DSA Final

1. Our team

隊名：明星花露水(dsa15_final16)
組員：
林庭宇(B03902061)
董文捷(B03902062)
楊松道(B03902064)

2. Resbonsibilities

林庭宇：git branch maintenance、bank template design
董文捷：DS main coder
楊松道：debug script 、 test
Together：discuss the frame、coding、stay up、see sunrise、breakfast

3. data structures and results:

(1) Before we start to code:

a. transfer list

The biggest problem we encounter is how to update the transform history. After merge operation, not only should the history of the two accounts be updated, but also many accounts having records with the merged account should update their history, too. Browsing all account history to check if they are associated with the merged account and update them one by one may be time-consuming. The first solution come to our mind is disjoint set, which is mentioned in hw6. However, it may be troublesome to handle the situation if you delete an account and create an account with same name. We come up with a great idea in one weekend discussion, if you save all transform history in a big vector and save the pointer to the history in the two associated account, you can update efficiently after merge by change the history pointed by the pointer in the merged account. Any history associated with the merged account will be updated automatically since they share the same pointer.

b. Find operation

I consult the concept on http://www.cs.princeton.edu/courses/archive/spr09/cos333/beautiful.html“A Regular Expression Matcher” and write a function to verify if an existing ID matches a wildcard ID in a recursion way. Although C++ also offer <regex>, we actually don’t need to check some special characters. There are only ? and * and can be handled in a concise recursion function.

c. ID recommendation

For recommending existing IDs and non-existing IDs, we use different algorithms based on their different properties

(a) Recommend non-existing IDs

According to the rules, least scores are first. When it comes to exactly the same score, smaller ranking by the dictionary order are better. We solve this problem by setting current target score from 1 to 10 and arrange them by dictionary order with the help of recursion. Recursion starts from the 0th character of ID. Program determines what to do among three the options: stop adding character, add different character, or add the same character, and then runs recursion on the next character. According to the definition of dictionary order, when the string in front of index i is identical, stop adding character < add a different smaller character in dictionary < add the same character < add a bigger character in dictionary. Therefore, by keeping proper execution order and testing existence of accounts, we can easily reach the goal.

(b) Recommend existing IDs

Because scores can be very big when finding existing ID, listing all legal IDs by score from 0 to 10 will be very slow. Therefore, instead, we list all exting IDs and find 10 IDs with least scores and dictionary order. When it comes to special data structure like trie, maybe some method can be adopted to avoid this kind of iterating and accelerate the program.

There are three significant part about this bank system. First and foremost, it's the main data structure we use. Furthermore, how to deal with transfer list is important, as mentioned above. Last but not least, some special function like find and recommend ID is also crucial. The second and third part is completed in the early stage, it is roughly fixed and can only be faster if there are some special data structure, so we focus on the main data structure in the late stage. The key points of the data structure are how to search and delete string(ID) in a fast way, and time needed to traverse may have a huge influence, too. According to those requirements, we come up with some data structures.

(2) Data structure

a. Map

An intuitive data structure is map since map is implemented in red-black tree and thus support a fast search. At first, we use string for the sake of convenience. However, to make program faster, we use c-style character array and cstdio instead.

b. Trie

Trie is a dictionary tree whose node stores character. Depth i's node store the character of index i in a string. This kind of data strutructure is seemly advantageous because depth is limited to 100. However, the result didn't turn out as we expected. We have come up with the folloing possible reasons:

(a) Every node stores 62 pointers that points to nodes and therefore consumes a huge amount of memory. It might take much time adding / deleting nodes.

(b) When ID length is long, there will be useless empty nodes and slows down the traversal. On the other hand, a rb tree is unlikely to reach the depth of 100

because it needs about 100^2 nodes. To sum up, trie is good when IDs' length are short and IDs' quantities are many.

(c) Some parts of program aren't optimized for trie yet.

c. Ternary Tree

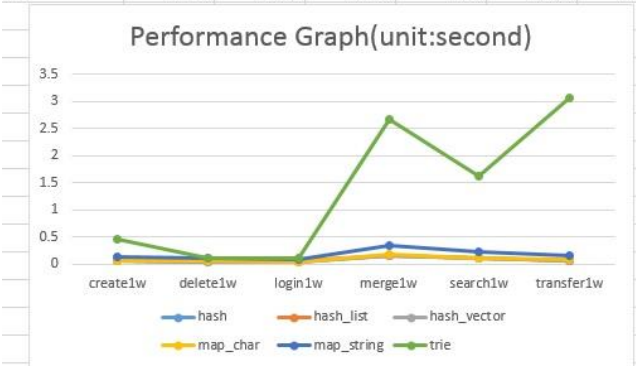
We also find a string-related data structure Ternary-tree in http://www.drdobbs.com/database/ternary-search-trees/184410528 , which is said to combine the time efficiency of tries with the space efficiency of binary search trees, we make some effort to implement Ternary_tree by replacing array with map to store children in the node of a trie.

