Ryan Foundoulis

605107291

Project 4



**Structure:**



My setup is largely dependent on two large hash tables. Both of these being vectors that hold lists within each slot. Each position within those lists holds a pointer to the Info struct that was already defined within the .cpp file. Essential, in the user hash, we use the user’s name as the input for the mapping function. The list that we are brought to then will hold pointers to all of the chats that the user is associated to. In the case above, Gary’s list has two slots, one that has a pointer to his Info associated with the dog chat and one with the cat chat. Fred has one pointer in his list associated with the dogs chat. In the other hash we have it almost in a reverse organization. Our chat name is used to map to a specific spot within the vector that holds a list of pointers that keep track of the different users within that chat. In the example above, the list for dogs holds pointers for Fred and Gary while the cats list holds a pointer to Gary’s cat Info. The numbers with the info are the number of contributions that an individual has made within that chat.

I chose to make two independent structures that organize the data within Info different to help address the different ways that the functions need to access and alter the data. For example, when telling a user to join, contribute, or leave a chat, we must be able to get access to all of the user’s chats. While calling terminate, we have to get access to all of the users that are associated with it. The reason I decided to have the Info structs be their own independent objects that are pointed to by the pointers held in the lists in the vectors is because the program will only have to create one info object that both the hash table structures are able to access. I originally tried to create two hash tables that actually held the Info struct within them, but I quickly realized that the amount of extra storage and processing required to do it that way was unreasonable.

**Algorithms:**

**Join -** The join function is pretty simple. We start by finding the bucket associated with the user and iterating through each slot in that user’s list. If we find the chat associated with that user, we will erase the pointer to it from the list and push a new pointer pointing to the same struct to the front.

If we do not find a pointer associated with that chat, we create a new Info struct and push a pointer to it in both the user’s bucket in the hash table and to the chat’s bucket in the chat hash table.

**Terminate -** The terminate function will start by finding the bucket associated with the inputted chat and initialize a totalCount int to zero. The function will then iterate through each of the users in the chat, add their count to total count, have that user leave from the chat (to get rid of the pointer from the user’s side), and then delete that Info Struct, and then erase that position from the list.

**Contribute -** Contribute will find the bucket associated with the specific user and if the bucket is not empty, it will add one to the count of the first Info struct that has the same user associated as we are looking for (since collisions may occur in the bucket) and return the new count value. If it is empty, or this user does not have any chats its associated with it will return 0;

**Leave(user) -** This function will go to the bucket associated with the user and iterate through to the first Info struct associated with that user (again, collisions), it will store the value of that user’s contributions to that chat, erase that pointer from the list and return the stored count. If there is no chat found it will return -1;

**Leave(user, chat) -** This function will go to the bucket associated with the user and iterate through to the Info struct associated with that user and chat, it will store the value of that user’s contributions to that chat, erase that pointer from the list and return the stored count. If that chat is not found it will return -1;

2. Pseudo code;

**Join(user,chat)** {

Create iterator at the start of the user’s list;

While the iterator does not equal the end of the list {

If the user at iterator equals user and their chat equals chat {

push a pointer to this struct to front of list;

Erase the current iterator;

Return;

}

Increment the iterator;

}

Create new Info struct with user and chat

Add pointer to this struct to the start of that user’s chat list and that chat’s user list;

}

**terminate(chat)** {

Create iterator at the start of the chat’s list;

While the iterator does not equal the end of the list {

If the iterate->chat equals chat {  
 At that iterators count to total count;

leave(iter->user,chat);

Delete the Info Struct iter points to;

Erase iter;

}

Else {

Increment the iterator

}

}

Return totcount;

}

**Contribute(user)** {

Create iterator at the start of the user’s list;

While the iterator does not equal the end of the list {

If the iter’s user equals user {

Increment iter’s count;

Return the new count;

}

Increment the iterator;

}

Return -1;

}

**Leave(user, chat)** {

Create iterator at the start of the user’s list;

While the iterator does not equal the end of the list {

If the iter’s user equals user and iter’s chat equals chat {

set count int to iter’s count;

Erase iter from the list;

Return count;

}

Increment iter;

}

Return -1;

}

**Leave(user)** {

Create iterator at the start of the user’s list;

While the iterator does not equal the end of the list {

If the iter’s user equals user {

set count int to iter’s count;

Erase iter from the list;

Return count;

}

Increment iter;

}

Return -1;

}

**~ChatTrackerImpl()** {

For k = 0 and k is less than chat vector size, increment k each time {

Create iter at chat bucket k’s beginning;

While iterator does not equal chat bucket k’s end; {

Delete what iterator is pointing to;

Erase iter;

}

}

}

3. There are no known bugs in the way the code runs and it consistently runs on g32fast at just about 46 ms. The only issue I really had was continuously reworking the code and setup of the structures until they were able to run the processes fast.