  :  checking if a follower with the given email is already in the list(helper
func)
//*                 *IMPORTANT NOTE*: If finding the follower in the list, the iterator will be
on him
//* Parameters   :  list - the list to check in
//*                 string - the follower email
//* Return Value :  true or false
//*****************************************************************************************
**********
bool checkExists(List& list, string email) {
    list.goHead();
    Follower* tmp = static_cast<Follower*>(list.getValue());
    while ((tmp != NULL)) {
        if (tmp->GetEmail().compare(email) == 0) return true;
        list.getNext();
        tmp = static_cast<Follower*>(list.getValue());
    }
    return false; // not found in list
}
```

```cpp
#include<iostream>
#include "Leader.H"

using namespace std;

//**************************************************************************************************
**********
//* function name:  Leader
//* Description  :  Constructor of Leader class
//* Parameters   :  name - name of the Leader
//*                 email - email of the Leader(must be UNIQUE)
//*                 password - password of the Leader
//* Return Value :  None
//**************************************************************************************************
**********
Leader::Leader(string name, string email, string password):
    Follower(name, email, password), followlist_(FOLLOWER) {};

//**************************************************************************************************
**********
//* function name:  ~Leader
//* Description  :  Destructor of Leader class
//* Parameters   :  None
//* Return Value :  None
//**************************************************************************************************
**********
Leader::~Leader() {}

//**************************************************************************************************
**********
//* function name:  AddFollower
//* Description  :  Adding a follower to the leader. If already exists return FAILURE
//* Parameters   :  pFollower - a pointer to the follower
//* Return Value :  SUCCESS or FAILURE
//**************************************************************************************************
**********
Result Leader::AddFollower(Follower* pFollower) {
    if (checkExists(followlist_, pFollower->GetEmail())) return FAILURE; // Already following
    return followlist_.add(pFollower);
}

//**************************************************************************************************
**********
//* function name:  RemoveFollower
//* Description  :  Removing a follower from the leader's list. If does not exists return FAILURE
//* Parameters   :  pFollower - a pointer to the follower
//* Return Value :  SUCCESS or FAILURE
//**************************************************************************************************
**********
Result Leader::RemoveFollower(string email) {
    if (checkExists(followlist_, email) == false ) return FAILURE; // not following
    return followlist_.deleteNode(); // iterator should be on the requested follower to be
    removed
}

//**************************************************************************************************
**********
//* function name:  GetNumOfFollower
//* Description  :  returning the number of followers the leader has
//* Parameters   :  None
//* Return Value :  Number of followers (int)
//**************************************************************************************************
**********
int Leader::GetNumOfFollower() const {
```

```
    return followlist_.getSize();
}

//**************************************************************************************************
**********
//* function name:  BroadcastMessageToAll
//* Description  :  Broadcasting all followers a message with <subject> and <content>
//* Parameters   :  subject - message's subject
//*                 content - message's content
//* Return Value :  None
//**************************************************************************************************
**********
void Leader::BroadcastMessageToAll(string subject, string content) {
    Message newMessage(this->email_, subject, content);
    followlist_.goHead();
    int i;
    int n = followlist_.getSize();
    for (i = 0; i < n; i++) {
        (static_cast<Follower*>(followlist_.getValue()))->addMessage(newMessage);
        followlist_.getNext();
    }
}
```

```cpp
#include "SocialNetwork.H"
#include <iostream>


//**************************************************************************************************
**********
//* function name:  SocialNetwork
//* Description  :  Constructor of SocialNetwork class
//* Parameters   :  name - the social network name
//*                 password - the admin password for the network
//* Return Value :  None
//**************************************************************************************************
**********
SocialNetwork::SocialNetwork(string name, string password) :
    name_(name), password_(password), leaders_(LEADER),
    followers_(FOLLOWER),connectedtype_(NONE), currentuser_(NULL) {}

//**************************************************************************************************
**********
//* function name:  ~SocialNetwork
//* Description  :  Destructor of SocialNetwork class
//* Parameters   :  None
//* Return Value :  None
//**************************************************************************************************
**********
SocialNetwork::~SocialNetwork() {
    followers_.goHead();
    int n = followers_.getSize();

    for (int i = 0; i < n; i++) {
        delete static_cast<Follower*>(followers_.getValue());
        followers_.getNext();
    }
    leaders_.goHead();
    n = leaders_.getSize();
    for (int i = 0; i < n; i++) {
        delete static_cast<Leader*>(leaders_.getValue());
        leaders_.getNext();
    }
}

