Aiman Bickiya

UCLA ID: 105339123

Description of algorithms and data structures:  
**Info struct:** This struct is the same as the one in the slow version of the program. It stores the name of a user, the name of a chat, and an int that represents the user’s contributions to that chat. The constructor takes two strings, one for the user and one for the chat, and initializes the member variables with them. The count variable is always initialized as 0.

**Chat struct:** This struct stores a string to represent a chat name, and an int that represents the total contributions to the chat. The constructor takes a string and initializes it to the name member variable, and the count is initialized as 0

**info (vector of linked lists of Info objects):** This vector functions as the hash table for Info objects, where the bulk of the information for the program is stored. When trying to store or access an object, the index is obtained by using the hash template class from the <functional> library with the user string, which is then divided with the % operator and the number of max buckets given during construction. To handle collisions, each index of the vector is a separate, doubly linked list (standard list type) of Info objects. Collision handling is vital because each chat that a user is in creates a separate Info object, but it’s hashed by the name of the user which is the same.

**chats (vector of linked lists of Chat objects):** This vector functions as the hash table for chat objects. The index is obtained by using the hash template class from the <functional> library with the chat string, which is then divided with the % operator and the number of max buckets given during construction. This hash table also has collision handling by making each index of the vector a separate linked list (standard list type) of Chat objects. Although in this case, the collision handling is for any collisions that may happen by chance- there are no ‘intentional’ collisions as there are with the info vector.

**ChatTrackerImpl(int maxBuckets):** As part of the initializer list, this function initializes the given value of maxBuckets to the private member variable buckets. It also resizes the vectors ‘info’ and ‘chats’ to the value of buckets.

**initializeChat(string c):**

Calls the hash function on the given string (which represents a chat name)

If the linked list at that index is empty

create and pushback a new Chat object created with the string c

exit

Else

iterate through the linked list at that index

if there is a chat object who’s name is the same as c

exit (a chat with that name already exists)

create a chat object with c as the parameter and push it to the linked list

exit

The purpose of this function is to make sure a Chat object exists in the chats hash table, and if it doesn’t, the function will create one. This is necessary so that there is no trouble finding a chat when we want to increase its contributions count.

**join(string user, string chat):**

Calls the hash function on user

If the linked list at that index of the info vector is empty

create and pushback a new Info object with parameters user and chat

exit

Else

iterate through the linked list at that index of the info vector

if there is an Info object with the same user and same chat

move that Info object to the front of the linked list

exit

create a chat object with c as the parameter and push it to the linked list

exit

The purpose of this function is to create an Info object for a given user and chat, or if it already exists, to make it the user’s current chat by moving it to the front of the linked list.

**chatContribute(string c):**

Calls hash function on c

Iterate through the linked list at that index of the chats vector

If Chat object’s name is equal to c

increment count variable by 1

exit

The purpose of this is to keep track of and increment a chat’s total number of contributions.

**contribute(string user):**

Calls hash function on user string

Iterates through linked list on that index of the info vector

If the user of an Info object matches the given user string

Increment that Info objects count variable

call chatContribute with the chat stored in that info object

return that Info object’s count

return 0 //failsafe

This function increments the count variable for the user’s current chat (because of the way the join function is structured, this will always be the first Info object with a matching user in the linked list). This calls another function to increment the chat’s total contributions as well.

**leave(string user, string chat):**

Calls hash function on user string

Iterate through the linked list at that index of the info vector

If the Info objects user and chat match the passed parameters

Store the value of count in a new variable

Erase that Info object from the linked list

return the value of count

return -1 //error

This function deletes the Info object for a given user and chat.

**leave(string user):** Works in the same way as the other leave function, but it deletes the first Info object with a matching user. Because of the way join is structured, the first object with a matching name will always be the user’s current chat.

**terminate(string chat):**

For each index of the info vector

Iterate through the linked list at that index

If the info object’s chat is equal to the chat parameter

erase that Info object from the linked list

Call the hash function on the given chat string

Iterate through the linked list at that index of the chats vector

If the Chat object’s is equal to the chat parameter

store the value of that object’s count variable in a new variable

erase that Chat object from the linked list

return the new variable that has the count value

return 0 //failsafe

This is the most time consuming function because it has to go through all the nodes of each linked list in the vector.

Notes:

At first I attempted to modify the Info struct to hold a vector of Chat objects, so that each chat a user joins would not have to create a new Info object. Although there might be some merit to this approach, have so many nested structures (a vector of linked lists of Info objects which contains a vector of Chat objects) became too unwieldy to work with and I adopted the same general approach for determining a user’s current chat as was used by the slow version of the program.