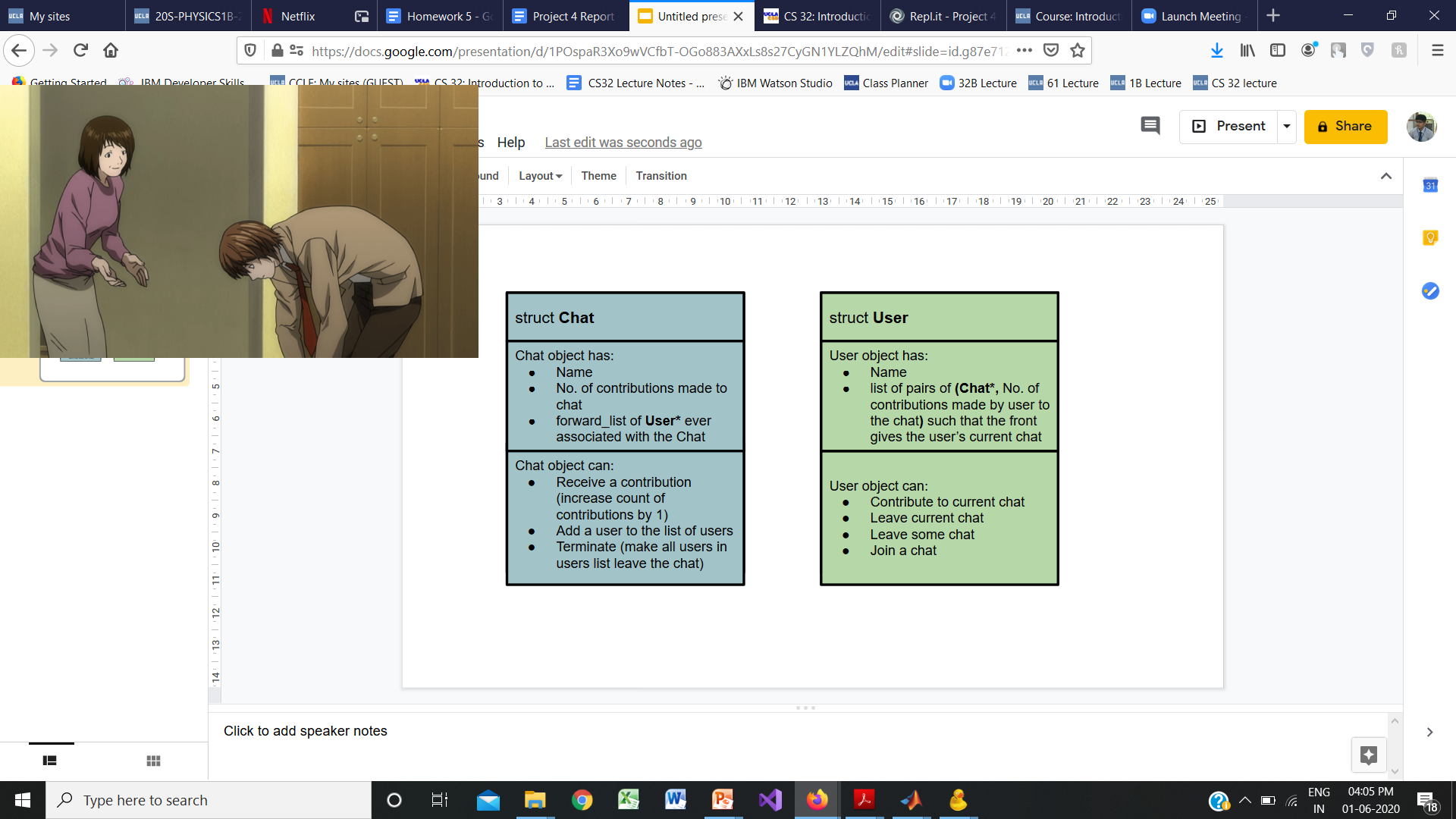
**Report**

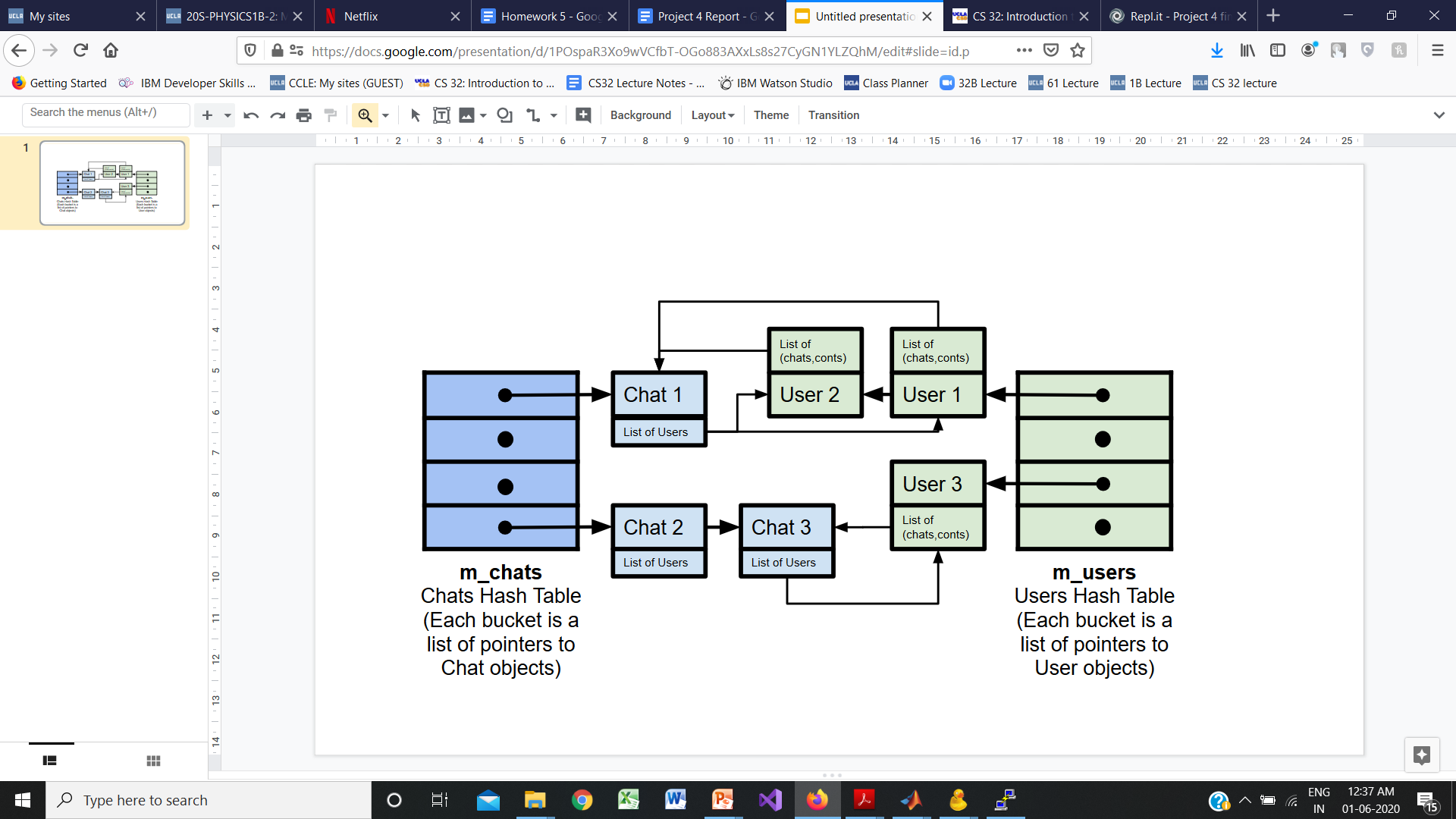
Author: Jagrit Digani

**Algorithms and Data Structures**



Given below is a list of data structures and algorithms used in the **Chat** and **User** structures, and the application motivating their use:

