Data Structures and Algorithms
Final Project Report

Team Members

李奕儒 (B00104009)
Ericka Andrea Valladares Bastías (B03902117)
Eui-Jin Jung (T03505201)

I. Work Distribution

A. Analysis and Selection of Data Structures

1. Eric Lee: Vector and Unordered Map
2. Eui-Jin Jung: Map and Unordered Map
3. Ericka Valladares: Array

B. *Coding*

Each made own data structure. Before making, we discuss what to make. Eric focuses on Vector, Eui-Jin focuses on Map, and Ericka takes responsible for Array. However, after a long-term process of discussing, we decide to implement the third data structure as "Unorderd Map" rather than Array. For debugging, we did all together, especially Eric did much.

API：

1. Eric: string compare functions for find , merge, transfer, delete and search.
2. Eui-Jin: recommendation related functions, create, deposit and withdraw.
3. Ericka: score.

C. Report Confection

Each described their own structures and rewrote together.

II. General Introduction for three implementations

A. Data Structure

We divide whole system as two parts. One is for personal information such as id and password. The other is for history. For example, for second data structure we used C++ STL map for personal information and C++ STL Vector for history.

The biggest reason why we divide structure as two data table is for "data integrity". If we add history as a attribute of personal information table, then it is hard to guarantee that one transaction can change both in correct way at once. For instance, transaction "merge" can be a problem. Assume that ID1 has a transaction with ID2 and merge ID1 into other ID. In this case if we change only other ID's history, errors would occur like false history for ID2. We can change both ID's history but it is too effort-consuming. So we made a history table for ensuring data integrity.

Second reason is “redundancy". If we save histories in personal information table, we need to save “From A to B, 1000 NTD” to both ID A and ID B. And also need to save timestamp to check both ID’s history which has been executed at the same time. It is totally waste of memory to save same information twice and to save time stamp.
The last, “reasonableness”. We think it is reasonable to make two different type of information stand separately.

In order to connect two data table, we add “t_index” attribute to each ID, which indicates numbers of history related with corresponding ID. And t_index is C++ STL vector of int which are index of history. We use vector for “History table” and “t_index”, because it is resizable and index show the order of saved time, which means timestamp. So if a certain ID has t_index as {1,2,4}, it means that history number 1,2,4 is about that ID’s transaction. It cost just O(1) time to access. Because we all use vector as history table.

B. Implementation of Recommendation

In this bank account management system, there are two different kinds of recommendation. one is recommendation for “transfer” and another is recommendation for “create”. The former is choosing among all existing IDs, while the latter is generating IDs which does not exist in system. So algorithm of two is thoroughly different.

<Recommendation for transfer>
For this step, we calculate all existing ID’s score and sort it by increasing order using std::sort function. So It takes O(n log n) times regardless of which kind of data structure we use.

<Recommendation for create>
In this step, we should generate new ID. We generate IDs which has a certain score until we finish to generate 10 IDs. So we use while loop score by score.

In order to generate IDs which have certain score, we divide score as two score: score for modification and score for length difference. There is two way to generate ID by using two sorts of score. One is to modify the input ID first and then add some extra characters by score for length difference, the other is to extract substring of input ID by score for length difference and modify the substring.

For those steps, we made some functions:

1. Function for getting permuted string which has certain length (a.k.a. permute function in codes).
2. Function for getting positions to modify by score for modification (a.k.a. get_pos_set function in codes).

3. Function for modifying string using positions (a.k.a change_char_at_pos function in codes). But this function should generate output in every execution, so we make global variable and flag to calculate just once and reuse.

III. Three Structures

A. Using Vector for Personal Information

Vector is one of C++ STL container similar to array. But its size can be changed. It takes O(1) time for search, access, insert, delete function.

We save IDs into vector in sorted form in order to easily find it. When we want to find certain ID, if it is not sorted, we should look up all the element. But if it is sorted, we can use binary search method.

<Implementation for each transaction>

1. Login:
   we need to figure out whether ID exists or not. Using binary search, it takes O(log n) time.
2. Create:
   First, find ID. If ID already exist, use recommendation for create function. Else create it and add to Vector. It takes O(n) times. Because we need to sort it which means moving bigger elements to left, making Vector in ascending order.
3. Delete
   Exactly same as login, except deleting the input ID. But searching time and deleting time is same for Vector, because it also needs to move elements left.
   Time complexity : Average : O(n) , Worst : O(n)
4. Deposit
   Just finding logged-in ID in Vector and change the balance. Logged-in ID's index is already on memory so it takes constant time. However, if lots of ID account had been deleted before deposit, logged-in ID may exceed the range of Vector, so sometimes we need to check it again.
   Time complexity: Average: O(1) , Worst: O(log N)
5. Withdraw
   Exactly same as deposit.
   Time complexity: Average: O(1) , Worst: O(log N)
6. Transfer
   Fine one input ID and logged-in ID. If input ID exists, find 2 IDs and change balance of both. In this case, transfer is a combination of withdraw and deposit which takes O(log n) time for implementation because we need to find index of input ID, and we need to save transaction as history. Saving history takes O(1) time complexity, because add a element to vector take O(1) time
   Time complexity : Average : O(log n) , Worst : O(log n)

