



**BINUS UNIVERSITY  
BINUS INTERNATIONAL**

**Assignment Cover Letter  
(Group Work)**

Student Information:	Surname	Given names	Student ID Number
1.	Yowen	Yowen	2301902390
2.	Edgari	Eric	2301902352
3.	Lucianto Santoso	William	2301890390

**Course code : COMP6571**

**Class : L2AC**

**Major : CS**

**Course Name : Data Structures and Algorithms**

**Name of Lecturer(s) : 1. Andreas Kurniawan**

**Title of Assignment : How to efficiently manage a general ID system**  
(if any)

**Type of Assignment : Final Project**

**Submission Pattern**

**Due Date : 18-06-20      Submission Date : 18-06-20**

The assignment should meet the below requirements.

1. Assignment (hard copy) is required to be submitted on clean paper, and (soft copy) as per lecturer's instructions.
2. Soft copy assignment also requires the signed (hardcopy) submission of this form, which automatically validates the softcopy submission.
3. The above information is complete and legible.
4. Compiled pages are firmly stapled.
5. Assignment has been copied (soft copy and hard copy) for each student ahead of the submission.

**Plagiarism/Cheating**

BiNus International seriously regards all forms of plagiarism, cheating and collusion as academic offenses which may result in severe penalties, including loss/drop of marks, course/class discontinuity and other possible penalties executed by the university. Please refer to the related course syllabus for further information.

**Declaration of Originality**

By signing this assignment, I/we\* understand, accept and consent to BiNus International terms and policy on plagiarism. Herewith I/we\* declare that the work contained in this assignment is my/our\* own work and has not been

submitted for the use of assessment in another course or class, except where this has been notified and accepted in advance.

Signature of Student:	(Name of Student)
Yowen	Yowen
Eric	Eric Edgari
William	William Lucianto Santoso

## Problem description

The problem our group is trying to tackle is an efficient method for managing a general id system. For the sake of this project, we decided a simple id-name key pair is sufficient for the data. Like any other id management system, the program should be able to manipulate data based on its id (add, delete, search). An added functionality we deem to be important is it should also be able to search data based on the entry's name. We feel like this part is important considering the fact that the data consists solely of key-pair values. Thus, the main problem lies in how to develop a program using appropriate data structures to efficiently manipulate the data in an id management system by both its id and the entry's name.

## Possible alternative data structures

### 1. Unsorted Linked List

It is possible to create this program by simply making use of a single linked list. We can implement the node structure with an int for the id and string for storing the name. To manipulate the data with this data structure, the program will have to perform a linear search through the entire list to find whether the data matches or not. This gives us a time complexity of  $O(n)$  for searching which is inefficient as the data gets larger. Similarly, when deleting, the program will still need to perform a linear search to locate the specified data before deletion thus resulting in yet another time complexity of  $O(n)$ . In both cases, we assumed the worst case scenario but if we consider the average, we can obtain a time complexity for  $O(n/2)$  which would usually be considered  $O(n)$ , but for the sake of comparison we will keep it as it is. For addition however, since we do not need to sort the list, we can simply add the new data as the new head/tail thus giving us a time complexity of  $O(1)$ . The takeaway here is that this method is possible and can be very fast for a small amount of data, however, as the data gets larger, searching and deleting items from the list may take some time compared to other methods.

### 2. Sorted Linked List

When talking about a sorted linked list, the time complexity can dramatically change compared to an unsorted linked list. Similarly like the unsorted linked list, the node structure would remain the same, however, we can implement a sorting algorithm whenever a new item is added to the list to sort the list by id. By sorting the data based on id, instead of having to perform a linear search, we can perform a much more efficient binary search. With binary search, we are able to search data by splitting it in half thus reducing time complexity from a worst case of  $O(n)$  to  $O(\log n)$ . The great thing about binary search is that its worst case is its average case thus both worst and average results in a time complexity of  $O(\log n)$  which is an improvement. The same thing applies to deletion in a sorted linked list. Since we need to search for the data first, the complexity of deletion for both worst and average is  $O(\log n)$ . For addition, since we need to search where the data needs to be placed on the list, the time complexity changes to  $O(\log n)$ ,

however, after we locate the position where it needs to be inserted, we still need to traverse through the list thus its true time complexity is actually  $O(n)$  with an average of  $O(n/2)$  which is worse compared to an unsorted linked list's  $O(1)$ . This method looks prominent until we go back to our problem description and realize that our program is supposed to also be able to search data by name. This throws all of our previous boasting out the window since now we are back at square one using linear search with time complexity of  $O(n)$ . The takeaway here is that this method is decent for deletion and searching by id, however, it is not efficient when the user searches by name and we also believe that we still can improve its addition time complexity with the use of a different data structure.