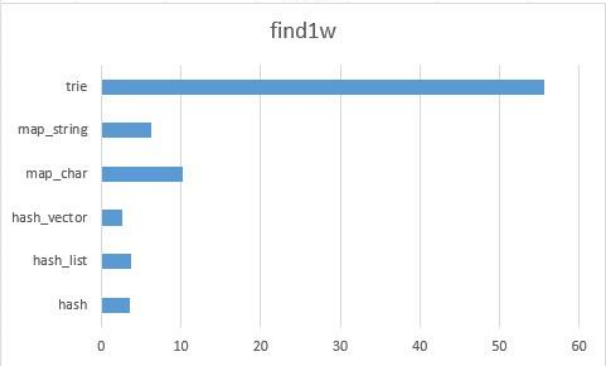
d. Hash

After some discussion, we find that whether the data structure is ordered doesn't matter a lot, since only find operation need an extra sort. If you want to access some data fast, hash table may be an ideal choice. We implement hash table by unordered_map in C++, because we use const char * as key, we also have to write a hash function on our own. We compare some hash function and finally decide to choose MurmurHash3. According to cplusplus.com "unordered_map containers are faster than map containers to access individual elements by their *key*, although they are generally less efficient for range iteration through a subset of their elements.", to improve the speed, we try to add another data structure(vector or list) to store accounts' pointers for faster traverse

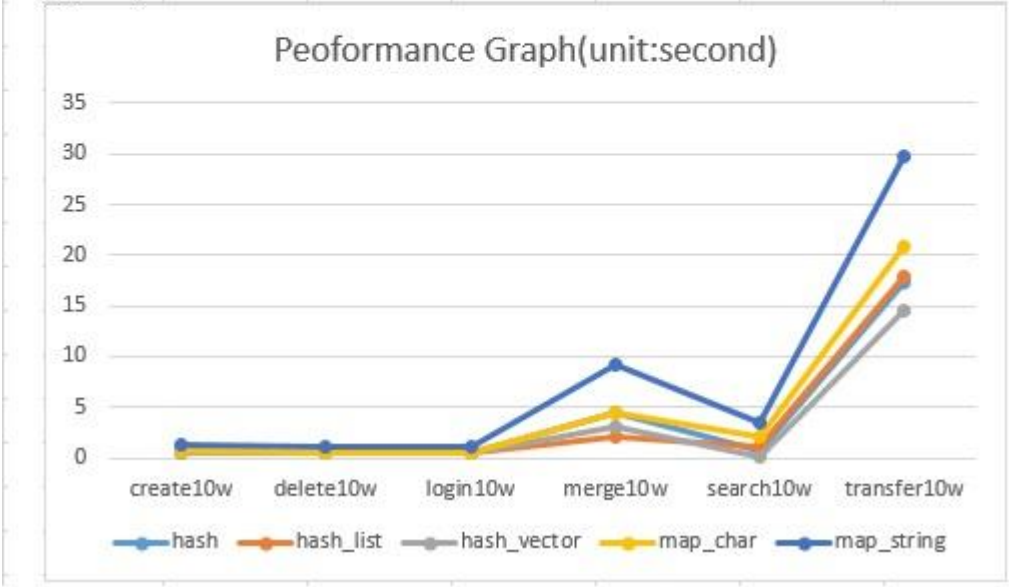
(3) Local Test

ds\cmd	create1w	delete1w	login1w	merge1w	search1w	transfer1w	find1w
hash	0.059	0.047	0.045	0.162	0.095	0.062	3.591
hash_list	0.059	0.047	0.053	0.16	0.098	0.059	3.755
hash_vector	0.057	0.046	0.047	0.164	0.097	0.06	2.654
map_char	0.061	0.051	0.047	0.178	0.105	0.085	10.214
map_string	0.125	0.098	0.09	0.348	0.23	0.157	6.345
trie	0.466	0.117	0.095	2.66	1.615	3.076	55.65