7. Merge
   Merging transactions takes O(n log n) time. Because we combine two index Vector together, and sort them. Also it needs kind of transfer and delete function.
8. Search
   During Searching transaction, we should check all the Vector elements by order. If we assume the size of t_index vector as n, it always takes O(n) time.

Pros:

1. Good for memory space, spend less than other structures.
2. Easy to access if we know index.

Cons:

1. If we use sorted vector, it takes more time to insert.
2. Sometimes need to check if logged-in ID exist or not.

B. Using STL map for Personal Information

1. Login:
   Figure out whether input ID exists or not in map. Next to check password. Just like searching in RB Tree.
   Time complexity : Average : O(log n) , Worst : O(log n)
2. Create:
   Check whether input ID exists or not in map. If it exists, then recommend at most 10 IDs using recomendation algorithm. Else, Add ID into personal information table and it takes O(log n) times.
3. Delete:
   Exactly same as login except deleting the input ID. But searching time and deleting time is same for RB Tree.
   Time complexity : Average : O(log n) , Worst : O(log n)
4. Deposit, Withdraw:
   Not that different from Vector implementation.
   Time complexity : Average : O(1) , Worst : O(1)
5. Transfer
   Not that different from Vector implementation.
   Time complexity : Average : O(log n) , Worst : O(log n)
6. Merge
   Same as Vector implementation. So it takes O(n log n) time.
7. Search
   Same as vector implementation. So it takes O(n) time.

Pros:

1. Data are ordered.
2. Intuitive to understand because it is an associative container.

Cons:

1. If system doesn't need ordered structure, it spends meaningless time to sort.
2. Needs more memory than Vector.

C. Using STL unordered_map for Personal Information

1. Login, Delete, Deposit, Withdraw, Transfer:
   In these transactions, implementations are exactly same. The only difference lies in characteristic of STL unordered_map and the fact that these consist of only lookup, access, insert and delete, so time complexity changed. However, if there are some collision, it would be O(n).
   Time complexity : Average : O(1) , Worst : O(n)
2. Create:
   Check whether input ID exists or not in map. If it exists, then recommend at most 10 IDs using recomendation algorithm. Else, Add ID into personal information table and it takes O(1) time.
3. Merge
   Same as Vector implementation. So it takes O(n log n) time.
4. Search
   Same as Vector implementation. So it takes O(n) time.

Pros:

1. So fast to access.
2. Good for system which has many lookup procedure.

Cons:

1. Problems from unsorted structure.
2. Needs additional memory for hash table.
3. If there're too many elements in bucket, time complexity gets similar to O(n).

IV. How to complie codes

1. Type "make run1" to execute data structure 1 using vector.
2. Type "make run2" to execute data structure 2 using map.
3. Type "make run3" to execute data structure 3 using unordered_map.
4. Type "make run_best" to execute recommended data structure.

V. Conclusion

We recommend 3rd structure using STL unordered_map, despite of the fact that all the data structure earn the same points on online judge system (The first one is Unordered Map; the Second & the Third are Map; the rest is Vector). However, in our long-term test, running about 200,000 pieces of data, Map shows the average time of 5sec, so is Vector. Unordered

Map shows the half time(2.5sec).

dsa15_142	2015-07-01 06:47:28	llllll	225450.000000
dsa15_142	2015-07-01 01:59:19	mmmmmmmmm	217992.000000
dsa15_142	2015-07-01 01:57:02	god plz plz plz	223258.000000
dsa15_142	2015-07-01 01:25:21	rarararararara	239228.000000

Of course, system should be fast and unordered_map show the fastest speed. For this Bank Account Management System, look-up ID function is often used. So data structure should display remarkable ability to do it. That's why we choose it.

Even though it has some disadvantages we describe, we think it doesn't that matter. First, memory usage. STL unordered_map needs more memory, but it can be solve with better computing environment. And being frugal of using money for memory could be bigger loss when customers left the system, disappointed with slowness of system.

Second, unsorted account information. In reality, there's no need to sort all the ID in lexical order. This system is just for money transfer, not for a telephone number directory. So it also doesn't matter.

Above are the reasons why data structure using STL unordered_map is the best of us.