


## (1) Result Screenshot

```
Jamei@DESKTOP-PGELV1B MINGW64 ~/Desktop/E94086107_張娟鳴/code
$ ./hw6<input0_windows.txt>output.txt

Jamei@DESKTOP-PGELV1B MINGW64 ~/Desktop/E94086107_張娟鳴/code
$ diff -c output.txt output0_windows.txt
```

Screenshot of command line

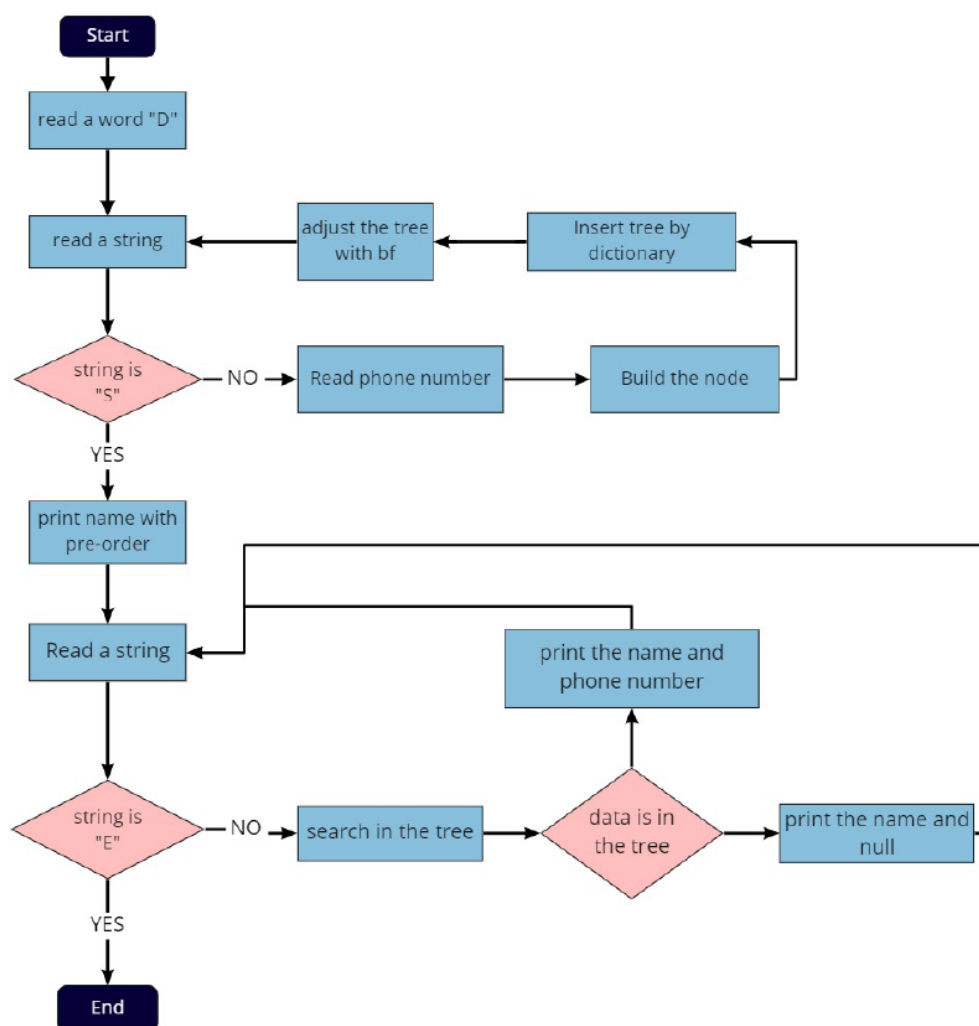
 output.txt - 記事本

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
```

“output.txt” screenshot

## (2) Program Architecture



### (3) Program function

```
struct treeNode{
    char phone[11];
    char name[21];
    int bf;
    struct treeNode * ltree;
    struct treeNode * rtree;
}; //structure of AVL tree
```

#### Parameters

`phone[11]` - the string used to save the phone number.

`name[21]` - the string used to save the name.

`bf` - height of left tree minus the height of the right tree.

`ltree` - a pointer used to point the left side of node.

`rtree` - pointer used to point the right side of node.

```
typedef struct treeNode node;
typedef node * nodeptr;
```

Type define list, I will use this abbreviation later.