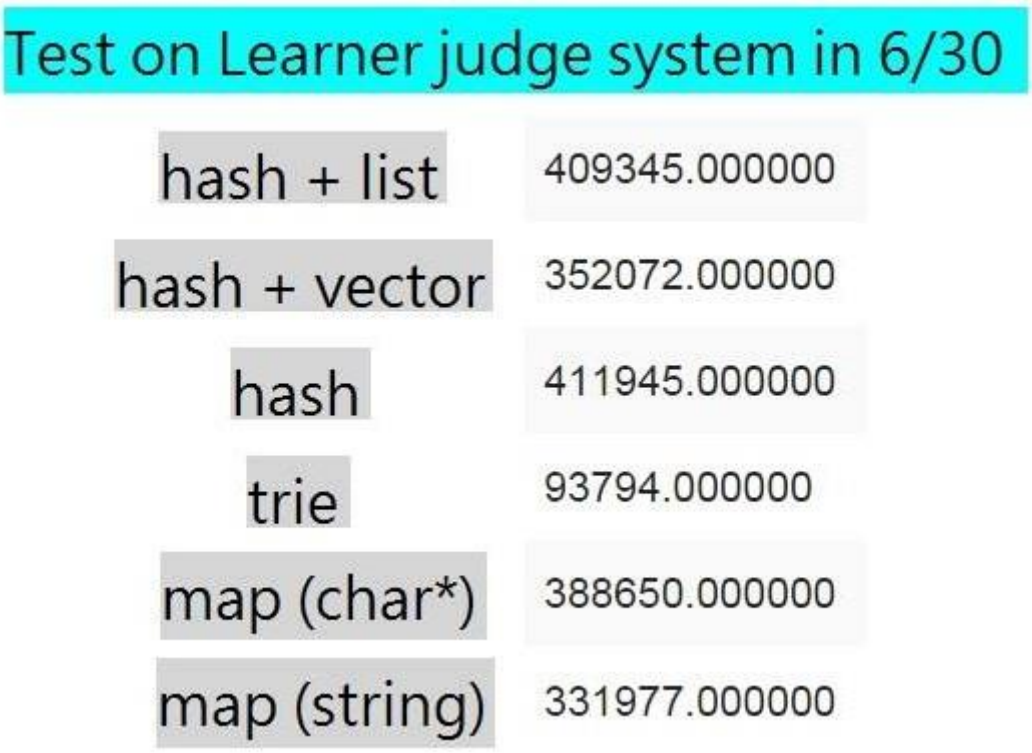


ds\cmd	create10w	delete10w	login10w	merge10w	search10w	transfer10w
hash	0.607	0.49	0.477	4.384	0.649	17.28
hash_list	0.571	0.516	0.503	2.052	1.11	17.927
hash_vector	0.564	0.497	0.498	3.126	0.171	14.523
map_char	0.671	0.541	0.512	4.464	2.074	20.938
map_string	1.296	1.108	1.041	9.104	3.445	29.738



This local test shows relative speed of different date structures.
They are tested on linux20, whose load is little.
Data structure trie and find10w are both slow thus excluded from graphs.

online test:
Highest result in judge system:
map_string 333547
map_char 496055
trie 95970
hash 455023
hash_vector 363620
hash_list 420575

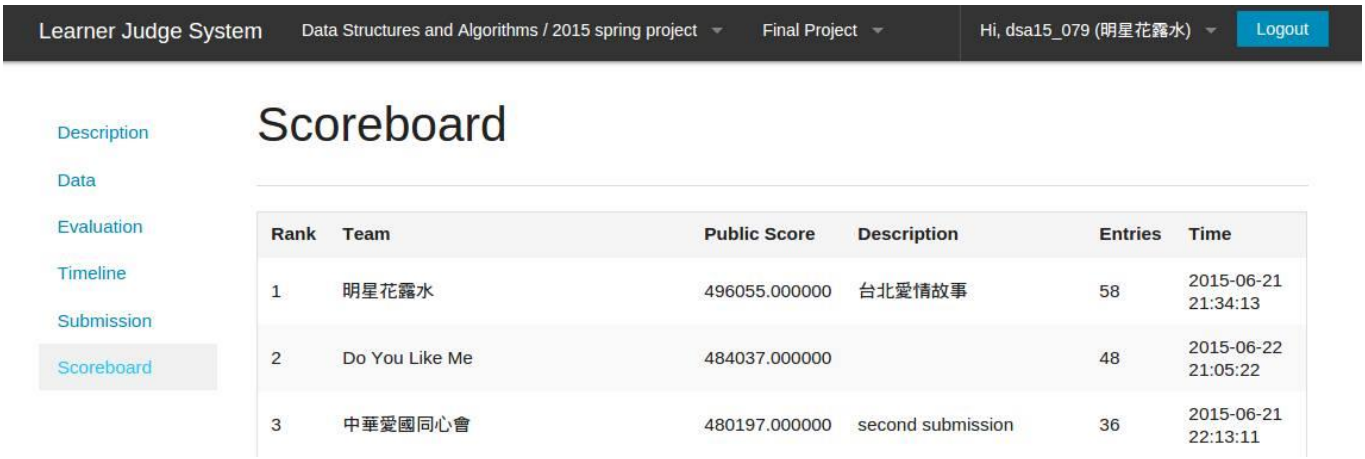


4. Recommended data structure - Hash

(1) Pros: Fast finding IDs. With a good hash function, collision can also be avoided.
(2) Cons: Traversal will take more time, so an additional list that stores all existing accounts is used to make up for this draw-back.

5.

How to compile: $make
How to use: As indicated in spec.



(We were Rank 1 before rejudging on 7/1)

6. Review:

Final project is never an easy task, we made every effort and endeavored to better our program. Although the proccess was hard and we stayed up until seeing the sunrise many times, the experience acquired and the sense of accomplishment when seeing score progress are worthwhile. I believe those interesting stories with my teammates will become precious moments in my life.