ask 1: Living on the Internet: the `Network` class

Define and implement the `Network` template class with a header and a source file. In this project you will design and implement the network as a "bag of accounts". Although the network will only ever hold and manipulate accounts, you will implement it as a template class for pedagogical reasons, i.e. so that you can practice writing a template class. You can refer to the `Bag` class (provided with the starter files for this project) to implement the bag-like container operations. What we call Bag functions below are mostly identical to the corresponding `Bag` operations but implemented for the `Network` class instead. These will vary whenever indicated in order to handle network- or account-specific operations. This may seem counterintuitive at times: you will be calling account-specific functions on an object of type ItemType, but it's ok because we are assuming that our Network will always be instantiated as a Network of Accounts (i.e. `Network<Account>`).

Private member variable:

- An array (with a capacity of 200) that stores POINTERS to the template items.

- A count of the current number of accounts in the network.

- A vector of Posts: the Network's feed where all posts posted by accounts in the network will be stored.

Private member Functions:

- A function that returns the index of a given account within the Network.

```
/**
   private function
   @param          :   string - the username of item we want the index of
   @return         :   int - the index of the item, -1 if the account is not found
*/
getIndexOf
```

Public member function:

Constructors:

```
   /**
      Default constructor.

      Initializes private variables to default initial values. */
```

Bag functions:

```
  /**
    Accessor function
    @return          : int -  the current size of the network (number of items in the array)
  */
  getSizeOfNetwork

  /**
    @return          : bool -  true if the bag is empty, false otherwise
  */
  isEmpty

  /**
    @param           : a POINTER to the item that will be added to the Network
    @return          : bool- true  if the item was successfully added, false otherwise
    @post            : stores a pointer to the item, if there is room. This is a network
specific function,
                  it has an additional constraint: the item will be added only if no other
item in the network
                  has the same username (usernames are considered unique)
                  REMEMBER WE ARE STORING POINTERS TO ITEMS, NOT ITEMS.
                  Finally, the Account being added to the network will update it's private
member to point to this network
                  (see Account class modifications below, you may come back and
implement this feature after
                  you work on Task 2, but don't forget!!!).
                  NOTE: every object in C++ has access to a pointer to itself called `this`,
thus the nework can pass `this` pointer to the account!
  */
  addAccount

  /**
    Mutator function
    @param           :   a POINTER to the item that will be removed from Network
    @return          :   true if if the item was successfully removed, false otherwise
    @post            :   updates the bag/network to remove the pointer, if a pointer to that
item was found.
  */
  removeAccount
  /**
     Mutator function
     @post            :   Empties the bag/network
  */
  clear

 /**
     @param           :   a POINTER to the item to find
     @return          :   true if the item was successfully found, false otherwise
  */
  containsAccount
```

Network Specific functions:

These functions are specific to the Network. These are not adapted from the Bag class, instead you must implement these from scratch.

```
  /**
     Mutator function
     @param           :   the name of an input file
     @pre            :   The format of the text file is as follows:
                       username password
                       username password
                       ;

                       where ';' signals the end of the file. Check the provided example
(accounts.txt).

     @post            :   Reads every line in the input file, creates an Account from that
information and
                       adds the Account to the network. If a line of input is missing some
required information,
                       it prints out "Improper Format" and terminates.
                       Keep in mind that, although the input will always be an Account, the
Network class is
                       a template, thus it will store a pointer to a generic type. It will do so by
creating a new
                       dynamic object of type ItemType but passing the account information
(username and password)
                       as to the Account constructor. This is a bit hacky, but it will work for
our Network.
```

*/
populateNetwork

/**
Mutator function
@param : a reference to another Network

@post : Removes from the Network any items that also appear in the other Network.
In other words, removes elements from the network on the left of the operator that
are also found in the network on the right of the operator.
Why is this useful? For example, given a network of known bot accounts, remove all bots from this Network.
*/
operator-=

Account class modification:

Now that accounts are part of a Network, you must also adapt the `Account` class so that accounts can follow one another.

Private member variable:

- A vector of usernames the Account is following
- A pointer to the Network this account is part of. Before the Account is added to a Network,
  The pointer is `nullptr`.

Public member functions:

/**
Accessor function
@return : the pointer to the Network the account is in
*/
getNetwork

/**
Mutator function
@param : a pointer to a Network
@post : the Network pointer private member points to the input Network
*/
setNetwork

ask 2: You seem like a nice person

We are now ready to allow accounts in a `Network` to follow one another.

The `Account` class must be further modified to incorporate the following *public member functions* :

```
  /**
     @param          :   the username of the Account to follow
     @return         :   true if the account was successfully able to follow, false otherwise
     @post           :   adds the username to the vector of following accounts only if
                        it is affiliated to a Network AND it is not already following an account
                        with the same username.
  */
  followAccount

 /**
     @return         :   the vector of usernames the Account is following
  */
  viewFollowing
/**
     MODIFY this function to also add the Account username to the Post and then add
the Post
     to its Networks feed if it is connected to one. NOTE: you will need to add this
functionality
     to the network as well (see Network modifications below - you may want to create a
STUB for it,
     or implement this functionality after you modified the Network.)

     @param title   : A reference to the title used to generate the Post object
     @param body    : A reference to the body used to generate the Post object
     @post          : generates a Post with the given title and body and adds it to it's vector
                   of posts AND to the Network's feed if it is connected to a Network.
     @return        : Will return true if the Post does not have an empty title or body and
the
                   Post is successfully added to the vector
  */
  addPost
```