//**************************************************************************************************
**********
//* function name:  AdminLogin
//* Description  :  Login for admin
//* Parameters   :  password - password input
//* Return Value :  None
//**************************************************************************************************
**********
void SocialNetwork::AdminLogin(string password) {
    if (password.empty())
        cout << ADMIN_LOGIN_FAIL << endl; // no pass entered
    if (password.compare(password_) == 0) {
        connectedtype_ = ADMIN;
        cout << ADMIN_LOGIN_SUCCESS << endl;
    }
    else cout << ADMIN_LOGIN_FAIL << endl; // wrong pass
}

//**************************************************************************************************
**********
//* function name:  Login
//* Description  :  Login to the network ( Leader of Follower)
//* Parameters   :  email - user's email
```

```
//*                  password - password
//* Return Value :  None
//*****************************************************************************************
**********
void SocialNetwork::Login(string email, string password) {
    Type usertype = NONE;
    // login user type
    if (checkExists(followers_, email) == true) usertype = FOLLOWER;
    if (checkExists(leaders_, email) == true) usertype = LEADER;
    if (usertype != NONE) {
        Follower* activeFollower;
        if (usertype == FOLLOWER) activeFollower =
        (static_cast<Follower*>(followers_.getValue()));
        else activeFollower = (static_cast<Leader*>(leaders_.getValue()));
        if (activeFollower->isPassword(password)){
            connectedtype_ = usertype;
            currentuser_ = activeFollower;
            cout << LOGIN_SUCCESS << endl;
            return;
        }
    }
    cout << LOGIN_FAIL << endl;
}

//*****************************************************************************************
**********
//* function name:  Logout
//* Description  :  Logging out the user
//* Parameters   :  None
//* Return Value :  None
//*****************************************************************************************
**********
void SocialNetwork::Logout() {
    if (connectedtype_ == NONE)
        cout << LOGOUT_FAIL << endl; // no one connected!
    else {
        connectedtype_ = NONE;
        currentuser_ = NULL;
        cout << LOGOUT_SUCCESS << endl;
    }
}

////////////////////
/// Admin actions ///
////////////////////

//*****************************************************************************************
**********
//* function name:  CreateLeader
//* Description  :  Create a leader with the given parameters. Must have a unique email and an
admin must
//*                  be logged.
//* Parameters   :  name - the leader name
//*                  email - the leader meail
//*                  password - the leader password
//* Return Value :
//*****************************************************************************************
**********
void SocialNetwork::CreateLeader(string name, string email, string password) {
    if ((connectedtype_ != ADMIN) || // admin not logged
        (checkExists(followers_, email) == true) || // follower with the same mail exists
        (checkExists(leaders_, email) == true) // already a leader!
        ) {
        cout << CREATE_LEADER_FAIL << endl;
```

```
        return;
    }
    Leader* newLeader = new Leader(name, email, password);
    leaders_.add(newLeader);
    cout << CREATE_LEADER_SUCCESS << endl;
}

//**************************************************************************************************
**********
//* function name:  RemoveFromLists
//* Description  :  Given an email, deletes the user from his friends' lists and all friend
requests.
//*                 If it's a leader then also remove his followers from following him. (help
func)
//* Parameters   :  None
//* Return Value :  None
//**************************************************************************************************
**********
void SocialNetwork::RemoveFromLists(string email, Type type) {
    List *target = (type == LEADER) ? &leaders_ : &followers_;
    target->goHead();
    int n = target->getSize();
    for (int i = 0; i < n; i++) {
        Leader* tmp = static_cast<Leader*>(target->getValue());
        if (tmp->GetEmail().compare(email) != 0) {
            tmp->RemoveFriend(email);
            tmp->RemoveFriendRequest(email);
            if (type == LEADER) tmp->RemoveFollower(email);
        }
        target->getNext();
    }
}

