DSA Final Project Report

1. Members and Job Responsibility

B03902082 江懿友：coding

B02705019 陳姿穎：report

B02705021 謝志邦：coding

2. Packages and Libraries Used in Our Project

We have used an MD5 library provided by Frank Thilo at http://www.zedwood.com/article/cpp-md5-function. The library allows us store an user's password not in plain text but in hashed string to promote the security of our system.

MD5 is a kind of hashing algorithms which hashes a string of any length into a string of a fixed-sized of 128 bits. The algorithm works as the following flow. First, the input string will be padded so that it will become divisible by 512 bits. Then, we divided the string into four chunks, each of them is 128 bits. The hashing of a string contains 4 rounds, each of them performs 16 similar operations on a chunk. We then rotate the order of the chunks after 64 operations. We joint all the chunks and get the result hashed string in the end.

3. Data Structures and Algorithms We Used and Comparisons

Our system mainly maintains a class called AccountSystem, which contains some data structures recording accounts and the transferring histories. In this section, we are going to describe the data structures and algorithms we used in our project and make comparison between some of them.

(1) Main Structures

In this section, we are going to describe our data structures in detail, such as what kind of data we want to maintain in AccountSystem, and how we record the data of an account and its transferring histories. We used three data structures to record all these information called Trie, which is used to save all created IDs, Account, which is used to store all information of an account, and TransferReport, which is a subclass in Account, to record all the transferring histories of an account. The structure of AccountSystem is illustrated in the figure below[Figure 1].

In AccountSystem, we have some variable storing some data needed by the system. We used an integer named timeStamp to record the time when transferring money, and an integer named unusedHashID to record the smallest unused integer, which will be map to a string when an new ID is created. We also use a vector of integer to record the parent of each node, this is used in the