The `Network` class must be further modified to incorporate the following *public member functions*:

```
/**
    Accessor function
    @param          :   a reference to the item whose feed will be displayed
    @post           :   prints the feed of the given account by checking who they are
following
                        and displaying all the posts from the feed that were made by those
accounts.
                        Keep in mind that the Network parameters are general template types
                        in this project rather than accounts, although this functionality is
                        specific to accounts.
*/
printFeedForAccount

/**
    @param          :   a reference to an item (account) and the username of the
account
                        it wants to follow
    @return         :   true if the item was authorized to follow, false otherwise

    @post           :   the referenced Account follows another account with the username
                        specified by the second argument if they both exist in the network
*/
authenticateFollow
/**
    Mutator function
    @param          :   a reference to a Post be added to the feed
    @return         :   returns true if the Post was successfully added to the feed, false
otherwise
    @post           :   Adds the post to its feed only if the Post was created by an
Account
                        in this Network.
*/
addToFeed
```

You are submitting

Account.hpp Account.cpp Network.hpp Network.cpp

```
#ifndef ARRAY_BAG_
```

ADDITIONAL FILE “ArrayBag.hpp” and “ArrayBag.cpp”

ArrayBag.hpp:

```
#define ARRAY_BAG_
#include <vector>

template <class ItemType>
class ArrayBag
{
public:
  /** default constructor**/
  ArrayBag();

  /**
    @return item_count_ : the current size of the bag
  **/
  int getCurrentSize() const;

  /**
    @return true if item_count_ == 0, false otherwise
  **/
  bool isEmpty() const;

  /**
    @return true if new_entry was successfully added to items_, false otherwise
  **/
  bool add(const ItemType &new_entry);

  /**
    @return true if an_entry was successfully removed from items_, false otherwise
   **/
  bool remove(const ItemType &an_entry);

  /**
    @post item_count_ == 0
   **/
  void clear();

  /**
    @return true if an_entry is found in items_, false otherwise
    **/
  bool contains(const ItemType &an_entry) const;

  /**
   @return the number of times an_entry is found in items_
  **/
  int getFrequencyOf(const ItemType &an_entry) const;
protected:
  static const int DEFAULT_CAPACITY = 10; //max size of items_ at 10 by default for this project
  ItemType items_[DEFAULT_CAPACITY];      // Array of bag items
  int item_count_;                        // Current count of bag items

  /**
     @param target to be found in items_
     @return either the index target in the array items_ or -1,
     if the array does not contain the target.
     **/
  int getIndexOf(const ItemType &target) const;

}; // end ArrayBag

#include "ArrayBag.cpp"
#endif
```

ArrayBag.cpp:

```
#include "ArrayBag.hpp"
#include <iostream>

/** default constructor**/
template<class ItemType>
ArrayBag<ItemType>::ArrayBag(): item_count_(0)
{
}  // end default constructor

/**
 @return item_count_ : the current size of the bag
 **/
template<class ItemType>
int ArrayBag<ItemType>::getCurrentSize() const
{
	return item_count_;
}  // end getCurrentSize

/**
 @return true if item_count_ == 0, false otherwise
 **/
template<class ItemType>
bool ArrayBag<ItemType>::isEmpty() const
{
	return item_count_ == 0;
}  // end isEmpty

/**
 @return true if new_etry was successfully added to items_, false otherwise
 **/
```

```
template<class ItemType>
bool ArrayBag<ItemType>::add(const ItemType& new_entry)
{
        bool has_room = (item_count_ < DEFAULT_CAPACITY);
        if (has_room)
        {
                items_[item_count_] = new_entry;
                item_count_++;
     return true;
        }  // end if

        return false;
}  // end add

/**
 @return true if an_etry was successfully removed from items_, false otherwise
 **/
template<class ItemType>
bool ArrayBag<ItemType>::remove(const ItemType& an_entry)
{
  int found_index = getIndexOf(an_entry);
        bool can_remove = !isEmpty() && (found_index > -1);
        if (can_remove)
        {
                item_count_--;
                items_[found_index] = items_[item_count_];
        }  // end if

        return can_remove;
}  // end remove

/**
 @post item_count_ == 0
 **/
template<class ItemType>
void ArrayBag<ItemType>::clear()
{
        item_count_ = 0;
}  // end clear

/**
 @return the number of times an_entry is found in items_
 **/
template<class ItemType>
int ArrayBag<ItemType>::getFrequencyOf(const ItemType& an_entry) const
{
  int frequency = 0;
  int cun_index = 0;       // Current array index
  while (cun_index < item_count_)
```

```
   {
      if (items_[cun_index] == an_entry)
      {
         frequency++;
      }  // end if

      cun_index++;          // Increment to next entry
   }  // end while

   return frequency;
}  // end getFrequencyOf

/**
 @return true if an_etry is found in items_, false otherwise
 **/
template<class ItemType>
bool ArrayBag<ItemType>::contains(const ItemType& an_entry) const
{
            return getIndexOf(an_entry) > -1;
}  // end contains

// ********* PRIVATE METHODS **************//

/**
            @param target to be found in items_
            @return either the index target in the array items_ or -1,
            if the array does not containthe target.
 **/
template<class ItemType>
int ArrayBag<ItemType>::getIndexOf(const ItemType& target) const
{
            bool found = false;
  int result = -1;
  int search_index = 0;

   // If the bag is empty, item_count_ is zero, so loop is skipped
   while (!found && (search_index < item_count_))
   {
      if (items_[search_index] == target)
      {
         found = true;
         result = search_index;
      }
      else
      {
         search_index++;
      }  // end if
   }  // end while
   return result;
}  // end getIndexOf
```