//**************************************************************************************************
**********
//* function name:  DeleteUser
//* Description  :  Given an email, deletes the user(follower or leader). Afterwards broadcast
the news to
//*                 all of the friends and leaders of the fallen comarade. Fail if email does
not exist
//*                 or admin is not logged in.
//* Parameters   :  None
//* Return Value :  None
//**************************************************************************************************
**********
void SocialNetwork::DeleteUser(string email) {
    if ((connectedtype_ != ADMIN) || // admin not logged
        ((checkExists(followers_, email) == false) && (checkExists(leaders_, email) == false))
        // user doesn't exists
        ) {
        cout << DELETE_USER_FAIL << endl;
        return;
    }
    RemoveFromLists(email, FOLLOWER);
    RemoveFromLists(email, LEADER);
    if (checkExists(followers_, email)) { // case Follower
        delete (static_cast<Follower*>(followers_.getValue()));
        followers_.deleteNode(); // iterator should be on the requested node
    }
    if (checkExists(leaders_, email)) { // case Leader
        delete (static_cast<Leader*>(leaders_.getValue()));
        leaders_.deleteNode(); // iterator should be on the requested node
    }
    cout << DELETE_USER_SUCCESS << endl;
```

```
}

//////////////////////
/// Leader actions ///
//////////////////////

//*****************************************************************************************
**********
//* function name:  BroadcastMessage
//* Description  :  Send a message with <subject> and <content> to all followers of the logged
Leader
//*                 Returns a failure message if not logged in as a Leader
//* Parameters   :  subject - message subject
//*                 content - message content
//* Return Value :  None
//*****************************************************************************************
**********
void SocialNetwork::BroadcastMessage(string subject, string content) {
    if (connectedtype_ != LEADER) { // not connected as a leader
        cout << BROADCAST_MESSAGE_FAIL << endl;
        return;
    }
    (static_cast<Leader*>(currentuser_))->BroadcastMessageToAll(subject, content);
    cout << BROADCAST_MESSAGE_SUCCESS << endl;
}

////////////////////////////////////////////
/// Follower actions (also leader actions) ///
////////////////////////////////////////////

//*****************************************************************************************
**********
//* function name:  CreateFollower
//* Description  :  Adds a new follower user to the network. Fails if a user with the same
email exists
//* Parameters   :  name - the new follower name
//*                 email - the new follower email(Must be unique
//*                 password - the new follower password
//* Return Value :  None
//*****************************************************************************************
**********
void SocialNetwork::CreateFollower(string name, string email, string password) {
    if ((checkExists(followers_, email) == true) ||
        (checkExists(leaders_, email) == true)){ // user already exists
        cout << CREATE_FOLLOWER_FAIL << endl;
        return;
    }
    Follower* newFollower = new Follower(name, email, password);
    followers_.add(newFollower);
    cout << CREATE_FOLLOWER_SUCCESS << endl;
}

//*****************************************************************************************
**********
//* function name:  ShowFriendRequests
//* Description  :  Showing the friend requests of the logged user
//* Parameters   :  None
//* Return Value :  None
//*****************************************************************************************
**********
void SocialNetwork::ShowFriendRequests() {
    if ((connectedtype_ != LEADER) && (connectedtype_ != FOLLOWER)) {
        cout << SHOW_FRIEND_REQUESTS_FAIL << endl;
        return;
```

```
    }
    (static_cast<Follower*>(currentuser_))->showFriendRequests();
}

//**********************************************************************************************
**********
//* function name:  ShowFriendList
//* Description  :  Showing the friend list of the logged user
//* Parameters   :  None
//* Return Value :  None
//**********************************************************************************************
**********
void SocialNetwork::ShowFriendList() {
    if ((connectedtype_ != LEADER) && (connectedtype_ != FOLLOWER)) {
        cout << SHOW_FRIEND_LIST_FAIL << endl;
        return;
    }
    (static_cast<Follower*>(currentuser_))->showFriendsList();
}

