壹、隊名及組員

隊名: 就是會對~

組員: B03902078 林書瑾, B03902080 黃子賢, B03902024 鄭筱樺

貳、工作分配:

林書瑾: clientMap, recommend non-existing ID, trie, bonus, report

黃子賢: ID_table, recommend existing ID, main, bonus, report

鄭筱樺: clientMap, historyVec, map_version, bonus, report

參、Data structures

我們使用了 map, unordered_map,以及 tries 實作我們主要的資料結構，以下將針對每個指令分析比較各個結構。

一、使用者 ID 與資料 (不含 history)

(1) login, create, delete, merge

- map : RB tree 的搜尋、插入與刪除，複雜度都是 O(logN)
- unordered_map : 雜湊的最壞狀況是 O(N)，但是實際上並不會有這麼多碰撞，故他的平均複雜度是 O(1)
- tries : 無論哪個指令，都必須對字串的每個字元做 traverse，搜尋是 O(strlen)

(2) find (wildcard)

- 這部分演算法雖然我們已經盡可能優化，但基本上還是需要 traverse 所以 ID，然而我們使用的資料結構都不適合 traverse，所以這裡另外使用 vector ID_table 去紀錄 ID 以便處理。
- 演算法是摘自 Kirk J. Krauss 的 “Matching Wildcards: An Empirical Way to Tame an Algorithm”
  http://www.drdobbs.com/architecture-and-design/matching-wildcards-an-empirical-way-to-t/240169123
  他與普通 wildcard matching 實作不同，用迴圈取代遞迴，用 bookmark 變數紀錄上次做過的位置，並全部使用 pointer 實作，效率非常驚人。

(3) deposit, withdraw, transfer

基本上都是做 find()，在對資料做存取，複雜度與 find()相同

(這裡的 find 指的是用 key 找其 data 的意思)

(4) search

此函式牽扯到 history 的實作--vector，在報告後面會詳細提到。

Pros

- map: Map 是一棵 RB tree，find 的 time complexity 一定是 O(log(mapSize))，不會像 hash 會被 input id 影響
- unordered_map : hash function 採用 BKDR hash algorithm，我們測過隨機生成約 50 萬筆 id，entry 開到一百萬，發現 collision 最多不超過 5，所以 find ID 的 time complexity 接近 O(1)，做查詢非常有效率
- trie : 字典樹，每個節點開 62 個靜態陣列指標，對於新增、刪除、查詢，都只需要 O(string length)

Cons

- map: find, insert 都會需要比 unordered_map 久的時間，RB_tree 都需要 log(N)
- unordered_map : 利用 Hash 會花多一點的 memory，因為有一部份 hash 裡面是空的，不是都有存 data，較花費記憶體空間
- trie : 因為需要開長度為 62 的陣列，每個節點就必須花費大量的記憶體空間，如果字串沒有重複部分，那麼記憶體用量將會非常可觀。而且，trie 靠指標去連接節點，與 linked list 運作模式相同，比較沒有效率

二、 History

我們使用了 vector + pointer 以及 vector + disjoint set、 linked list + pointer 實作 history，兩者皆使用 time stamp 去紀錄 history 的順序。

vector + pointer : vector 裡面，from 方及 to 方各用 pointer 指向對方的歷史紀錄，串聯兩筆資料以利更改。

vector + disjoint set : 沒被 merge 之前，每個現有的 id 都存在，但被 merge 之後，被 merge 方的 id 會變成 merge 別人的那方，所以所有帳戶的 history 資料都要改。我們利用 disjoint set 的搜尋 owner 概念來時做搜尋真正的 id。

以下將針對相關指令分析比較結構。

(1) Merge

如同 merge sort, 利用 time stamp 排序，複雜度 O(vectorSize1 + vectorSize2)。

(2) Transfer

增加 history:history.push_back(): amortized time 是 O(1)，並連接兩者指標 ，也是 O(1)

(3) Search

Traverse 他的 history vector: O(vectorSize)

Pros

- vector + pointer :

(a) 因為 history 只需要 insert 沒有需要 delete，所以使用 vector 的 push_back 即可 O(1)達成 insert

(b) 當每兩個 account 有 transfer 關係時，都會用 pointer 互相指向對方，所以當 merge 需要改第三者資料時，找到第三者該筆資料 time complexity 是 O(1)，而且改第三者 id 時是和 merge 一起做，所以不會造成額外 traverse 的狀況

(c) vector 的 traverse 比 linked list 快

- vector + disjoint set : 實作上比較不容易出錯(與 pointer 相比)
- linked list + pointer : 可以做到真正 O(1)的 delete