### 3. Hashmap with linear probing

When using a hashmap, we can take 2 approaches to this matter. We can either use direct hashing to use the id as the key then store the name within it or we can convert the name into an integer using ascii code which we will use as the key then store the id within it. Both options have their own advantages and disadvantages which we will discuss about. If we go with the first option, addition of new data into the database becomes efficient with a time complexity of  $O(1)$  since the program does not need to search for anything, even on the case where the id already exists, a simple function to determine whether a specified bucket is occupied can be implemented with a time complexity of  $O(1)$  which will completely nullify the downsides of linear probing. For searching, since we can guarantee every key is unique, it also results in an impressive time complexity of  $O(1)$ . Same thing applies to deletion and so far, this method seems to be the fastest one yet with a constant time complexity of  $O(1)$  for all addition, deletion and searching. The problem comes when the user wants to search data by name. It is not possible to search the data by name. Since we are using direct hashing, to search data by name means to search the entire hashmap. This gives us a time complexity of  $O(n)$  with  $O(n/2)$  as average.

It is difficult to talk about the second method in general as its performance is often dictated by its hashing function. In a perfect world, where every new data added gets hashed to a different bucket, the complexity for addition is an impressive  $O(1)$  since all the program has to do is to apply the hashing function to convert the name into the key. The same applies to searching and deleting data since this 'perfect algorithm' is able to allocate a new bucket every time. However, it is not realistic to assume our hashmap is going to always perform perfectly. It is very likely that multiple data will equvalate to the same key and a linear probe needs to be performed. So realistically speaking, our addition would have a time complexity of  $O(n)$  due to linear probing. Similarly, for searching data, we need to assume that a linear probe process needs to be done to obtain the data thus resulting in another time complexity of  $O(n)$ . Same thing applies to deletion thus leaving us with a time complexity of  $O(n)$  for all 3 processes which is not impressive. Another problem with this method is the fact that you will be unable to track whether a specified id already exists within the database. To fix this issue a new list will need to be implemented to store all existing ids and a function will need to be created to search through the list. This means a linear search has to be done within the list first prior to adding the data so even if we take the best case scenario of  $O(1)$ , the program still needs to check whether the specified id already exists within the list thus the time complexity remains  $O(n)$ . One can argue the fact that we can instead use a sorted list for storing the list of ids thus enabling us to perform binary search instead of linear search but we have to keep in mind when adding new ids to the sorted list, we still have to traverse through the list to the designated index thus the time complexity still remains  $O(n)$ . Lastly, by using this method, we are unable to search the data by

id efficiently since we would be forced to perform a linear search through all the contents of the hashmap leaving us with yet another time complexity of  $O(n)$ .

Another problem with the use of linear probing in general is that a predetermined amount of buckets needs to be set. This means the program has a limited amount of data it can store and is not dynamic. Lastly, since a predetermined amount of buckets is set, the program will consume much more memory than it needs since empty buckets which the user does not need will still consume memory from the computer. The takeaway from this method is that if we use the first method, that is direct hashing, this program has potential to be one of the fastest in terms of manipulating data by id, however, it is not that impressive when it comes to searching by name and if we use the second method, with a good hashing function, it might be decent at manipulating data by name, yet it remains unimpressive when it comes to manipulating data by id but all in all, the fact that the program will not be dynamic and the fact that a predetermined amount of memory is going to be consumed by the program encourages us to stray from this path.

### **Theoretical analysis**

After taking into consideration the possible alternative solutions to this matter, we decided that if we want to actually create a program that is capable of efficiently handling both manipulation by name and id, we would need a much more complex data structure and maybe employ multiple types of it. After some consideration, we have come up with a solution, that is to employ a combination of an AVL Tree for manipulation with id, and a hashmap using chaining instead of linear probing for manipulation with name.

