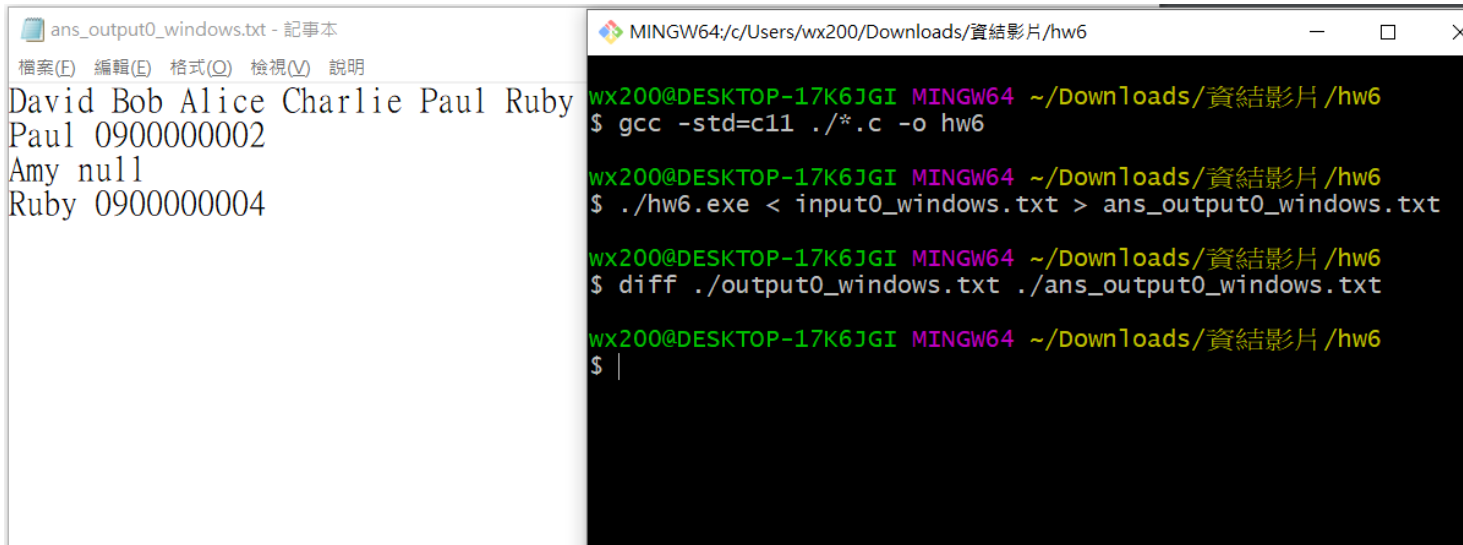


# Result Screenshot



```
ans_output0_windows.txt - 記事本
檔案(F) 編輯(E) 格式(O) 檢視(V) 說明
David Bob Alice Charlie Paul Ruby
Paul 0900000002
Amy null
Ruby 0900000004

MINGW64:/c/Users/wx200/Downloads/資結影片/hw6
wx200@DESKTOP-17K6JGI MINGW64 ~/Downloads/資結影片/hw6
$ gcc -std=c11 ./*.c -o hw6

wx200@DESKTOP-17K6JGI MINGW64 ~/Downloads/資結影片/hw6
$ ./hw6.exe < input0_windows.txt > ans_output0_windows.txt

wx200@DESKTOP-17K6JGI MINGW64 ~/Downloads/資結影片/hw6
$ diff ./output0_windows.txt ./ans_output0_windows.txt

wx200@DESKTOP-17K6JGI MINGW64 ~/Downloads/資結影片/hw6
$ |
```

## Program Structure

程式的結構上主要分為

1. 處理輸入 (在 main 函式裡)
2. 新增節點 (包含處理 Data 的複製和樹的遞迴走訪)
3. 處理 AVL 的高度平衡 (分為 LL、RR、LR、和 RL 情況)
4. 尋找節點 (單純是二元樹的搜尋而已)
5. 用 preorder 印出 AVL 樹

## program functions

這次寫的副程式(函式)較多，主要是希望能讓結構更分明

處理插入節點所使用的函式有

`AVL_Tree* Insert_AVL_Node(AVL_Tree *now , Data *tmp)` 插入節點

接下來則是輔佐插入節點所設計的函式

`int Get_Node_Height(AVL_Tree* now)` 用於取得高度

`int compare_name(const char *a ,const char *b)` 用於字串間的比較

`int compare_data(const Data *a ,const Data *b)` 用於資料間的比較

`LL_rotate` 、 `RR_rotate` 、 `LR_rotate` 、 `RL_rotate` 四種情況

而處理搜尋的函式則是：

`AVL_Tree* Search_AVL_Node(AVL_Tree* now , char *name)`

處理題目的走訪使用的函式：

`void pre_order(AVL_Tree *now)`

`Insert_AVL_Node` 的部分主要是

先插入新的節點，再更新節點高度（使用 `Get_Node_Height()`）

如果發現有失衡的情況，就開始去作旋轉(四種旋轉中的一個)

LL 情況的話則進行順時針旋轉

RR 情況則進行逆時針旋轉

LR 的話 先對左子樹作 RR 旋轉，再對自己作 LL 旋轉

RL 的話，先對右子樹作 LL 旋轉，再對自己作 RR 旋轉

LL\_rotate 、 RR\_rotate 、 LR\_rotate 、 RL\_rotate

這四個函式就是上述四件事

由於更新高度的情況都是從下往上更新的

因此發現失衡時的節點會是最靠近的節點，就不用多加判斷了

而尋找節點的部分就跟之前的二分搜尋樹寫法一樣，

往下遞迴就可以了

Search\_AVL\_Node()就是搜尋函式

在此題比較特別的是比較的部分

平常在寫樹時是利用數字的大小去作比較依據

該題則是使用字典排序依據

因此我的 compare\_name(const char \*a ,const char \*b)

就是在作字典比較依據的，其他都跟一般的 AVL 樹一樣

而 compare\_data()內部程式碼只是去呼叫上面的 compare\_name()

最後再寫好 preorder()，該題就結束了。