FINAL PROJECT REPORT

TEAM MEMBER

B03902017 曹峻寧　　B03902071 葉奕廷　　B03902104 趙一穎

TASK DISTRIBUTION

所有資料結構/分割討論、Code Optimization、Report Writing 皆由三人共同完成

曹峻寧：

```
create (score2) / transfer
```

葉奕廷：

```
create (score1,general solution) / delete / login / deposit /
withdraw / merge (Linked  list, vector search)
```

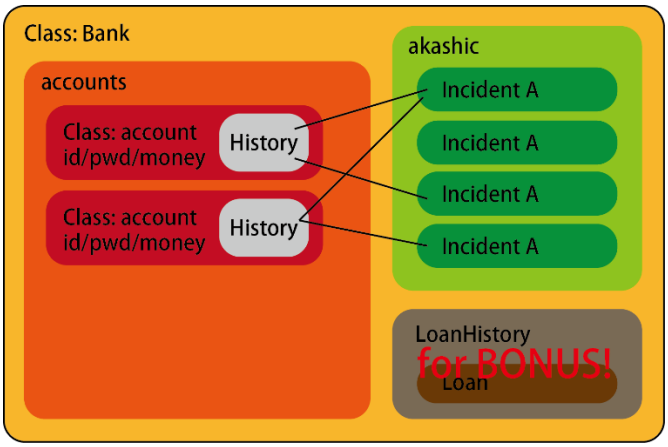
趙一穎：

```
find / map->unordered map / bonus part
```

ABSTRACT

這次 final project 的核心是建構一個銀行帳戶管理系統，並支援一些基礎的操作，並比較不同資料結構的差異。在討論之後我們把要儲存的資料劃分成兩塊，一塊是 History，負責儲存銀行內所有轉帳的紀錄，另外一部分是 user account，負責儲存帳號資訊。以下我們會分別對於 user account 的資料結構和 History 部分的資料結構做討論，並討論簡單的優化方式，最後是 bonus 功能和 compile 原始碼的簡單介紹。

下圖提供了對我們的系統一個簡單的巨觀圖像：



DATA STRUCTURE FOR user accounts

在儲存所有 user account 的資料方面，我們有比較過三種 Data Structure，分別是

- std :: map (r-b tree)
- std :: unordered_map (hash table)
- trie

在 user account 的操作中，主要對資料結構的操作是 insert / delete / find ，以下我們分別以這三種資料結構對於這三種基本操作作效率分析：

std :: map

map 最大的優點就是好寫。最初我們認為 map 因為是 r-b tree，所以其內部的資料有經過一定程度的排序，在寫 find 時，用 iterator 去遍歷 map 可以省掉以字典序去 sort 的麻煩，但 map 的 insert,find 皆為 $\Theta(\log n)$，而大部分的 operation 皆需要找到特定的 account 去進行操作，每次都花費 $\Theta(\log n)$ 的時間顯然不是個好方法。當實作時我們發現，即使 map 本身有 order，find 後還是需要 sort，因此總共的時間複雜度仍為$\Theta(n \log n)$。

std :: unordered_map

unordered_map 是個已經實作好的 hash table，其 insert , find 基本上皆可以維持在 constant time，最差的情況頂多為Θ(bucket 大小)，雖然 find 在輸出前需要多 sort 一次，在 find 裡的平均時間複雜度是$\Theta(n \log n)$ (depends on std::sort)，只要 find 不是找到太多符合的 ID，sort 的效率其實也不會太差，所以整體而言 unordered_map 優於 map。

trie

最後有比較過的就是 trie，trie 的好處在於 find 時可以不用遍歷 account 就可以找到符合的 ID，而且在 insert 和 find 上是Θ(length of ID)，在這次作業 ID 長度被限制<=100 所以差不多能當 constant time，也就和 unordered_map 一樣，但優點似乎有限。

考慮 create 是找沒用過的 ID，當 score 增長時對應到的 ID 的可能性也爆炸式的增長，這顯然在速度上並不會比 unordered_map 用其 find 去查找每個可能的 ID 是否存在來的快，transfer 找已用過的 ID，但我們能想到的唯一方法也就遍歷所有 account，這樣的話其速度和 map 或 unordered_map 是一樣的，而且 trie 太肥，ID 長度每增長 1，就要再多開一個長度為 62 的 trie node array 指向 tree 的下一層，這似乎太浪費記憶體了，考量到時間的投資報酬率，分析完後我們選擇不實作 trie。在最後我們決定主要用 unordered map 去實作。

對於三種資料結構的時間複雜度比較如下表：

Function	map	Unordered_map	Trie
Login	$\Theta(\log n)$	$\Theta(1)$	$\Theta(\text{len})$
Create	$\Theta(\log n)$[1]	$\Theta(1)$	$\Theta(\text{len})$
Delete	$\Theta(\log n)$	$\Theta(1)$	$\Theta(\text{len})$
Find	$\Theta(n \log n)$	$\Theta(n + m \log m)$	$\Theta(\text{len})$
Deposit	$\Theta(1)$*	$\Theta(1)$	$\Theta(1)$*
Withdraw	$\Theta(1)$*	$\Theta(1)$	$\Theta(1)$*
Transfer	$\Theta(\log n)$[2]	$\Theta(1)$	$\Theta(\text{len})$