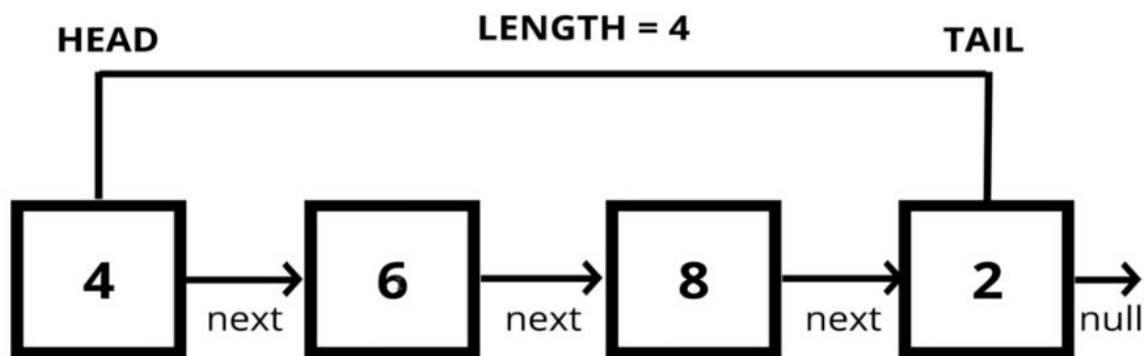
#### **AVL Tree**

An AVL Tree is an auto-balancing binary search tree. It is able to automatically manipulate the leaf nodes contained within and restructure itself to keep the shape of the tree balanced and ideal. This is a great implementation of a tree as it nullifies the major downfall of binary search trees which is the fact that an unbalanced binary search tree is inefficient. In a worse-case scenario, traversing through data on a binary search tree can take up a time complexity of  $O(n)$  in which all the data are only being added to either the left side or the right side. This means that it is pretty much just acting like a normal linear unsorted linked list when instead, it's supposed to instead be an efficient sorted data structure. An AVL tree prevents this by always keeping itself balanced. This means that the height of the tree will always be minimum and by using the formula  $\text{minimum height} = \log_2 n$ , we can essentially keep the time complexity of traversing through the tree  $O(\log n)$ . Now that is impressive, being able to add, search and delete data by id with logarithmic time but this is still not better when compared to the first method of hashmap whose time complexity is a staggering  $O(1)$  as stated above. The main selling point of this data structure is its ability to maintain the data in a sorted manner. On the GUI side of our application, we make use of 2 vectors containing a list of ids and a list of names. These 2 vectors are kept sorted so that our application can display all the entries in a sorted manner. When using an AVL tree, this is extremely easy to do as we can simply traverse through the tree in order. When using a hashmap however, we would probably have to traverse through the entirety of the database as hashmaps, in general, are not meant to be sorted. By using an AVL tree, we can also avoid unnecessary consumption of memory as our program will not preallocate a set amount of memory, it will only use up the required amount of memory to support the current amount of data. Another benefit of using an AVL tree compared to the linear

probing hashmap is that it is dynamic in nature. The user can add a theoretical infinite amount of data so long as they have enough memory thus there is no longer a static predefined max amount of data possible for the program. The main selling point of this data structure for us is its ability to keep itself sorted, able to manipulate data by id with decent efficiency (faster than a sorted linked list but not as fast as the hashmap), only taking up enough memory for the program (unlike the hashmap) and being a dynamic data structure thus not having a limit to max amount of entries.

Linked list is a linear data structure where each element is a separate object object.the elements are linked using pointers. Each node of a list is made up of two items, the data and a reference to the next node. The last node has a reference to null. For this project, the data will be string name, and integer id.

