

# Data Structure and Algorithm Final Project Report

B00201008 張大衛：負責結構 skip list 部分、report

B00201011 蘇傳堯：負責結構 avl tree 部分

B00201033 楊慕：負責結構 hash map、其他程式部分

## 一、主程式資料結構概述

我們的主程式的架構主要可以分為兩個部分，第一個部分我們稱為 Account Map，這個部分主要的功用為儲存各個帳號，以及帳號裡面的所有資訊內容，並且建立可以搜尋的機制，同時也是我們進行三種結構比較的部分，這將在下一個部分內詳細描述。

第二個部分我們稱為 History Map，其主要的功用的是處理每個帳號下面相關的歷史訊息，因為單一帳號相關的歷史訊息可能非常多，而且也涉及到搜尋，因此特別把這個部分獨立處理。這個部分主要需要負責的是存放歷史訊息，以及搜尋的機制，並且要可以從頭走到尾以利將所有的歷史訊息印出。

因此整個程式的架構，是使用 id 作為 key 的 Account Map，透過 id 可以找到每個人的儲蓄以及使用 History Map 所儲存的和這個人有關的歷史訊息。

## 二、History Map 之資料結構

History Map 中的每個節點需要處理存放歷史訊息，有幾個關鍵點是歷史訊息需要有辦法可以從第一筆讀取到最後一筆，同時必須依照時間順序印出，也要能夠處理將兩個節點合併的情況。乍看之下如果一個節點中有  $n$  筆歷史資料，使用 binomial heap

作為節點的話可以很有效率的處理合併，只需要  $O(\log n)$ ，但在從頭掃到尾並且依照時間順序印需來的部分可能會用到  $O(n \log n)$  的時間，並不是很好。

如果以從頭掃到尾來說，使用 list 最為方便，又很剛好的因為排序是照放進去的時間排序，也就是我們所需要的順序，所以這個 list 只需要用  $O(1)$  的插入就可以形成 ordered list，而且印出成想要的樣子也只要  $O(n)$  即可，同時合併的時候就如同第二次作業一樣是合併兩個 ordered list，所以也只需要  $O(n)$  的時間，而在一般生活的情況下，從頭印到尾 (search) 應該遠比合併 (merge) 還要常發生，所以我們決定以 list 作為 History Map 中的節點。而 Map 本身的部分，因為涉及到搜尋，但只有涉及到搜尋單一的 id，因此使用 Hash Map<sup>1</sup>，並且用 id 作為 key。

實作上，在 id 為 A 的 History Map 中，使用 key 為 B 所找到的節點，裡面包含了兩個部分，第一個部分為一些指標，有通往 B 的 History Map 中對應 A 的那個節點的指標，以及指到自己屬於誰的節點，這些互通資訊在合併的時候會用到；第二個部分則是儲存歷史訊息的 list，每個訊息中有是傳入還是傳出，以及金額，另外這邊有分為兩條 list，一條是存與現存的 B 有關的資訊，另一條則是存過去叫做 B，但現在已經消失的資訊，這是用來處理「同名帳號」所設計的。

當這個存在的 B 要消失時，只需要把其他人對 B 代表現在的 list 與代表過去的 list 合併即可。當 B 要合併進 A 時，假設其他人叫做 C，需要處理的部分有 A 中的 B、C 中的 B、B 中的 A、B 中的 B 以及 B 中的 C，用 \* 來表示所有人。首先，將 \* 中的 B 併入對應的 \* 中的 A，再將 B 中的所有 C 併入 A 中對應的 C，最後，再將 B 中的 A 併入 A 中的 A（來自上次合併）即可。

	A	B	C
起始	ABC	ABC	ABC
* B to * A	(AB)C	(AB)C	(AB)C
BC to AC	(AB)(CC)	(AB)	(AB)C
BA to AA	(ABAB)(CC)		(AB)C

<sup>1</sup> 本文中所提到的所有 Hash Map 都是用 STD 中的 unordered\_map，以及預設用在 string 上面的 hash function 來實作。

