DSA Final Project Report

1 The team members' names and school IDs

宋子維 B03902042 謝致有 B03902044

2 How you divide the responsibilities of the team members

宋子維: Main program, Operations that are not bound to last successful login, **trie implementation**.

謝致有: Operations that are bound to last successful login, writing the report, **self-implementation of unordered_map**.

3 The data structures you compared, including the results submitted to the mini-competition site

Map: 375660 (0.043s)

Unordered_map: 414026 (0.034s)

Self-implement of Unordered_map: 397332 (0.036s)

Trie: average 429006 of 22 submissions (0.018s)

Best result(Trie): 472477.000000

* in () is the time to run the release data on our local machines

4 The data structure you recommend

There are four main data structures in our recommendation, which are **Trie, Unordered_map, Array of Account's Information,** and **Array of Vector of Transfer History** respectively.

Trie: **Trie**, which is known as a **dictionary tree**, offers a fast method to insert, lookup and delete a string. Hence, we use it to

store IDs (string) and each ID has a unique index. The index is the creating order (the i-th ID to be created) and is stored in the last char node of the string in the **trie**.

Unordered_map: the key is the unique index mentioned above, and the value is the ID. If we use **trie** to get all the existing IDs, it will take **O(number of all nodes)**, which is a terrible complexity when there are too many IDs and each of them is too long. Instead of that, we use this **unordered_map** to go through all existing IDs, which only takes **O(n)** where n is the number of existing IDs. And if we store the IDs in a **vector**, it will take **O(n)** to find it when we perform a delete operation, instead of **O(1)** which we can do by finding the index then delete it from the **unordered_map**. Moreover, since the creating order is unique and **std::hash<int>** returns the original value as its hash value, we claim that there is no collision, that is, we don't need to afford **O(n) complexity in worst case** when inserting, erasing, and accessing an element in this **unordered _map**.

Array of account's information: We use the unique index mentioned above as the index of this array, so the account information of the i-th created account will be stored in array[i].

Array of vector of transfer history: the index of this array is the index mentioned above, so the transfer history of the i-th created account will be stored in array[i]. The transfer history will store the money, the target, the time of this transfer. The

time is stored so that when we merge two accounts the history will be in the correct order.

The account information: the money in this account and its hashed password.

5 The advantages of the recommendation

Fast search ID: Given a string, it takes O(S) to check whether it exists, where S is the length of string. If it does, we can get a non-negative index, otherwise, we get -1. So we can know whether this ID exists or the index of this ID in a short period of time.

O(1) to access information and transfer history of an account: Giving an index, it only takes O(1) to access information and transfer history of a specific account since we implement them with arrays.

If combining the four data structures we recommend, now given an account ID, we first take O(S) to find its corresponding index in trie, and then we only need O(1) to access an account's information and transfer history since the index of this two array is the unique index of this ID. Moreover, we use the unordered_map mentioned in 2. to store all existing ID, and it offers O(1) time complexity to insert, find and delete an ID when index is given. Hence, it takes total O(S) to insert, find, delete and access an account when ID is given.

For other data structures we compared with, we use the ID as the key and the value is its corresponding account in map, like **map<string, Account>** & **unordered_map<string, Account>**. The first one takes O(log n) to insert, find, delete and access the account; the second one takes O(1) in best case but O(N) in worst case to do those operations. Here is a table below.

	Our method	map	unordered map
Time complexity of insert, find, delete and access	**O(S)**	**O($\log n$)**	**Best case: O(1)** **Worst case: O(N)**

*** S is the length of ID string and N is number of accounts**

6 The disadvantages of the recommendation

Extra space using: To implement a trie, we store **an array of pointers** in each node, the **array** is size 62, each one for a character from 0 to 9, and uppercase/lowercase alphabets. So some of the elements in this array will be ***nullptr***, which is a waste of space. Further, once an account is removed or merged to another one, the corresponding index will be set to -1, and there is a waste of space since the original data in those two array (storing the account information and transfer history respectively) won't be removed.

7 How to compile your code and use the system

Type "$ make" in each file of src directory on Github and use it as descriptions in spec.

8 Bonus

Self-implementation of Unordered map: we tried to implement a data structure to cater to the needs of this project,

but unfortunately, we found out it is the same thing as the unordered_map, and even slower. Our approach is to hash an ID into an integer, and use it as an index into our bucket array which is an array of map. We think the problem is that our hash function is causing more collisions than the unordered_map does, so finding ID will become slower.

Trie-implementation: We have talked about the waste of space in trie, it can be solved easily to store **a list of pointers** instead of **an array of pointers** in each node. However, it takes more time to find a specific character since we need to traverse the list whose time complexity is **O(n)**, where **n** is the size of **list.**

Between a waste of time and space, we choose to speed up our program, that is, we implement trie with an array of pointers.

Algorithm of recommend ID during transfer: Our way is to put all IDs into a vector, and select the ten smallest score IDs and print it out, which is a kind of like selection sort. Although time complexity of selection sort is $\mathbf{O(n^2)}$, since we only need the top ten ID, the time complexity become **O(10n)**.

We also tried to use deque, and go through every ID and find the ten best ID then push it into the deque. We always keep the size of deque 10 and check where the input ID should be placed. The complexity of this method is **O(10n)** in worst case too, but it does less 15342 commands(0.003 s slower) than last one. We think the problem is that there are too many “IF-ELSE” statements in this method which may slow down our program.

Compare more data structures: We know that **map in STL** is implemented by **red-black tree**, and how about we use AVL tree to do this? It occurs us to use RB tree, AVL tree, and BST in hw6 to implement our banksystem. Each node in the tree has an account. The algorithms of those three banksystems is the same. What is the result?

BST: 348694 (0.044s)
AVL: 367636 (0.040s)
RB: 371440 (0.036s)
* in () is the time to run the release data on our local machines

BST does the least operations among these three trees, and RB performs much better than AVL does. Since BST is not height balanced, it offers slowest insertion, deletion and searching in worst case of all. Hence, BST performs the worst. However, most of commands should check whether the ID exists, and as far as we are concerned, AVL offers a faster lookup than RB in general. But, the result is that AVL does less operations than RB. We think the problem is that the testdata in judge system won't make RB too unbalanced **(2O(** $\log n$ **))** and AVL does much more rotations in insertion than RB. Hence, the result is that RB does more operations than AVL.

---

wildcmp function in our programs is from the website mentioned in README.md on Github and is modified by 謝致有.