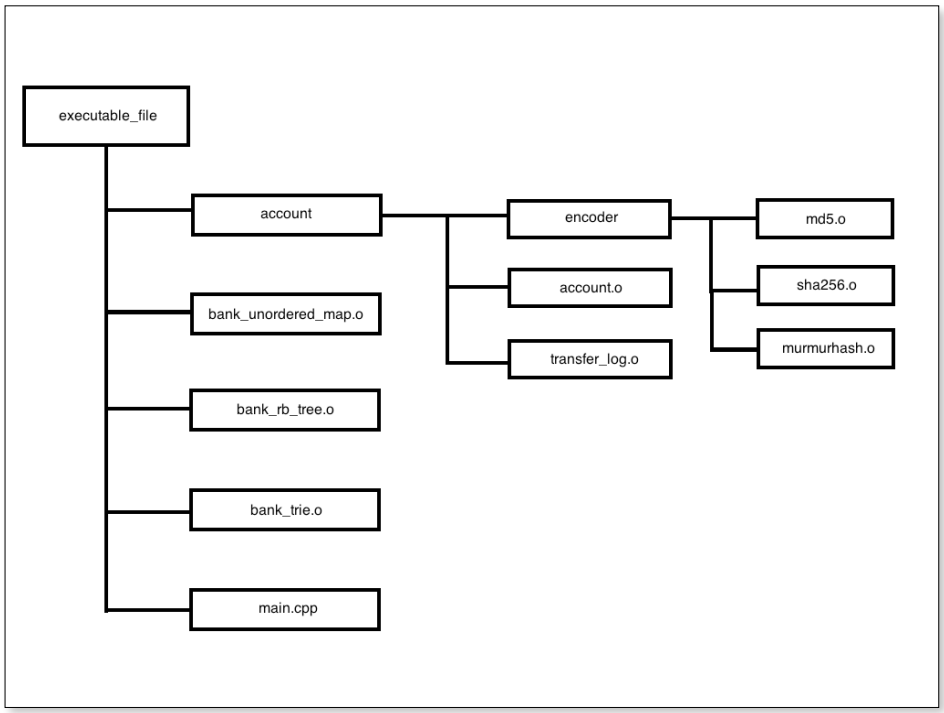
DSA 2015 spring final project report

dsa15_final15

Framework of Files



Framework of Programs - Modular Programming

Our project is basically seperated into four layer

1. First layer - Main program
   (1) process basic input and call bank system
2. Second layer - Bank system
   (1) create and delete accounts
   (2) process account login
   (3) traverse and find account
   (4) recommend exist or not exist accounts
3. Third layer - Account & Transfer_log
   - Account
     (1) verify password with encryption tools
     (2) deposit and withdraw money
     (3) merge the other account
     (4) transfer to the other account and keep logs
     (5) search transfer logs
   - Transfer_log
     (1) keeps all transfer logs
     (2) add new transfer log
4. Fourth layer - Encryption tools

Our Team

- Team Members: B03902048 林義聖、B03902105 劉岳承、B03902020 李睿軒
- Responsibitilies of Members:

	劉岳承	李睿軒	林義聖	各種資源
README.md	v			
Makefile			v	
main.cpp		v		
bank_unordered_map.o			v	
bank_unordered_map_murmur.o			v	
bank_rb_tree.o		v		
bank_trie.o	v			
recommend.h	v			
regex_translate.h		v		
account.o			v	
transfer_log.o			v	
md5.o				v
sha256.o				v
trie.o	v			
rb.o				v
murmurhash.o				v

Data Structure - Hash Map

Hash map can use hashed key to quickly create, delete and find the stored data.

- Original STL unordered map
- STL unordered map with MurmurHash

It accomplished 318962.000000 lines input on the online judge system.

Data Structure - Radix Tree

Radix tree, based on trie, is one of complex trie that can compress useless tree nodes to reduce its tree height. In this way, we consider radix tree will speed up searching in our bank system. It accomplished 278729.000000 lines input on the online judge system.

Data Structure - Red Black Tree

Our red black tree is from GNU libavl, which we got from HW6. We consider it may have better insertion and deletion speed than AVL tree. It accomplished 300043.000000 lines

input on the online judge system.

Comparison

For different operation, we made different datasets to test for comparison. (Environment: Mac OS X + Intel i5 + 8G RAM)

- Normal (10000 lines from judge system)

	Unordered_map	RB_tree	Radix_tree
user time	0.20s	0.21s	0.23s
system time	0.02s	0.02s	0.02s
total time	0.232s	0.24s	0.254s

- Find (create * 5000 + find * 10000)

	Unordered_map + WildCard compare	RB_tree + WildCard compare	Radix_tree + own traversal for matches
user time	4.6s	8.07s	13.96s
system time	0.07s	0.09s	0.11s
total time	4.701s	8.208s	14.125s

- Create (create * 100000)

	Unordered_map	RB_tree	Radix_tree	Unordered_map + MurmurHash
user time	3.220s	3.258s	3.036s	3.161s
system time	0.144s	0.138s	0.152s	0.136s
total time	3.369s	3.400s	3.198s	3.301s

- Login (login * 100000 do after creation)

	Unordered_map	RB_tree	Radix_tree	Unordered_map + MurmurHash
user time	2.825s	3.112s	2.930s	2.975s
total time	2.924s	3.230s	3.038s	3.096s

- Delete (delete * 100000 do after creation)

	Unordered_map	RB_tree	Radix_tree	Unordered_map + MurmurHash
user time	2.873s	3.175s	2.973s	2.917s
total time	2.969s	3.369s	3.074s	3.022s

Recommendation

Based on our comparison of these data structures, we will recommend the hash table as the best bank system.