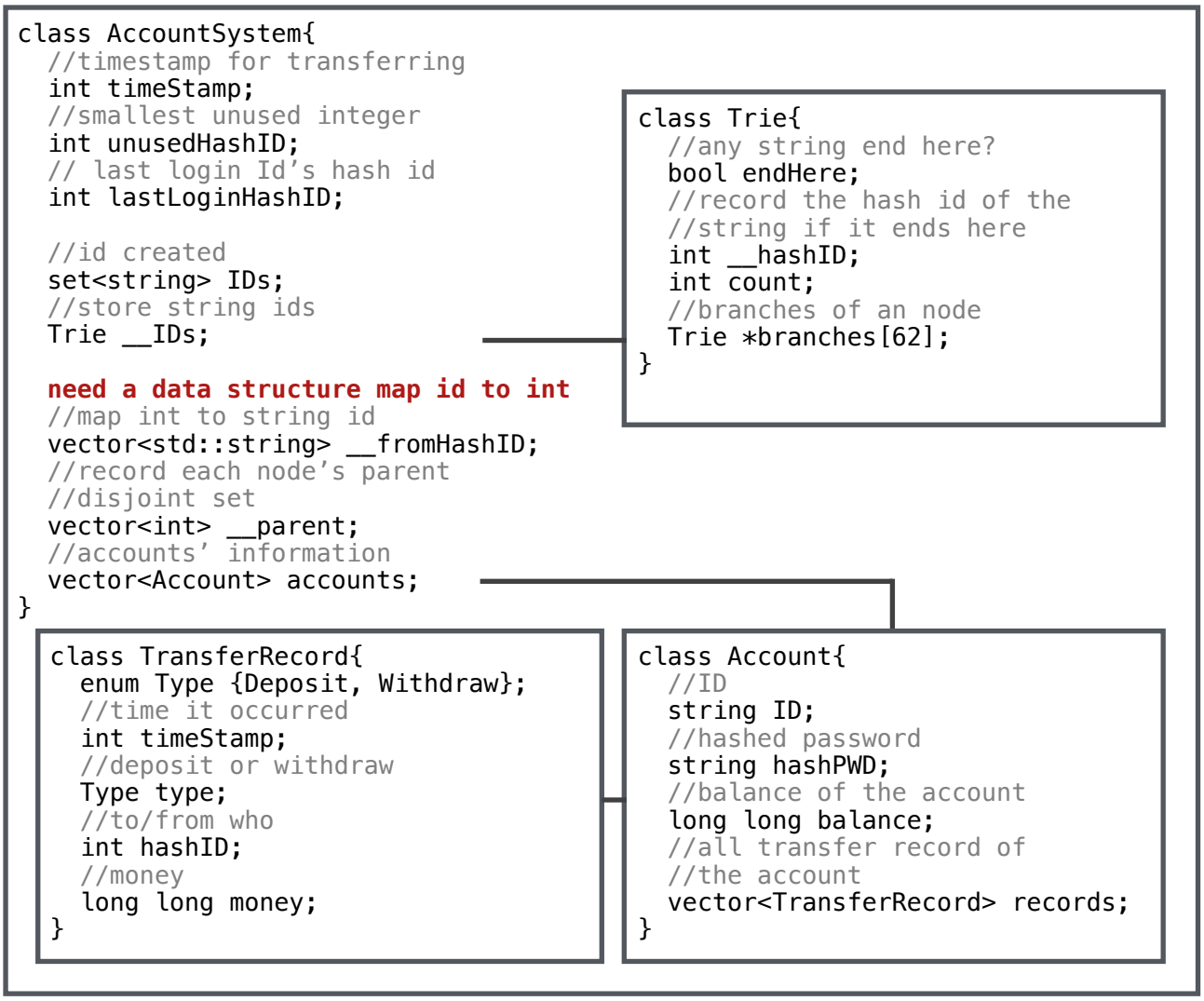

Figure 1 The structure of our system

"merge" function. We used the concept of disjoint set when merging two accounts, that is, we modify the parent of the account in the vector but not in the transferring histories. Thus, we only need to modify an integer when merging, and find out the root of an account when we want to print out the transferring histories.

Another point of our system is Trie, which is a kind of tree mostly used to store strings. In our system, we used Trie to store our accounts' IDs. Since our IDs contains upper and lowercase alphabets and digits, a node in the Trie has 62 branches in total. Trie in our system is mainly used for ID recommendation at first, but we have stored other information on Trie later, which we will discuss in detail in the comparison part of this report.

In the next section, we are going to compare between some data structures and algorithms we implemented on storing hashed IDs and string matching.

(2) Different Data Structures and Algorithms and Their Comparison

In the project, we have made two comparisons. One is comparing the data structures for storing hashed IDs, and the other is comparing the algorithms for string matching, which is an

important topic in the “find” function. In this section, we are going to describe how we implement these data structures and algorithms, justify their time and space complexity and show the results of our comparison.

1. Data Structures for Storing Hashed IDs

In our system, we hash a string ID to a smallest unused integer, which help us record some information such as “last login” or “a parent of an ID” in an easier way, and the integer will plus 1 after a new account is created. Thus, it is important for us to find a data structures that allow us to search and insert a hashed ID with a given string ID. We compared three kinds of data structures: map, unordered_map and Trie.

The first method we implemented is STL map provided by C++ is implemented with red-black tree. Both of insertion and searching take time complexity O(log n), where n implies the number of the number of the entries storing in the map.

The second method we implemented is STL unordered_map, which is also provided by C++. Unordered_map is a kind of hash map, thus, the time complexity of searching and inserting are both O(1). The result of comparing between map and unordered_map is shown in the figure below[Figure 2].

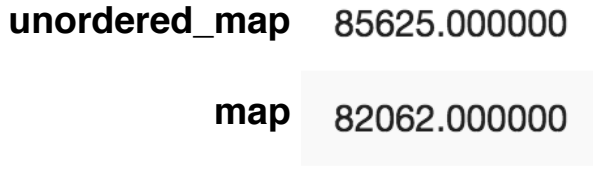

Figure 2 A comparison between map and unordered_map. The figure shows the entries we passed in the judge system

The third method we implemented is Trie. We already discussed how Trie stores string IDs in the last section, and we combined storing hashed IDs with Trie by storing the value of hashed ID in the end node of a string. The complexity of searching and inserting in Trie is O(n), where n implies the length of the string we want to search or insert. Because we need to traverse all the characters of the string in the Trie. Below we provide a comparison between Trie and unordered_map[Figure 3], and a simple illustration about how searching a hashed ID with a given string ID on a Trie[Figure 4].