Cons

- vector + pointer :
  (a) Merge 時需要第三個的 vector 來存 merge 結果，不能像 linked list 直接在兩個 linked list 中互相 merge，vector 的 merge 會消耗比較多 momery
  (b) pointers 的指向在實作上容易出錯
- vector + disjoint set: 在找真正 id 時所花的時間比較 pointer 長
- linked list + pointer : linked list 的 traverse 與 vector 相比，效率較低

三、Recommend non-exist ID:

首先從 score1 開始向上窮舉，針對每個 score 列出所有可能的 candidate pattern(ex. abc? abc?? abcd?)，這裡需要用到 subset sum 的遞迴函式，接著每組 pattern 舉出一筆放進一個 priority_queue 中，回傳 pop 的結果，並再從 candidate 中補足一筆進入 priority_queue。

因為只有 10 筆，所以 priority_queue 以 vector 實作，每次都 traverse 陣列找出最小的並 pop。

四、Recommend exist ID:

我們利用 prioirty_queue 及 vector 來實作。另開一個資料結構，以 ID 長度分類，開 vector 陣列，當 ID create 時，分別以長度 push_back 到各自長度的 vector，所以要 O(1)，而 delete 則會在帳戶端存下此 ID 在 vector 的哪個位置，這樣找到位置是 O(1)，接著因為 vector 是亂的，沒有排序，所以 delete 時就將與 delete ID 相同長度的 vector 最後一個 ID 補上，時間複雜度為 O(1)，在將補上 ID 客戶資料中的位置改變，因為沒有 sort 的緣故，每次推薦都要將所選到的 vector traverse 一遍，除了想過 vector，在此之前我們先考慮 link list 因為較好將中間數 delete 掉，可是 link list 的 traverse 比 vector 慢很多，所以並沒有使用。

方法一: 暴搜，使用 priority_queue 來存需要印出的資料，priority_queue 中固定 size <= 10，在暴搜中，每一個 ID 都先計算 score，如果 priority_queue 的 size < 10，則直接 push 到 priority_queue ，如果 size = 10，則將目前比對的 IDscore 和 priority_queue 的 ttop 比較，如果前者較小，則將後者 pop 出去，再將前者 push 進 priority_queue ，否之，如果後者較小，則不做任何事，繼續比對下一筆 ID，如果相同則用 strcmp 來比較，處理方式同上。

Pros:

a. 因為 priority_queue 中固定 size 為 10，所以雖然 insert 是 O(log n)，但也可視為是 O(1)
b. priority_queue 可以將最大的數放在最上面，所以比對起來很快，再加上會自己維護，因此每次取到的值一定是最大的數

Cons:

a. 因為需要比較的關係，priority_queue 是由大到小，所以要印出來時要先放到 vector 中反轉在印出來

方法二: 每次推薦 ID 時，從與不存在 ID 長度相同的 vector 開始找，並找長度正負一的範圍，將全部 traverse 一次，如果 priority_queue 的 size < 10 或 priority_queue 的 top 的 score 大於目前相差長度加一的 score 的話，就繼續往外推薦下去，反之則輸出答案。

(ex 假設要推薦長度相差正負二，則須 priority_queue 的 size < 10 或 priority_queue.top 的 score >= 3，反之，如果 priority_queue 的 size = 10 且 priority_queue.top 的 score < 3 則將答案輸出出來)。

Pros:

a. 能減少許多不必要的 traverse

Cons:

a. 需要許多的判斷，會將速度拖慢

五、總結

以下是三種資料結構的表格整理:

	unordered_map	map	trie
insert	O(1)	O(log N)	O(length)
delete	O(1)	O(log N)	O(length)
find	O(1)	O(log N)	O(length)
memory	O(Entry)	O(#AccountNum)	O(Strlen * #Word)
implement	modern	modern	difficult
higest score	464415.000000	420958.000000	442348.000000
average score	446428.000000	415437.000000	429843.000000
speed	fastest	slowest	median

肆、Optimize & the best data structure we recommend

1. 輸入指令時 strcmp 整個字串，只看 index 的第四個字元，因為所有指令的第四個字元都是不同的，所以有足夠的代表性

login,create,delete,deposit,withdraw,transfer,merge,search,find

輸入 800 萬筆指令的時間比較，得知 strcmp 第三個字元微快於 strcmp 整個字串

strcmp 整個字串: 1.457sec

strcmp 第三個字元: 1.190sec

2. 主體資料結構: Unordered map

因為 login, create, delete 的時間複雜度平均都是 O(1)

3. 歷史紀錄: vector+pointer

merge 的時間複雜度是 O(vectorSize1 + vectorSize2)，改 id 的時間複雜度則是 O(1)