## Singly Linked Lists



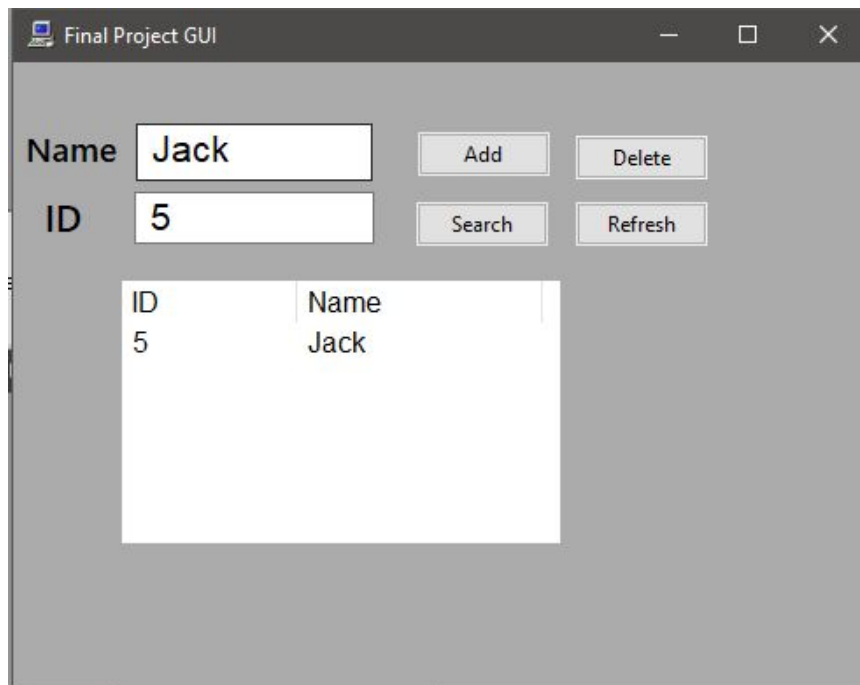
The singly linked list is implemented using the node structure with an int for the id and string for storing the name. For easier data manipulation we input some functions in the linked list such as the function to add a new node with name and id inside of it, the search function using the string parameter for the name we want to search, the delete function to delete the node that have the name we search using the string in the parameter with the search function, and the function to sort the data from the name. This linked list will be implemented in the hashmap where there will be 20 table size of linked list.

The hashmap will use a hash function that calculates the index by inputting the name and the function will automatically convert the name into a number and then mod 20 because the hashmap will have 20 table sizes of the linked list, to get the index of the linked list. After getting the index number we can add the data into the linked list we want by using the function in the linked list file. This hashmap will also have one extra linked list that will store all the data in every linked list so we can get all data sorted in one linked list. For the function we use the linked list function and modify it so it will automatically use the hash function to get the index so we can search and delete an item in the linked list.

The hashmap will have a faster time to search by name because we can get the index number where the name is stored so we don't need to search every data in every linked list where there are 20 linked list in our program. This hashmap can also save more time when there are many data stored in our program for example if we have 100 data which contain name and id, usually to search the data we want we need to search all 100 of them until we find the data we need, but with this hashmap the data will stored in 20 different location so by getting the index we will only search 1 linked list from 20 so the data we actually search will not be 100 but much more less.

## Program manual

This is our GUI program when it's running



Inputting Data : Enter ID and Number , ID cannot be the same as the one that's already been inputted to database , and ID cannot be longer than 999999999 integers