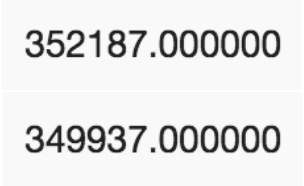

Figure 3 A comparison between unordered_map and Trie. The figure shows the entries we passed in the judge system

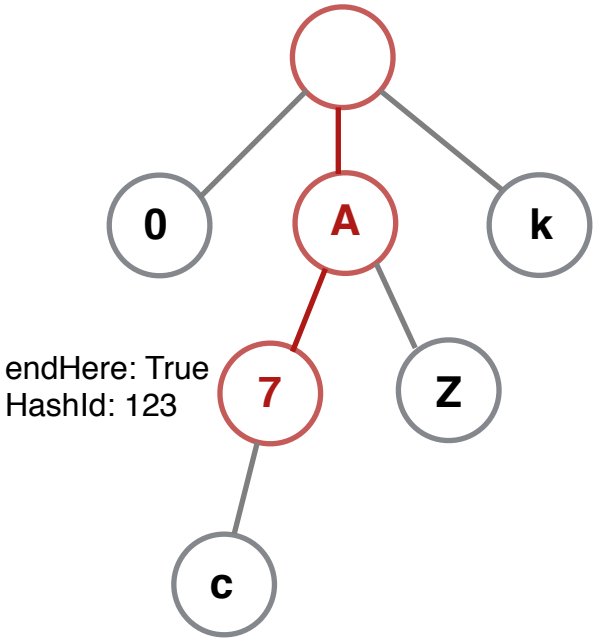

Figure 4 get the hashed Id of “A7”

2. Algorithms for String Matching

String matching is used in “find” function, and we compared three kinds of algorithms: regex library provided by C++, Trie DFS searching and KMP string matching algorithm.

The first method we used is regex, the C++ library which supports regular expression. It just fit our requirement of finding all possible IDs with a given string. However, it is so slow that we discarded this method in the end. It seems like regex can deal with complicated case efficiently. In simple cases, however, the overhead of its algorithm may be time consuming, such as the entries in the competition.

The second algorithm we implemented is Trie DFS algorithm, which apply depth-first-searching algorithm on the Trie to search all the possible strings with the given wildcard ID with time complexity of O(n), where n implies the length of the given string.

The third algorithm we implemented is KMP string matching algorithm, and the complexity is O(N + M), where N and M imply the length of given string and the string we are matching.

The result of comparing the three algorithms is show in the figure below[Figure 5].

Trie DFS	335051.000000
KMP	349937.000000
regex	81260.000000

Figure 5 A comparison between regex, KMP and Trie DFS. The figure shows the entries we passed in the judge system

(3) Recommendation

After comparing the data structures and the algorithms mentioned in the last paragraph, we decided to implement Trie for storing ID strings and their hashed ID, and KMP algorithm for string matching in “find” function.

Since we already implement ID storing with Trie on the “recommend” function, the advantage of storing hashed IDs on Trie promote the speed of execution. Because the hash function of unordered_map can be slow and unordered_map is a little bit fat. However, Trie itself is a waste of memory since every node on the Trie has 62 branches, indicating every upper and lowercase alphabets and digits, even if some branches are never used.

On the other hand, KMP algorithm for string matching performs well when observing our submission records. Its advantage is its low time complexity, which is O(N + M), where N and M imply lengths of the two strings we are matching. We couldn’t find the disadvantage of KMP algorithms in this project since its both time and memory saving.

The best performance of our system has passed 360959 entries, which is shown in the following figure[Figure 6].

Figure 6 The highest recored of our system, implementing Trie for hash ID storing and KMP algorithm for string matching

4. Usage of Our Makefile and System

Use the following command to compile our code and generate an execution file:

```
$ make
```

and use the following command to remove files generated by the Makefile:

```
$ make clean
```

The usage of our system, that is, commands, arguments and the outputs of a command are all following the format given by the spec.

5. Bouns

In this project, despite of implementing different data structures, we also compared some algorithms, such as Trie DFS and KMP algorithms. We study on these algorithms and implemented them on our system to make the performance of the system become better.