```
int flag=0, recalculate=0;
nodeptr root=NULL;
```

#### Parameter

`flag` - 確認是否已輸出任何數字，調整輸出格式

`recalculate` - 確認 AVL tree 的 bf 計算是否已完成，0 代表完成，1 代表還沒

`root` - AVL tree 的頂端

```
int main() {
    char ptemp[11]="";
    char ntemp[21]=""; //初始設定 string 為空字串
    char c1[]="S", c2[]="E"; //比對字串，遇到 S 和 E 時需要改變動作
    scanf("%s", ntemp); //先讀入第一個輸入"D"，此時並不用做任何事
    while (scanf("%s", ntemp) != EOF) {
        if (strcmp(ntemp, c1) == 0) //讀入下一個輸入，觀察他是否為"S"，不是的話就是名字，是的話則結束該迴圈
            break;
        scanf("%s", ptemp); //讀入電話號碼
        insert(ntemp, ptemp, &root); //用名字和電話號碼建立 node 並 insert 進 AVL tree
    }
    preorder(&root); //遇到"S"代表結束輸入，在開始搜尋前先使用 pre-order
```

```

printout 所有名字
    while (scanf ("%s", ntemp) != EOF) {
        if (strcmp (ntemp, c2) == 0) // 讀入字串，如果是 "E" 則結束迴圈，不是則代表字
串為名字
            break;
        puts ("");
        search (ntemp, &root); // 使用名字搜尋
    }
    return 0;
}

```

## Parameter

`ptemp[11]` - 暫時用來記錄讀入的電話，大小和 `node` 裡的 `phone` 大小一樣，方便複製

`ntemp[21]` - 暫時用來記錄讀入的名字，大小 `name` 一樣，兩者初始皆為空字串

`c1[]` - 比較用字串，裡面存 `S`，方便比較輸入是否單為 "`S`" (名字至少 2 個字)

`c2[]` - 比較用字串 `E`，用法和 `c1[]` 相同

## Return value

如果程式順利執行會 `return 0`

## Function

**getnode**-產生新的 AVL tree node 並初始化它

```

nodeptr getnode(char user[], char num[])
{
    nodeptr newptr = malloc(sizeof(node));
    strcpy(newptr->name, user); // 複製 user 裡面的內容給 newptr 指向的 phone
    strcpy(newptr->phone, num);
    newptr->ltree = NULL;
    newptr->rtree = NULL;
    newptr->bf = 0; // 新安插的 treenode bf 值均為 0，因為是 leaf node
    return newptr; // malloc 成功並初始化後回傳
}

```

## Parameter

`user[]` - 要安插的名字，`newptr` 指向的 `name` 是複製這個字串裡面的內容

`num[]` - 要安插的電話號碼，`newptr` 指向的 `phone` 會是複製這個字串裡面的內容

`newptr` - 一個指標，指向 `malloc` 出來的 `node` 空間

## Return value

該程式會把產生完的 `newptr` 會回傳給 `function insert()` 使用

**insert**-安插新節點並調整它使其平衡，和 `BST` 的差別在於調整

```

void insert(char user[],char num[],nodeptr *parent)
{
    if(*parent==NULL) { //如果 node 下面沒有其他數，便可以安插新的資料
        recalculate=1; //此時有部分 node 的 bf 需重新計算
        nodeptr newnode=getnode(user, num);
        *parent=newnode;
    }

    if(strcmp(user, (*parent)->name)>0) { //若想安插的資料之字典順序大於該
node，往右邊探尋
        insert(user,num,&((*parent)->rtree));
        if(recalculate==1) { //安插完後要來調整
            switch ((*parent)->bf) {
                case 1: //原本是左邊比較低，右邊插入後便一樣高
                    (*parent)->bf=0;
                    recalculate=0;
                    break;
                case 0: //原本是一樣高，右邊插入後右邊低一層 (lf-rf=-1)
                    (*parent)->bf=-1;
                    break;
                case -1: //原本是右邊低，右邊插入後低兩層 (lf-rf=-2)
                    rightrotation(parent); //此時需要做 right rotation
                    break;
            }
        }
    }

    if(strcmp(user, (*parent)->name)<0) { //若想安插的資料之字典順序小於該
node，往左邊探尋
        insert(user,num,&((*parent)->ltree));
        if(recalculate==1) { //安插完後要來調整
            switch ((*parent)->bf) {
                case -1: //原本是右邊比較低，左邊插入後便一樣高
                    (*parent)->bf=0;
                    recalculate=0;
                    break;
                case 0: //原本是一樣高，左邊插入後左邊低一層 (lf-rf=1)
                    (*parent)->bf=1;
                    break;
                case 1: //原本是左邊低，右邊插入後低兩層 (lf-rf=2)

```

```

        leftrotation(parent); //此時需要做 left rotation
        break;
    }
}
}

if(strcmp(user, (*parent)->name)==0) ; //如果資料是已經插入過的，不需要安插，也不需要調整
}

```

## Parameter

`user[]` –要安插名字，也是資料比對的基準

`num[]` –要安插的電話號碼

`*parent` –當前所在的 node

`newnode` –新產生的 tree node，要讓某個 node 指向它

## Return value

這個 function 沒有 return value

## search –找尋想要的名字及其電話號碼