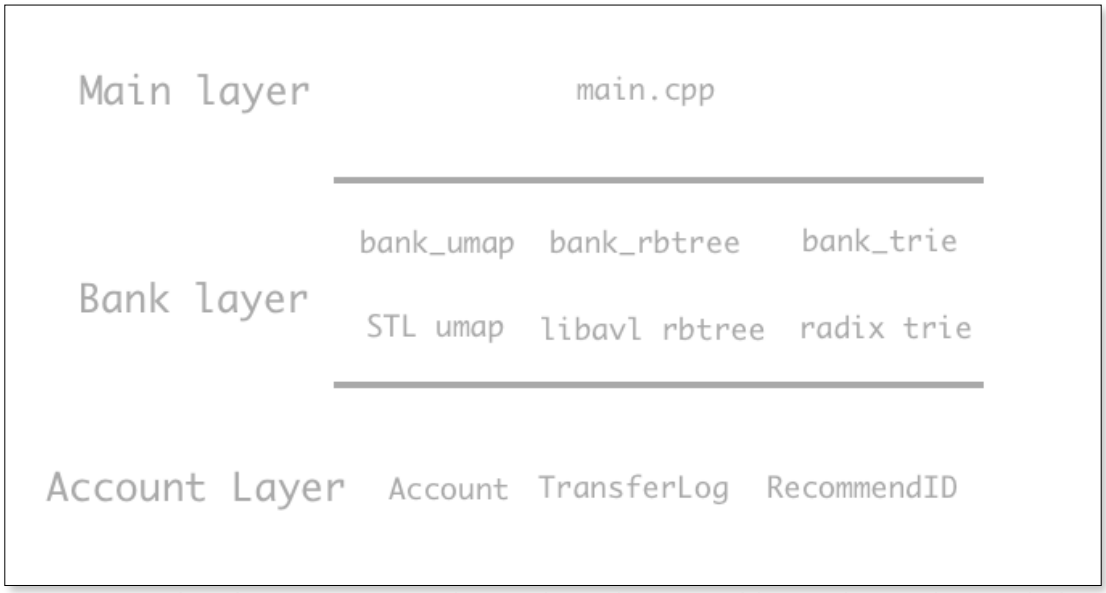
- Advantages
  1. average creation and deletion time is not bad.
  2. can quickly find the specific data. (judge by login() speed)
  3. its traversal speed is faster. (judge by find() speed)
- Disadvantages
  1. According to its key type, the hasher function is very important. (we choose murmurhash_x86_32bit, and we guess it cause more collisions than default hasher)
  2. As the creation or deletion of the data increase, the STL unordered_map will rehash the key. It consumes a lot of time.

How To Use Our System?

Because we used modular programming to separated our project into many layers and parts, we can use the same basic objects and main program with various definition to handle different data structure of bank systems. In our src/ folder, type “make bank1” to compile bank system implemented with STL unordered_map, type “make bank2” for bank_rb_tree, “make bank3” for bank_radix_tree, and “make bank4” for bank_unordered_map_murmurhash. All of their output executable file is named “final_project”, and you can type “./final_project” to execute it.

Bonus Features

- Well Modularized



Our system has been separated into three layer and lower layers has only been used by their parent layer. For the three bank systems we implemented, it’s actually in the Bank layer.

One of the best advantage is that if we want to implement a new bank system by another data structure. We only need to implement each method that the bank system

should offer. Without changing any other files, it can be easily and neatly accomplished. It also highly increase the readability of codes which is very important in cooperating.

Security

Because only the Bank Layer can touch the Account class, which means that the main program has no way to directly manipulate with any account. It can only call the methods that the bank system offer, such as verifying passwords or creating a new account. Besides, our Account class only store s the cypher text for each account which is also a private member of Account class. We believed that our system has a high security feature.

Double-Checking

The encoding function we used is MD5. But we also considered that it still may have collision even though the possibility is very low. Thus, we store another cypher password made by SHA256. If MD5 has unfortunately collided, double checking the SHA256 cypher will find out the differences between wrong password and the right one.

Easily Compiling

Our program can be easily compiled by adding a define parameter for the compiler. The define parameter will decide which bank system should be included for the main program.

Parameter for Compiler	the bank system included by main.cpp
-DBANK_UM	bank_unordered_map.h
-DBANK_RB	bank_rb_tree.h
-DBANK_TR	bank_trie.h

Radix Tree

Radix Tree (trie.h) is written by our own. It's complicated when we try to compressed the nodes of trie and also put accounts on nodes. Different from the normal trie, it maintain height of the trie. And also provide a traversing way to find wildcard string. But it didn't work well in our test. Comparing with the other way we wrote for finding wildcard string, simply take all IDs in the bank, and use the wildcard compare algorithm seems to be faster.

Also, from the online judge we found that the bank system made by radix tree is slower than the hashing map. We guess that the additional maintenances cost a lot more constant time than the hashing function for the hashing map when the string length is not that long.

Reference

- wildcard string compare function(we used it to compare wildcard input and exist account ID) - http://www.codeproject.com/Articles/1088/Wildcard-string-compare-globbing
- murmurhash.c(it was used in our bank_unordered_map_murmur.cpp as different hasher function from default one STL unordered_map used) - https://github.com/jwerle/murmurhash.c
- rb.c(we used rb.h and rb.c to implement our bank_rb_tree.cpp) - http://adtinfo.org
- Radix Tree(we studied it but wrote our own radix tree) - http://kukuruku.co/hub/algorithms/radix-trees