Final Project GUI

Name

ID

ID	Name
----	------

Click add button , and the data is inputted to the database


Final Project GUI

Name

ID

ID	Name
----	------

Delete

 Add data succesful

Final Project GUI

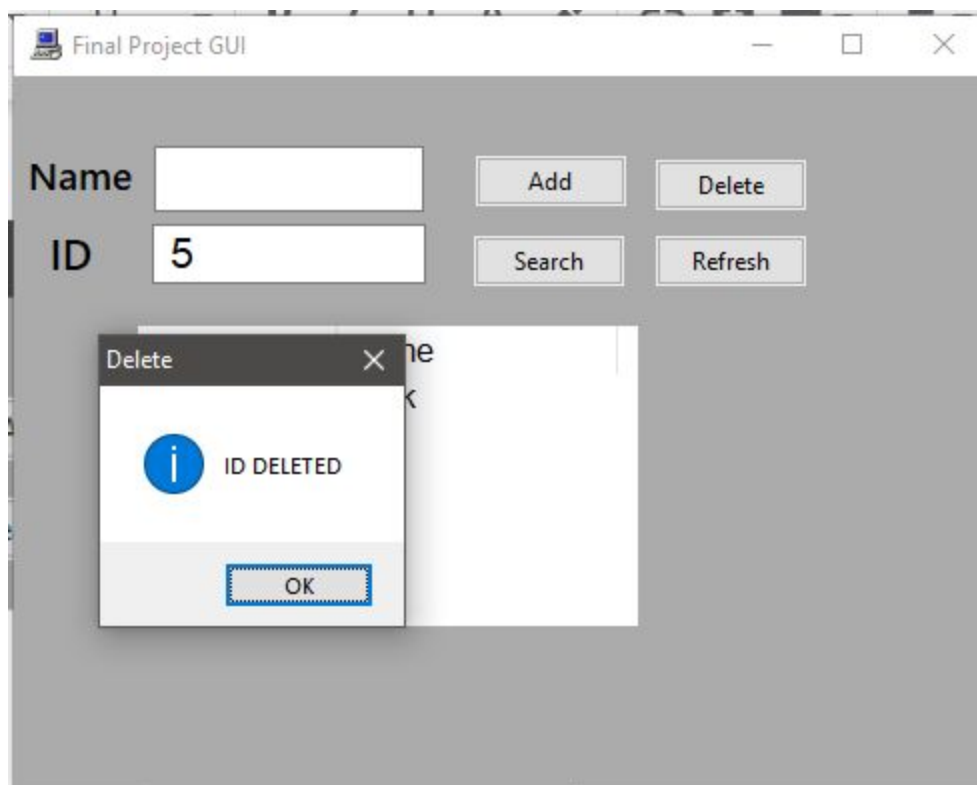
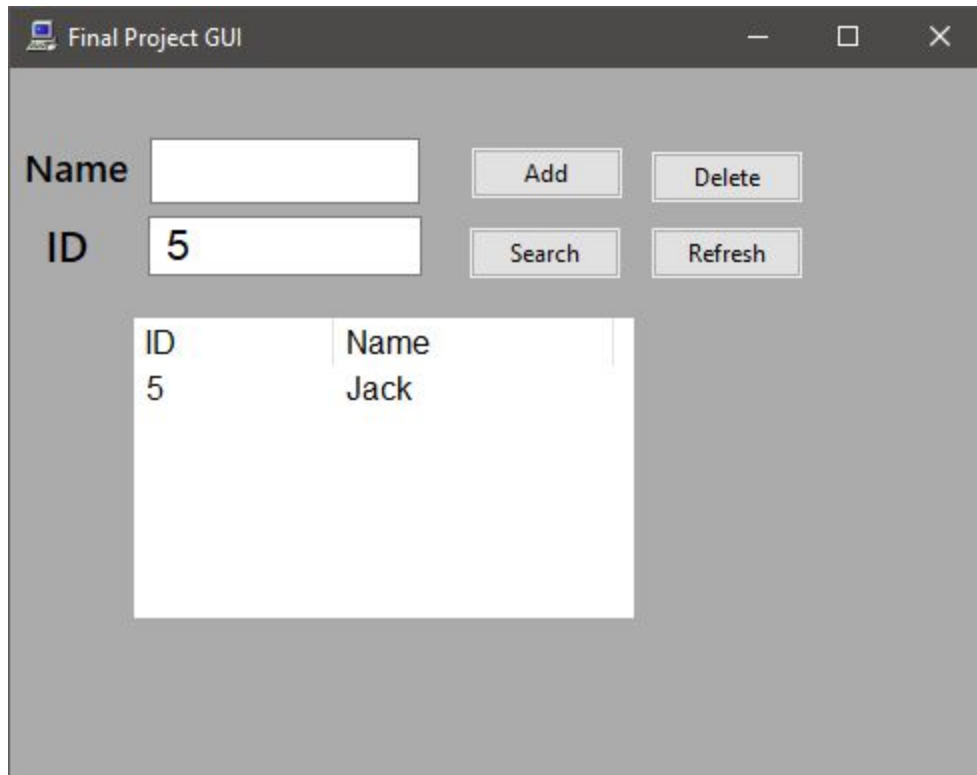
Name