//**********************************************************************************************
**********
//* function name:  SendFriendRequest
//* Description  :  Sending a friend request to your (hopefully) new amigo
//* Parameters   :  friendEmail - the friend's email whom you send the request to
//* Return Value :  None
//**********************************************************************************************
**********
void SocialNetwork::SendFriendRequest(string friendEmail) {
    if (((connectedtype_ != LEADER) && (connectedtype_ != FOLLOWER)) || // A user is not
    connected
        ((checkExists(followers_, friendEmail) == false) &&
        (checkExists(leaders_, friendEmail) == false)) || // Your friend is imaginary
        ((static_cast<Follower*>(currentuser_))->GetEmail().compare(friendEmail) == 0) //
        trying to become afriend with yourself
        ) {
        cout << SEND_FRIEND_REQUEST_FAIL << endl;
        return;
    }
    Follower* currentuser = static_cast<Follower*>(currentuser_);
    if (checkExists(followers_, friendEmail)) { // case follower
        Follower* tmp = static_cast<Follower*>(followers_.getValue()); // iterator should be on
        the friend
        if (tmp->addFriendRequest(currentuser)) cout << SEND_FRIEND_REQUEST_SUCCESS << endl;
        else cout << SEND_FRIEND_REQUEST_FAIL << endl;
    }
    else if (checkExists(leaders_, friendEmail)) { // case leader
        Leader* tmp = static_cast<Leader*>(leaders_.getValue()); // iterator should be on the
        friend
        if (tmp->addFriendRequest(currentuser)) cout << SEND_FRIEND_REQUEST_SUCCESS << endl;
        else cout << SEND_FRIEND_REQUEST_FAIL << endl;
    }
    else cout << SEND_FRIEND_REQUEST_FAIL << endl; // friend request already exists
}

//**********************************************************************************************
**********
//* function name:  AcceptFriendRequest
//* Description  :  Accept a friend request from the user with <email>. Removes your own
request if exists
//*                 Fail if no such request.
//* Parameters   :  friendEmail - the friend's email whom you accept his friendship
//* Return Value :  None
//**********************************************************************************************
```

```
**********
void SocialNetwork::AcceptFriendRequest(string friendEmail) {
    if (((connectedtype_ != LEADER) && (connectedtype_ != FOLLOWER))){ // A user is not connected
        cout << ACCEPT_FRIEND_REQUEST_FAIL << endl;
        return;
    }
    Follower* currentuser = static_cast<Follower*>(currentuser_);
    if (currentuser->AcceptFriendRequest(friendEmail)) {
        // now add the current user as a friend as well
        if (checkExists(followers_, friendEmail)) { // case the new friend is a follower
            Follower* tmp = static_cast<Follower*>(followers_.getValue());
            tmp->AddFriend(currentuser);
            cout << ACCEPT_FRIEND_REQUEST_SUCCESS << endl;
        }
        if (checkExists(leaders_, friendEmail)) { // case the new friend is a leader
            Leader* tmp = static_cast<Leader*>(leaders_.getValue());
            tmp->AddFriend(currentuser);
            cout << ACCEPT_FRIEND_REQUEST_SUCCESS << endl;
        }

    }
    else cout << ACCEPT_FRIEND_REQUEST_FAIL << endl; // no friend request
}

//*****************************************************************************************
**********
//* function name:  RemoveFriend
//* Description  :  Removing the user with the given email from the logged user friends list
//* Parameters   :  friendEmail - the removed friend email
//* Return Value :  None
//*****************************************************************************************
**********
void SocialNetwork::RemoveFriend(string friendEmail) {
    if (((connectedtype_ != LEADER) && (connectedtype_ != FOLLOWER))) { // A user is not
    connected
        cout << REMOVE_FRIEND_FAIL << endl;
        return;
    }
    Follower* currentuser = static_cast<Follower*>(currentuser_);
    string email = currentuser->GetEmail();
    if (currentuser->RemoveFriend(friendEmail) == SUCCESS) {
        // friend removed from OWN list, now remove yourself from his friends list
        if (checkExists(followers_, friendEmail)) {// case the ex-friend is a follower
            Follower* tmp = static_cast<Follower*>(followers_.getValue());
            tmp->RemoveFriend(email);
            cout << REMOVE_FRIEND_SUCCESS << endl;
        }
        if (checkExists(leaders_, friendEmail)) {// case the ex-friend is a leader
            Leader* tmp = static_cast<Leader*>(leaders_.getValue());
            tmp->RemoveFriend(email);
            cout << REMOVE_FRIEND_SUCCESS << endl;
        }
    }
    else cout << REMOVE_FRIEND_FAIL << endl;
}

//*****************************************************************************************
**********
//* function name:  ShowMessageList
//* Description  :  If a user is logged then prints his messages. If not then prints FAIL message
//* Parameters   :  None
//* Return Value :  None
//*****************************************************************************************
**********
```