表一。user account 三種資料結構時間複雜度比較

*註：因我們有記憶 login 的 iterator，故在可以支援$\Theta(1)$的 withdraw 和 deposit

DATA STRUCTURE FOR History

這次 poject 我們被要求儲存每個 user transfer 的紀錄，並在 merge 時將所有有關被 merge 的 user 的紀錄做 user id 的更改。

因為必須快速的存取到所有和特定 user 有關的 transfer 紀錄，所以我們的作法是將所有的 transfer 紀錄儲存到一個大的 array 命名為 akashic，每個 user 的 account 裡存的是一個指標陣列，內容指到 akashic 裡面和他有關的紀錄，這樣能在 merge 時快速的將有關特定 user 的紀錄上的名子做更改。

因為 search 時必須要照時間順序去輸出，為了節省 sort 的時間，我們決定在 merge 時就將所有紀錄的指標照著 timestamp 大小去排列，這時我們比較了兩個資料結構用來儲存指進 akashic 的指標：vector 和 Linked list。

在第一次實做的時候我們的選擇是 vector，因為 vector 非常好寫而且是非常直觀上的選擇。使用 vector 的話在 merge 時的演算法有兩種:

以下 merge_id 稱 a , merged_id 稱 b，

(1) 是全部 push 進 a 的 vector 後對 vector 進行 std::sort

(2) 多開一個 vector 叫 temp，同時遍歷 a,b 的 vector 並不斷將所指紀錄的 timestamp 相對較小的指標 push 進去，在最後把 temp 指定給 a 就好。

第一種的時間複雜度是$\Theta(\text{n} \log n)$，第二種是$\Theta(\text{n})$ ，n 為 a,b 的 vector 大小總和。很明顯的第二種是個比較好的做法，但相對也難寫了一點。在寫第二種的過程我們想到了可以用 Linked list 代替 vector，因為在第二種方法的最後一個步驟必須要進行一次 vector 的複製，複雜度為$\Theta(\text{n})$，但若是用 Linked list 的話只需要將 head node 進行更改就好，所以我們猜想 Linked list 是個更快的選擇。

Linked list 的最大的缺點就是比 vector 難寫非常非常多，因為內建的

[1] 表格中的 create 只考慮未存在的情況，當該帳號存在，時間複雜度由列舉的函式決定。遠大於資料結構中 find 的時間

[2] 這裡的 transfer 也不考慮列舉的情況，如有列舉則時間複雜度皆為$\Theta(\text{n})$

std::list 非常肥，因而我們想自己刻一個簡單的 Linked list，但這就代表在處理時必須要對指標的控制非常小心，否則很容易導致 seg fault，merge 時還得 new 一串新的 Linked list 並 delete 掉舊的 Linked list 的 node 否則會出現 memory leak，在記憶體上頻繁的 allocate 和 free 也拉慢了速度，最後寫出來的 Linked list 版本讓我們在 competition 的網站上平均加了 10000 分。

Function	std::vector	Linked list
Merge	$\Theta(n \log n) or \Theta(n)$	$\Theta(n)$ (but don't need to copy element)
search	$\Theta(n)$	$\Theta(n)$

表二。History 的兩種資料結構時間複雜度比較

Recommendation

如果追求最高速，同時在消耗記憶體上達成平衡，我們推薦在 user id 部分採用 unordered map ，在 history 的部分採用 Linked list。這考量了這次 project 的一些限制，詳述如下：

在這次 project 中，考慮正常的銀行系統運行狀況，最常會使用到的應該不會是 merge,find 這種複雜的指令而是像 login,deposit,transfer,create,delete 之類的簡單指令，而這些簡單的指令都要找到特定的 account 才能進行操作。unordered_map 的 hash_table 可以提供 O(1)的查找、插入和移除，會是我們認為最有利的選擇。而且在正常 user 使用的銀行裡，user 所取的 ID 應該是隨機散布的，所以可以預測到 unordered_map 的 hash 不會有太多 collision 必須處理。在 create 的推薦 ID 中，我們除了通解外還做了 score1,score2 的特殊解去加快系統運行的速度，但不管是通解還是特殊解，在演算法中都必須頻繁地去查找符合該 score 的 ID 是否已經存在，這讓 unordered_map 完全優於 map 和 trie 成為我們的最佳選擇。

Linked list 成為我們推薦的資料結構的原因有二，一是因為我們自己刻 Linked list 可以讓其記憶體用量比 std::vector 要來的小很多，二是因為 Linked list 只需要儲存 list 的 head 的性質讓我們在處理 merge 時的演算法可以少掉複製 vector 的 O(n)時間。這樣的優點讓我們認為如果有足夠的時間去 implement 和 debug，Linked list 是個完全優於 vector 的選擇。

使用內建的 unordered_map 有兩個缺點，首先是 key 必須是 std::string，若想用 c string 必須要自己去寫 hash 而沒有 std::hash 可以使用，使用 std::string 代表 scanf 進 c string 後還得進行一層轉換，這會消耗一些時間。另外 unordered map 基本上會占用比 map 更多的記憶體空間。