* Each **User** maintains a **list** of **pair<Chat\*, int>**
  + **pair**: Allows each user to keep track of contributions made to each chat joined
  + **list**: Allows each user to do the following with low cost:
    - Associate chats with them (join chats)
    - keep track of the order in which the chats were joined (front of the list always gives the current chat), and
    - leave any chat or make any chat already in the list the current chat
  + **Algorithms**: Since each user is not expected to join many chats (with only a few exceptions), the chats are only kept in the order they were joined in, and so a **linear search** is performed to locate a chat in the user’s list
* Each **Chat** maintains a **forward\_list** of **User\***
  + **forward\_list**:
    - A chat needs to access the list of its user only when being terminated
    - Assumption: user leaving a chat far more common than a chat being terminated
    - Hence, user not removed from chat’s list upon leaving (only chat removed from user’s list) to make leaving more efficient (calling leave on a user no longer associated with a chat does nothing)
    - Since only application of list is insertion to add user, and visiting all nodes on termination, a simple forward\_list is used since more functionality is not needed
  + **Algorithms**: No significant algorithms applied, since the only uses are insertion to list add user, and visiting all nodes on termination of chat.



Given below is a list of data structures and algorithms used in the **ChatTrackerImpl**, and the application motivating their use:

* **Hash Tables** of (dynamically allocated) **User\*** and of **Chat\*** are used
  + user’s name (string) and chat’s name (string) are used as keys to index the 2 hash tables
  + hash function from the C++ library functional is used (% maxBuckets to get bucket no.)
  + The hash tables themselves are **vectors** constructed to have maxBuckets no. of buckets
  + Each bucket is a **list** of **User\*** or **Chat\*** to accommodate for collisions
* **Algorithms**: hash function from the C++ library functional is used, % maxBuckets to get bucket no. and **linear search** is performed in each bucket to find the **Chat\*** or the **User\*** given the name (since each bucket is not expected to have too many members, and will be unordered)

**Pseudocode for non trivial algorithms**

void User::joinChat(Chat\* chat)

*for each member of the chat lists*

*if the chat to join already in list*

*remove the Chat contributions pair from its old place in chats list*

*move the pair to the front of the chat list // make current chat*

*return*

*// if user not already associated with the chat*

*make pair of chat to join with 0 contributions*

*push pair to the front of the chats list*

*add user to the chat*

int User::leaveChat(Chat\* chat)

*for each member of the chat lists*

*if the chat to join already in list*

*note the contributions by user associated with this chat*

*remove the relevant pair from the chats list*

*return the contributions by user associated with this chat*

*// if user not already associated with the chat*

*return -1*

int User::leaveCurrChat()

*if chats list is empty // User has no current chat*

*return -1*

*remove the first pair from the chats list*

*return the contributions by user associated with this chat*

Chat\* ChatTrackerImpl::findChat(const string& chatName)

User\* ChatTrackerImpl::findUser(const string& userName)

*find the bucket to which the user or chat belongs*

*perform a linear search to get pointer to the user or chat*

*return nullptr if no such user*

void ChatTrackerImpl::join(const string& user, const string& chat)

*find Chat object with the same name in hash table*

*if no such chat*

*make new Chat object and insert to hash table*

*find User object with the same name in hash table*

*if no such user*

*make new User object and insert to hash table*

*make User join the chat*

int ChatTrackerImpl::terminate(const string& chat)

*find Chat object with the same name in hash table*

*if no such chat*

*return 0*

*make the Chat terminate itself // make all its users leave*

*remove the chat from the hash table*

*deallocate the chat*

*return the total contributions made to the chat*