ID

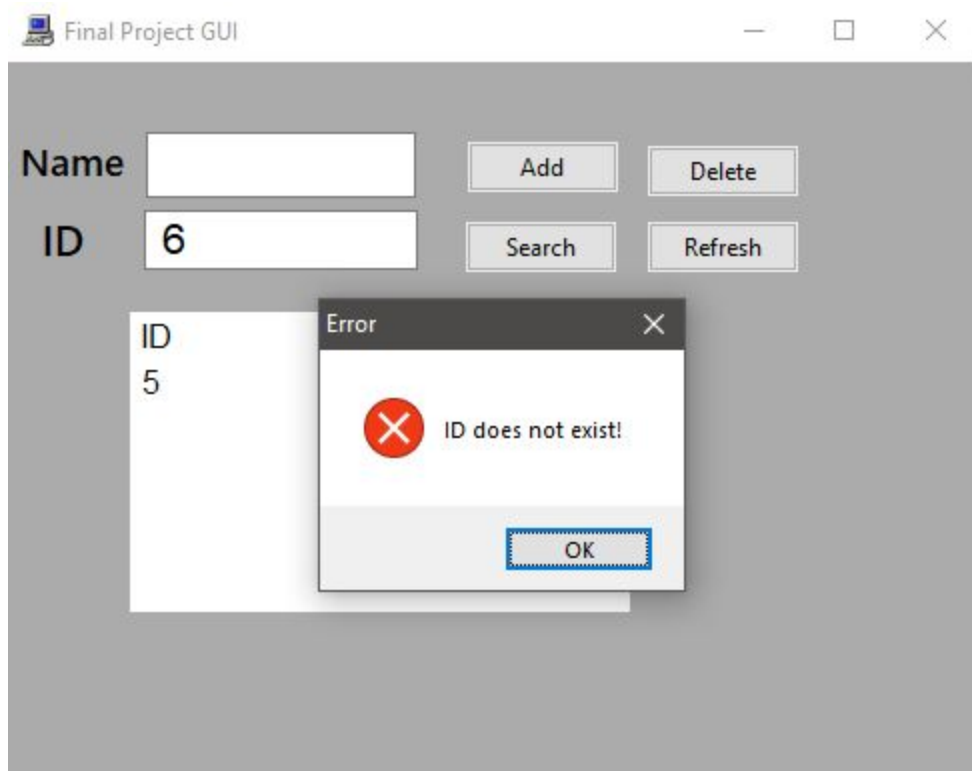
ID	Name
5	Jack

Deleting by , inputting only ID , and then click Delete button



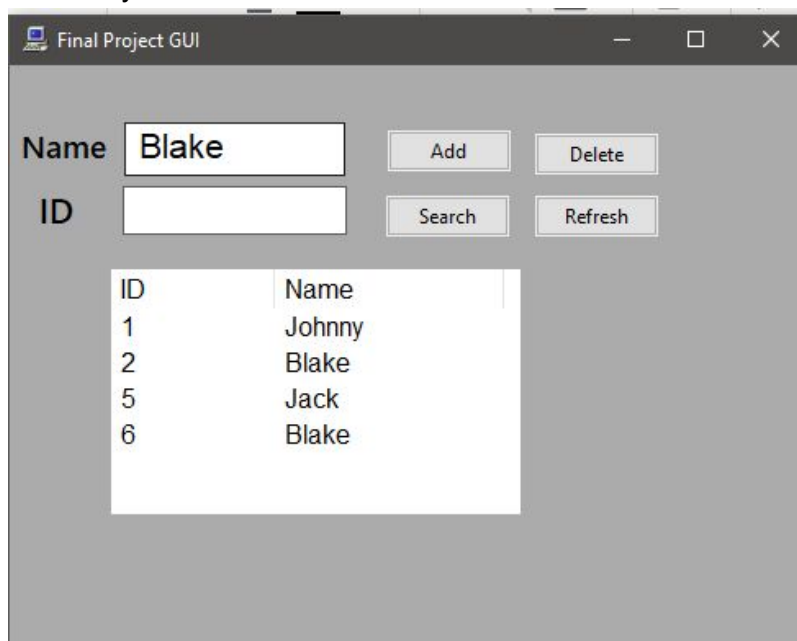


Error when id not exist



Enter name to the name field if want to search by name , or enter name to the id field if you want to search by id