而使用 Linked list 的缺點是維護上的困難，若操控指標不當的話容易造成

memory leak，且在 merge 時會頻繁地去 new 和 delete 掉之前的 node，若是採用更複雜的判斷去避免 allocate 和 delete 實際上會造成程式碼更難維護，也更不容易發現錯誤。

簡單來說 Linked list 的最大缺點就是非常非常難寫和不好 debug，寫出來的 code 也缺乏「美感」，大概是 2 個禮拜後自己也看不懂的程度，所以若是有足夠的時間靜下心來去寫它，它會是個很好的選擇，當然這前提常常不成立。

Optimization

Optimization 1 input/output command

最簡單的優化方式就是處理 c++ 的 iostream 或是 c 的 stdio。根據 competition 網站的測試結果平均可以多處理 40000 到 50000 筆指令。如果要要求效率，基本原則是兩者只用一種。這有兩種做法：

(1) 全部改成 c type 的 printf 和 scanf，但是因我們 map/unordered_map 的限制，key 的部分還是必須採用 std::string，所以我們的做法是先用 scanf 吃進 c type string 再用 std::string 的 constructor 將 c type string 構建成 std::string。

(2) 另一種是全使用 cin / cout，並且關掉 iostream 與 stdio 同步(sync)的功能，可以達到只略差於 scanf / printf 的效率。

兩種做法的效率差異主要是在吃 c type string 上和輸出時要用 std ::string.c_str()，兩者比較起來，第二種似乎會有比較好的效率。但我們是採用第一種作法，原因其實是 printf/scanf 控制格式上比較方便。但無論採用哪種作法，只採用一種而避免掉 iostream sync with stdio 都可以提升一定的效率。

Optimization 2 Fuction score optimization

Score 的計算和窮舉是這次 project 裡面比較麻煩的部分，尤其在處理任意的 score，當 score >= 5 之後可能性就會大幅成長。我們在 find_score 函式的通解上採用遞迴，這也代表著效率將隨著遞迴深度大幅遞減。

但實際上用到 score 的函式(create)在 id 重複的情況只要推薦 10 個 score 最小 id。以銀行帳戶的情境而言，取到非常類似 id 的可能性很低 (id 的基數是 1.76 * 10^179 ，應是離散的分布)因此我們合理推估在 create 中主要探討的 case 和計算量會集中在 score = 1 和 score = 2*[3]。

我們優化 create 的方式便是為 score = 1 和 score = 2 的 case 寫特殊解，這個部分是我們 bank.h /bank_um.h 的 best_10_unused_score1 和 best_10_unused_score2。

[3] 根據我們自己產生的測資，create a 要造成 Score3 以上的情況至少要在附近有 3906 個差不多的帳號，在基數如此大的情況帳號要這麼集中的機率實不大，因此以特解處理是很合理的。

Bonus function

BONUS 的部分我們新增了一個借貸功能，在登入的狀態下可以支援 borrow 指令，格式是：

```
# borrow [money] [day]
//money 是所要借的錢
//day 是要借的天數
```

我們計算時間的方式是以每一個 iteration 為單位(1 秒)，一天定義為 3600 秒(也就是 3600 個 iteration，定義在 second_per_day，定義成 86400 會需要太多指令不便測試)。這個借貸系統支援自動結算和利息，當借貸的期限到了就會強制扣除本金加上利息。如果帳戶內剩餘金額不足，便會使帳戶宣告破產。破產後便不能再借錢，也不能被 merge。若該帳戶仍有借貸關係存在，便不能被 merge，但可以被 delete。

實作上我們新增了一個 std::priority_queue，裡面裝的是借貸紀錄，以 deadline 作為 priority 並 overload operator <使其成為一個 min heap。每一個 iteration 在主要 command 執行完後都檢查是否有借貸到期，並進行結算。

使用 std::priority_queue 的優點是我們可以快速的檢查是否有到期，時間複雜度是Θ(1)，且可以自動依 priority 大小排序。但這個做法在 findLoan (尋找一個 id 是否存在借貸關係) 和 deleteLoan (刪除某一個 id 的全部借貸關係，用在 delete 時) 效率不佳，皆需要把所有的 element pop 出來到另外一個 priority_queue，再用新的 priority_queue 取代原來的。這樣的時間複雜度是Θ(2n)，n 是現存銀行借貸契約總數。對於每次處理 merge 前要先檢查 (findLoan，存在借貸關係時不能合併帳戶)，和 delete 時都會耗不少時間，我們覺得還有可以改進的空間。

Compile tool

我們原始碼的編譯方式是使用 Makefile，下面是不同的 makefile 指令支援不同版本

```
#make    預設版本是 umap (recommend)
#make umap  account 用 unorder_map 且 History 採 linked-list 的版本
#make map   account 用 map，History 採 std::vector 的版本
#make bonus      umap 的基礎上加上 bonus part 的功能(loan system)
```

無 Bonus 的可執行檔是 final_project，有 Bonus 的執行檔是 final_project_BUM