Final Project Report

隊伍: 舒潔三層抽取式衛生紙90抽

組員: B03902001 駱定暄, B03902039 施秉志, B03902107 鄭格承

Data Structure

1. Index: **STL set**, Transaction History: **STL vector**

   The reasons we decided to use **STL set** for Index are:
   (1) I had used STL set in my hw2 & hw6 before I started, so it is relatively familiar.

   (2) The elements in **STL set** are all unique, which is highly according to the account system. Besides, it will sort the elements automatically, which brings the efficiency when it goes to output something such as results of **find** or recommended IDs.

   The reason we decided to use **STL vector** for Transaction History is:
   The sequence of **STL vector** by pushing_back is what we need.

2. Index: **STL Map**, Transaction History: **STL vector**

   The reasons we decided to use **STL set** for Index are:
   (1) We think that **STL set** and **STL map** are two similar structure(rb-tree), and thus we want to compare this two.

   (2) Although **STL set** and **STL map** are two similar structure(rb-tree), we want to check out whether ID be the key will make the program faster.

3. Index: **STL Unordered_Map**, Transaction History: **STL vector**

   The reason we decided to use **STL Unordered_Map** for Index is:
   We found that sorted structure didn't highly fit our demands, so we tried the Unordered_map in the STL to test and compare the time each of the data structures will cost.

4. Index: **vector vector**, Transaction History: **STL vector**

   The reason we decided to use **vector vector** for Index is:
   On a brisk night, Ge-Cheng suddenly came up with an idea, and then he check out the first character. He found that if the number of IDs is big enough, the first character will have an equal division performance on each character in out character domain{0~9, A~Z, a~z}. So we designed a vector vector to hash the IDs by the first character.

5. Index: **Trie**, Transaction History: **STL vector**

   We have implemented a trie containing 128 sub-trie pointers and the customer's data pointer.If we don't run the list-10-exist-id function, which require traversing the whole trie,trie has the max speed in create, delete, merge,and so on. However,even though we better the trie by store an array that told the trie which child contain valid data,it still runs slowly.

6. Index: **RB**, **AVL**, **BST**, Transaction History: **STL vector**

   In addition, we implement three kinds of tree structures which we used in HW6. After uploading the programs to the judge system, we concluded that three data structures have the similar performance with the score about 260 thousand and the BST performs the worst. The result points out that the find operation is more important than insert operation, because the BST can insert an element faster but cannot ensure the balance.

As for Account Information, we designed a class : **Customer**, to contain it. The private part inside are **password**, **money** and **history**; and the public part are **ID** and some function that have to access private part, such like **authentication**, **deposit**, **withdraw**, **merge**, **search** and so on.

Algorithm (implement)

First, we outline a concept here about how we divided the project into several parts. The **main** function is designed only to read in the command**,** and then distribute the work to some function called **processXXX**.

Here are the functions(which according to the commands), and its **time complexity.**

* N: number of accounts(IDs), H: number of History, s: length of string(1~100)

- login: calls set.find(), O(logN).
- create: calls set.find(), O(logN); conditionally calls **listing function**.
- delete: calls set.find(), O(logN).
- merge: calls set.find() twice, O(2logN);
  merging two history, because two vectors are sorted, so the time complexity is $O(H_1+H_2)$, instead of O(NlogN).
- deposit: nothing special, O(1).
- withdraw: checking money, O(1).
- transfer: calls set.find(), O(logN); conditionally calls **listing function**.
- find: processes the finding string(ex. *x?0E), O(s);
  check all strings(IDs), O(N * s).
- search: go through the History, O(H).
- **listing function**:

  1.transfer: find IDs which exist: O(N).

  2.create: find IDs which don't exist: very tricky(maybe O(exp(N))…

We classified the case by modifying length or not:

(1) shorten length (2) same length (3) extend length

and we use some recursion function to determine how scores are distributed.

```
void extend(int cnt, int n_var, set<string> &listset2, string ID, int pos){
    if(cnt == 0){
        vector<int> needChange;
        pick_or_not(n_var, 1, needChange, listset2, ID, pos);
        return;
    }
    for(int i = 0; i < 62; i++){
        string ex_string = ID + strtable[i];
        extend(cnt - 1, n_var, listset2, ex_string, pos);
    }
}
```

extend is how we determine the length that the string will extend.

```
void pick_or_not(int n, int h, vector<int> NC, set<string> &listset2, string sample, int pos){
    if(n < h && n != 0){

        return;
    }else if(n == 0){
            //cout << "yeah!!!" << endl;
        all_digit_change(1, NC, listset2, sample, pos, sample);
        return;
    }
    pick_or_not(n, h+1, NC, listset2, sample, pos); // not pick
    NC.push_back(h);
    pick_or_not(n-h, h+1, NC, listset2, sample, pos); // pick
    //cout << "fuck yeah!!!" << endl;
}
```

pick_or_not is how we determine the position that the character will change.

```
void all_digit_change(int h, vector<int> &NC, set<string> &listset2, string &sample, int pos, const string origin){
    if(h > (int)NC.size()){
        listset2.insert(sample);
        return;
    }
    for(int i = 0; i < 62; i++){
        if(strtable[i] == origin[pos-NC[h-1]])
            continue;
        sample[pos-NC[h-1]] = strtable[i];
        all_digit_change(h+1, NC, listset2, sample, pos, origin);
    }
}
```

all_digit_change is how we determine the changing part of **pick_or_not**.

Test

1.**find**:

We first use **STL vector** and **STL set** to test the time:

(1) 30,000 IDs vs 100/1000 commands

vector: 0.678s/4.998s set: 0.794s/6.335s

(2) 30,000/300,000 IDs vs 10,000/1,000 commands

vector: 48.816s/47.793s set: 63.390s/65.278s

We observed some phenomena:

1. From(1), we found that 100-cmds time cost is more that 1/10 of 1000-cmds. This is due to the insertion of the data.
2. From(2), we found that in same level of (IDs * cmds), vector is more faster in more IDs; however, set is more faster in processing more cmds.

Conclusion: Vector is more faster because it doesn't cost much time inserting. On the contrary, set cost more time on inserting due to its maintenance of sorted.

2. **whole program**:

STL set vs **STL map** vs **STL unordered_map**

testdata : create 5000, deposit 5000, transfer 2000, search 2000, merge 2000, create 1000, deposit 4000, transfer 1000, search 1000

set: 0.604s / map: 0.595s / unordered_map: 0.582s

Conclusion: At the begin of the test, we assumed that unordered_map will be the fastest because the time complexity should be O(1) by good hash. However, the output was that they were so close to each other. After observing this, we found that the random factor plays an important in this test, and lead to the fact that there's no obvious difference between these.

3. **whole program**:

vector vector vs **STL unordered_map**

testdata : release data(10,000 data)

vector vector: 0.062s / unordered_map: 0.097s

Conclusion: Unlike the test we experimented before, we tried to compare the two data structure that is in very different implementing way. Although we found that the vector vector is like a once-hash table and we supposed that its speed will be much faster than other data structure; however, the score we got on the Learner Judge System was just closed to Unordered map due to the abundance of the data numbers.

Recommendation(data structure)

vector vector: open 128 vector, push back the customer class into the vector[i] with i = ID[0]. It is partial sort,to some extend, but doesn't take much time in maintain the structure(i.e. ID[0] as hash value). If the size of every vector[i] are close, and the input data are not too big, it wouldn't take much time in finding in finding a certain data.

How to compile your code and use the system

Compiler: g++. We use makefile to control different version of program.

Why we think we deserve the bonus

We didn't implemented some feature because we spent our time optimizing our program.

First, We implemented the data structure by **STL vector**, **STL set**, **STL map**, **STL unordered_map**, **vector in array**, **set in array**, **vector in vector**, **trie**.

On the other hand, we also change and try to compare the algorithm we use to fit different demands of this project. For instance, we change the algorithm in **recommend IDs** of **transfer** from sorting it by **STL set**'s origin insertion maintenance to some way like selection sort because we find that we only need to find out the smallest 10(in score and then lexically).

Work Distribution(very discrete but workload balance)

駱定暄：DS::Index(RB, AVL, BST), **HISTORY**(vector), **transfer**, **merge**, **search**, optimizer

施秉志：DS::Index(set, map, unordered_map), **login**, **find**, **recommend IDs**, REPORT, debug guy

鄭格承：DS::Customer, DS::Index(vector in array, set in array, trie), **create**, **delete**, **deposit**, **withdraw**, debug guy

Cite

http://www.zedwood. com/article/cpp-md5-function

http://adtinfo.org/ (hw6)