```
void SocialNetwork::ShowMessageList() {
    if (((connectedtype_ != LEADER) && (connectedtype_ != FOLLOWER))) { // A user is not
    connected
        cout << SHOW_MESSAGE_LIST_FAIL << endl;
        return;
    }
    (static_cast<Follower*>(currentuser_))->showMessages();
}

//**************************************************************************************************
**********
//* function name:  ReadMessage
//* Description  :  Reads a messages of a given serial number
//* Parameters   :  messageNum - message serial number
//* Return Value :  None
//**************************************************************************************************
**********
void SocialNetwork::ReadMessage(int messageNum) {
    if (((connectedtype_ != LEADER) && (connectedtype_ != FOLLOWER)) || // A user is not
    connected
        (static_cast<Follower*>(currentuser_))->ReadMessage(messageNum) == FAILURE) {
        cout << READ_MESSAGE_FAIL << endl;
        return;
    }
}

//**************************************************************************************************
**********
//* function name:  SendMessage
//* Description  :  Sending a message with <subject> and <content> to a user with <email>
//* Parameters   :  email - the destination user email
//*                 subject - message's subject
//*                 content - message's content
//* Return Value :  None
//**************************************************************************************************
**********
void SocialNetwork::SendMessage(string email, string subject, string content) {
    if (((connectedtype_ != LEADER) && (connectedtype_ != FOLLOWER))) { // A user is not
    connected
        cout << SEND_MESSAGE_FAIL << endl;
        return;
    }
    if ((static_cast<Follower*>(currentuser_))->SendMessage(email, subject, content) == SUCCESS )
        cout << SEND_MESSAGE_SUCCESS << endl;
    else
        cout << SEND_MESSAGE_FAIL << endl;
}

//**************************************************************************************************
**********
//* function name:  Follow
//* Description  :  Adding the logged user to the followers list of a leader. Fail if a user is
not logged
//*                 (Follower or Leader) or there is no leader with the given <leaderemail> or
the user is
//*                 already following the leader.
//* Parameters   :  leaderEmail - the email of the leader you want to follow
//* Return Value :  None
//**************************************************************************************************
**********
void SocialNetwork::Follow(string leaderEmail){
    if ((connectedtype_ != LEADER) && (connectedtype_ != FOLLOWER)){// A user is not connected
        cout << FOLLOW_FAIL << endl;
        return;
```

```
    }
    if (checkExists(leaders_, leaderEmail) == true) {
        Follower* currentuser = static_cast<Follower*>(currentuser_); // the iterator should be
        on the leader
        if ((static_cast<Leader*>(leaders_.getValue()))->AddFollower(currentuser)) {
            cout << FOLLOW_SUCCESS << endl;
            return;
        }
    }
    cout << FOLLOW_FAIL << endl;// No such leader or already following
}


//////////////////////
/// General actions ///
//////////////////////

//**********************************************************************************************
**********
//* function name:  FindUser
//* Description  :  print all the users that have the wanted partial name
//* Parameters   :  partialName - a string of a partial name
//* Return Value :  None
//**********************************************************************************************
**********
void SocialNetwork::FindUser(string partialName)
{
    cout << "Followers:" << endl;
    // Loop over all followers in network
    followers_.goHead();
    for (int i = 0; i<followers_.getSize(); ++i)
    {
        Follower* curFollower = static_cast<Follower*>(followers_.getValue());
        if (curFollower->GetName().find(partialName) != string::npos)
            cout << i + 1 << ") " << curFollower->GetName() << ": " << curFollower->GetEmail()
            << endl;
        followers_.getNext();
    }

    cout << "Leaders:" << endl;
    // Loop over all leaders in network
    leaders_.goHead();
    for (int i = 0; i<leaders_.getSize(); ++i)
    {
        Leader* curLeader = static_cast<Leader*>(leaders_.getValue());
        if (curLeader->GetName().find(partialName) != string::npos)
            cout << i + 1 << ") " << curLeader->GetName() << ": " << curLeader->GetEmail() <<
            endl;
        leaders_.getNext();
    }
}
```

```
#!/bin/bash
files=`find ./"$1" -type f -name "*.$2"`
# get all files that ends with .$2
for file in $files; do
  newname="${file%.$2}"
  newname="$newname.$3"
  mv "$file" "$newname"
done
# files converted!
exit 0
```