4. 推薦已存在 ID: 線性搜尋

我們這部分因為有用 vector 存起來，traverse 的速度比 map 或 unordered_map 快很多，所以最後選擇 O(n)的線性搜尋，對所有 ID 比對並計算 score 放進 priority_queue 中。

5. 推薦不存在 ID: 窮舉 score 即 pattern

雖然有想過使用遞迴的方法，但發現 score 3 以上的資料組合過多，超過二十萬種，會造成計算上的效率降低，而且遞迴的 coding 難度也較高，比較多 if-else 例外判斷。所以我們最後仍推薦採取窮舉所有 pattern 的方法，再針對每個 pattern 窮舉一筆並 sort，回傳這個 priority_queue 的 top。

6. 改寫 md5 : 原檔案過度頻繁的使用 c++ string 與 C string 互相轉換，我們改成只有 C string 的處理，增加效率。

伍、How to compile and use the system

我們撰寫了 Makefile，只要執行 make 即可編譯出執行檔案。

make

./final_project

陸、Bonus

一、指令顏色

1. Content: 針對指令的成功與否輸出紅色或綠色; 推薦 ID 部分，因為指令沒有成功與否的元素，故使用黃色; 倒數秒數用藍色
2. 利用 linux terminal 的跳脫字元達到著色的功能，無須安裝 package
3. Why deserve bonus: 讓使用者清楚的藉顏色判斷系統回應的資訊

二、 帳戶的金額

1. Content: 輸入 showMoney 可以顯示當前帳戶的存款金額
2. Why deserve bonus: 讓使用者知道有多少錢可以花

三、 打錯 password

1. Content: 打錯 password 三次之後就要隔 10 秒才能再輸入 password
2. 用 thread library 中的 this_thread::sleep_for 和 chrono library 中的 chrono::seconds(1)
3. Why deserve bonus: 終結盜密碼者

四、變更密碼

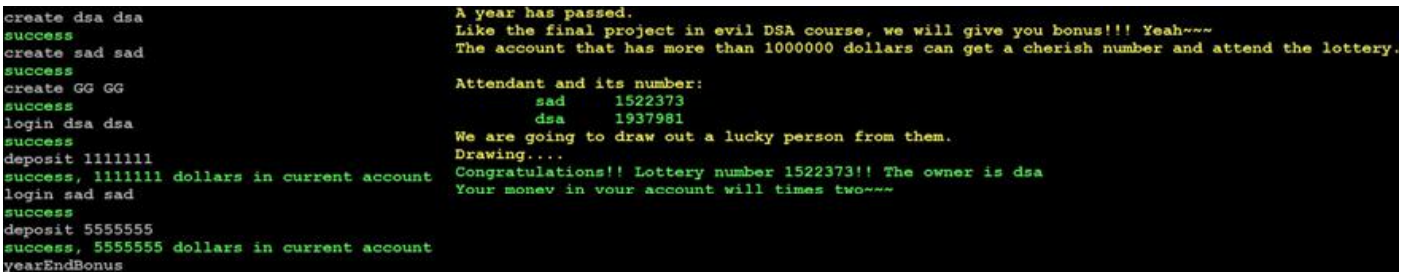
1. Content:輸入 changePassword 可以變更 last_login_ID 的密碼，打錯 password 三次之後直接強制登出，強制登出後須重新登入，否則銀行個人功能無法使用
2. 打錯 password 三次之後直接強制登出的作法是將 last_login_ID 改成不存在的 id，使其無法做任何需要登入的操作
3. Why deserve bonus: 時常更改密碼會使帳戶更安全

五、歷史紀錄

1. Content: 列出所有交易紀錄
2. Why deserve bonus: 可以很快知道交易情形及確保自己帳戶沒有被盜

六、年終抽獎活動

1. Content: 每個 money 超過 1000000 或的 account 都會隨機得到一個序號，可參加年終抽獎活動，由系統抽出一位幸運兒，可以將 money*2
2. 用 stdlib.h 中的 srand(), rand()即 time.h 中的 time(NULL)來隨機產生序號
3. Why deserve bonus: 鼓勵使用者存錢及吸引民眾使用我們的銀行



七、多國貨幣

1. Content:輸入 ForeignCurrencyExchange 可以兌換其他的貨幣，輸入 ShowExchangeRate 可以看各貨幣兌換的匯率。
2. 支援 3 種貨幣：NTD、USdollar、RMB
3. 匯率會根據系統中各種貨幣的總值而改變，模擬變動匯率
4. 貨幣總值多表示不值錢，貨幣總值少表示值錢，所以當總值多的貨幣換成總值少的貨幣時，匯率會小於 1，反之大於 1
5. Why deserve bonus: 真實世界不只一個國家，因此貨幣有很多種，各貨幣之間要能互換，但非 1 比 1