Search by name



Final Project GUI

Name

ID

ID	Name
2	Blake
6	Blake

Click refresh to get the sorted list box

Final Project GUI

Name

ID

ID	Name
1	Johnny
2	Blake
5	Jack
6	Blake

## Search by ID

Final Project GUI

Name

Add Delete

ID

Search Refresh

ID	Name
1	Johnny
2	Blake
5	Jack
6	Blake

Final Project GUI

Name

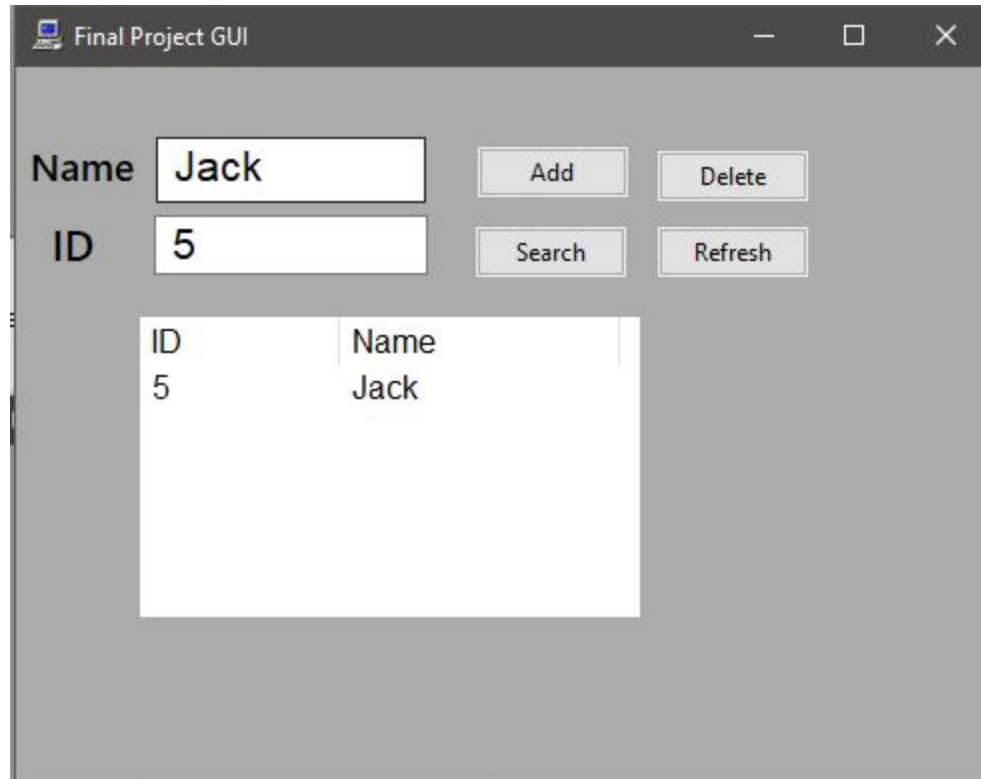
Add Delete

ID

Search Refresh

ID	Name
5	Jack

## Result



### Team contribution

Eric: I work on the frontend of this project , I am responsible for integrating the backend function to the Graphical user interface , and tried to make it as simple as possible . Because mainly the real life scenario that we are aiming for is people with minimal knowledge on how to use computer , with that in mind it becomes my main goal in making this GUI .

William: I work on the backend of this project mainly on the linked list and then the hashmap . I also contribute to the theoretical analysis that explains the linked list and the hashmap of this documentation.

Yowen: I work on the backend of this project mainly on the AVL tree. I also contributed to the possible alternative data structures and the AVL theoretical analysis of this documentation.

### Link to app demo

#### [Video Demo](https://drive.google.com/file/d/17qPBk1W5_R_mn0Y8ln39sLdnRbPzp_CJ/view?usp=sharing)

[https://drive.google.com/file/d/17qPBk1W5\\_R\\_mn0Y8ln39sLdnRbPzp\\_CJ/view?usp=sharing](https://drive.google.com/file/d/17qPBk1W5_R_mn0Y8ln39sLdnRbPzp_CJ/view?usp=sharing)

### Link github

<https://github.com/dankpanda/final-project-cpp>