實際上前兩個步驟可以用一個對所有 \* 的迴圈來寫，先執行步驟一，若 \* 不是 A 或不是 B 就執行第二步，以上都結束後執行一次第三步。

### 三、Account Map 之資料結構

在 Account Map 的部分，主要都是要用 id 作為 key，並且尋找帳號是否存在，若存在，要可以取得該帳號的資訊。這邊我們使用了幾種不同的結構，分別是 AVL Tree<sup>2</sup>、Skip List<sup>3</sup> 以及 Hash Map，前兩者是屬於先排序好的結構，最後一個則是沒有先排序好的結構。

如果只是單純的尋找帳號，那麼使用 Hash Map 絕對是最快的選擇，但是今天有萬惡的 find、search 等，會需要用到萬用字元或是尋找相近的 id，如果沒有類似「鄰近」概念的話，那麼就只能把所有的帳號通通都試一次，所以這時可能前面兩者的優勢會出現。

在搜尋符合的萬用字元時，可以先把前面沒有出現萬用字元的部分擷取出來，就只需要比較前幾個字元和這個擷取出來相同的部分，在字典排序上，這些字會被放在一個區間內，由於前兩者是有順序的儲存資料，我們可以利用這點來縮小需要檢查的範圍與資料數，但由於 Hash Map 沒有相鄰概念，故在搜尋萬用字元時可能仍需要每個字都比較一次。而在需要推薦 ID 時，也可以利用 score 的計分方式，注意到與一個特定字串比較  $\text{score} \leq n$  的情況下，最極端的情況只需要搜尋與輸入 ID 後  $n$  個不同的，以及尾巴多出  $x = (-1 + (1 + 8n)^{0.5}) / 2$ （無條件進位）的情況，這也會被包含在一個區間內。

當然，在整個系統中搜尋單一 ID 的情況還是最常見的情況，甚至 create 可能需要至少十次搜尋單一 ID 的動作才能做完，所以這需要經過測試才能決定。

### 四、三種 Account Map 資料結構測試結果

---

<sup>2</sup> 本文中所提到的 Tree 是使用與之前作業的相同來源的 libavl (<http://adtnfo.org/>) 所實作

<sup>3</sup> Skip List 是徒手刻出來的

## 五、推薦之資料結構與理由

## 六、其他雜項

這次的 final project 中要求把密碼存成 MD5 加密後的格式，這部分我們是直接使用 Open SSL 中的 MD5，之所以選擇它是因為他的程式內部有直接使用組合語言進行加速，效率應該會比較好。

我們這組的分工如標題下方所示，主要就是一個人負責一種 Account Map，其他雜工各個部分統一由一人解決，另外在 git 的部分也使用了 git flow 的想法，使用 develop 統合進度以及主架構，分出稱具有各個 feature 的 branch，每個人就可以在獨自一條 branch 上作業，當作業完成或是主架構有更動時，在將 develop 與 branch 合併。最後寫出的程式具有 library 的模樣，每個結構可以生出獨立的 .a 檔，主架構以及 History Map 也有各自的檔案，較易於更換結構和增加結構。

總結之，我們的程式有兩大部分，Account Map 與 History Map，雖然 Account Map 可能較為平常，但我們認為 History Map 的地方使用的結構可以有效率的進行輸出和合併，不但尋找快速，合併也不會太慢，也配合現實生活中合併較少使用的想法去設計，同時還可以解決同名帳號問題，可算是一個亮點。

## 七、如何編譯與執行程式

使用指令 make 來編譯程式，./final\_project 來執行，它會從 stdin 讀指令並輸出至 stdout。另外可以在 make 的時候使用 ACCOUNT = xxxx 來設定使用哪種結構。詳細的內容可以參考 README.md。