```

void search(char compare[], nodeptr *now)
{
    if(*now)
    {
        if(strcmp(compare, (*now)->name)==0) //如果找尋的東西就是該 node，列印
        printf("%s %s", compare, (*now)->phone);

        if(strcmp(compare, (*now)->name)>0) //如果找尋的東西在字典裡比該 node
        大，往右邊找
        search(compare, &((*now)->rtree));

        if(strcmp(compare, (*now)->name)<0) //如果找尋的東西在字典裡比該 node
        小，往左邊找
        search(compare, &((*now)->ltree));
    }

    if((*now)==NULL) printf("%s null", compare); //如果找不到東西，印出名字，但電話印"null"
}

```

## Parameter

`compare[]` –要搜尋的名字，被用來比較的

`*now` –當前所在的 node

## Return value

這個 function 沒有 return value

**preorder**-將 AVL tree 內的資料以 pre-order 方式訪問並 print out

```
void preorder(nodeptr *now)
{
    if(*now)
    {
        if(flag==0) //如果一開始還沒印出任何東西過，直接印資料並將 flag 變 1
        {
            printf("%s", (*now)->name);
            flag=1;
        }
        else printf(" %s", (*now)->name); //如果印過東西了，在每次印東西前先加空白
        preorder(&((*now)->ltree)); //印完當前的東西，往左子樹走
        preorder(&((*now)->rtree)); //走了左子樹換走右子樹
    }
}
```

### Parameter

*\*now* -當前所在的 node

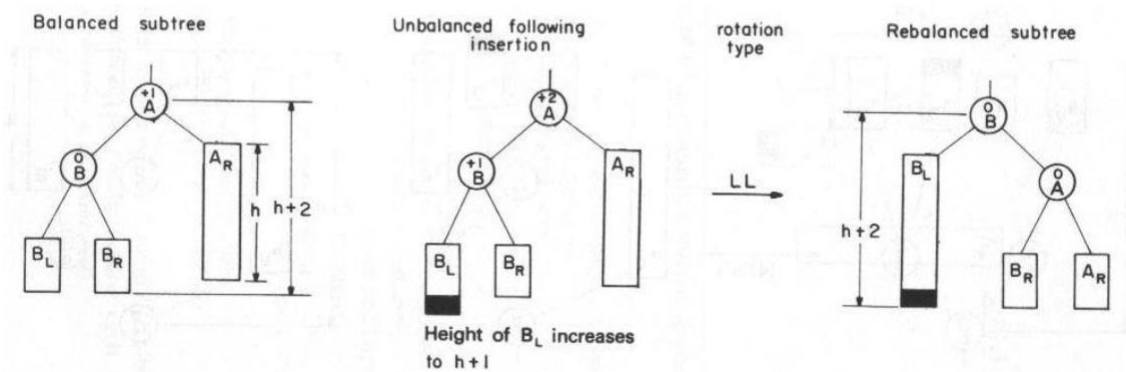
### Return value

這個 function 沒有 return value

**leftrotation** -當有 node 為+2 時，要做左邊子樹的旋轉

```
void leftrotation(nodeptr *parent)
{
    nodeptr grandchild, child;
    child=(*parent)->ltree; //左邊要旋轉，所以要討論的 child 為 parent 的左邊
    if(child->bf==1) { //檢查發生+2 的 node 的 lchild，若是+1 則為 LL，若是-1 則為 LR
        (*parent)->ltree=child->rtree;
        child->rtree=*parent;
        (*parent)->bf=0; //如此修正，原本發生+2 的 node 則變得左右平衡，bf=0
        (*parent)=child; //原本的 parent 被 child 取代
    }
}
```

以上為 LL rotation，child(B)會被提上去，其右子樹(Br)既要在 child(B)的右邊，又要在 parent(A)的左邊，所以 parent(A)的 ltree(AI)會接它(Br)，而 child(B)的 rtree 接 parent(A)，最後，名義上的 parent 變成 child(B)。



```

else{
    grandchild=child->rtree;//不論是 LL 或 LR，child 相對於 parent 的位置
    都一樣，而在 LR 中，grandchild 為 parent 的左邊的右邊
    child->rtree=grandchild->ltree;//grandchild 會被提上去，其左子樹要
    在 grandchild 的左邊，也要在 child 的右邊，所以 child 的 rtree 會接住
    grandchild->ltree=child;//這樣 grandchild 的左邊就可以接 child
    (*parent)->ltree=grandchild->rtree;//grandchild 的右子樹要在
    grandchild 的右邊，又會比 parent 小，所以它會在 parent 的左邊
    grandchild->rtree=*parent;//grandchild 的右邊接 parent
    switch (grandchild->bf) { //重新計算 bf
        case 1://如果 grandchild 下面是左邊低，則 parent 所接的 rf 會導致右
        邊低；而 child 則兩邊等高
            (*parent)->bf=-1;
            child->bf=0;
            break;
        case 0://如果 grandchild=0，代表下面是平衡的，分給 parent 和 child
        後兩個居兩邊等高
            (*parent)->bf=child->bf=0;
            break;
        case -1://如果 grandchild 下面是右邊低，則 child 所接的 lf 會導致左
        邊低；而 parent 則兩邊等高
            (*parent)->bf=0;
            child->bf=1;
            break;
    }
    *parent=grandchild;//變成 grandchile 在正中間了
}

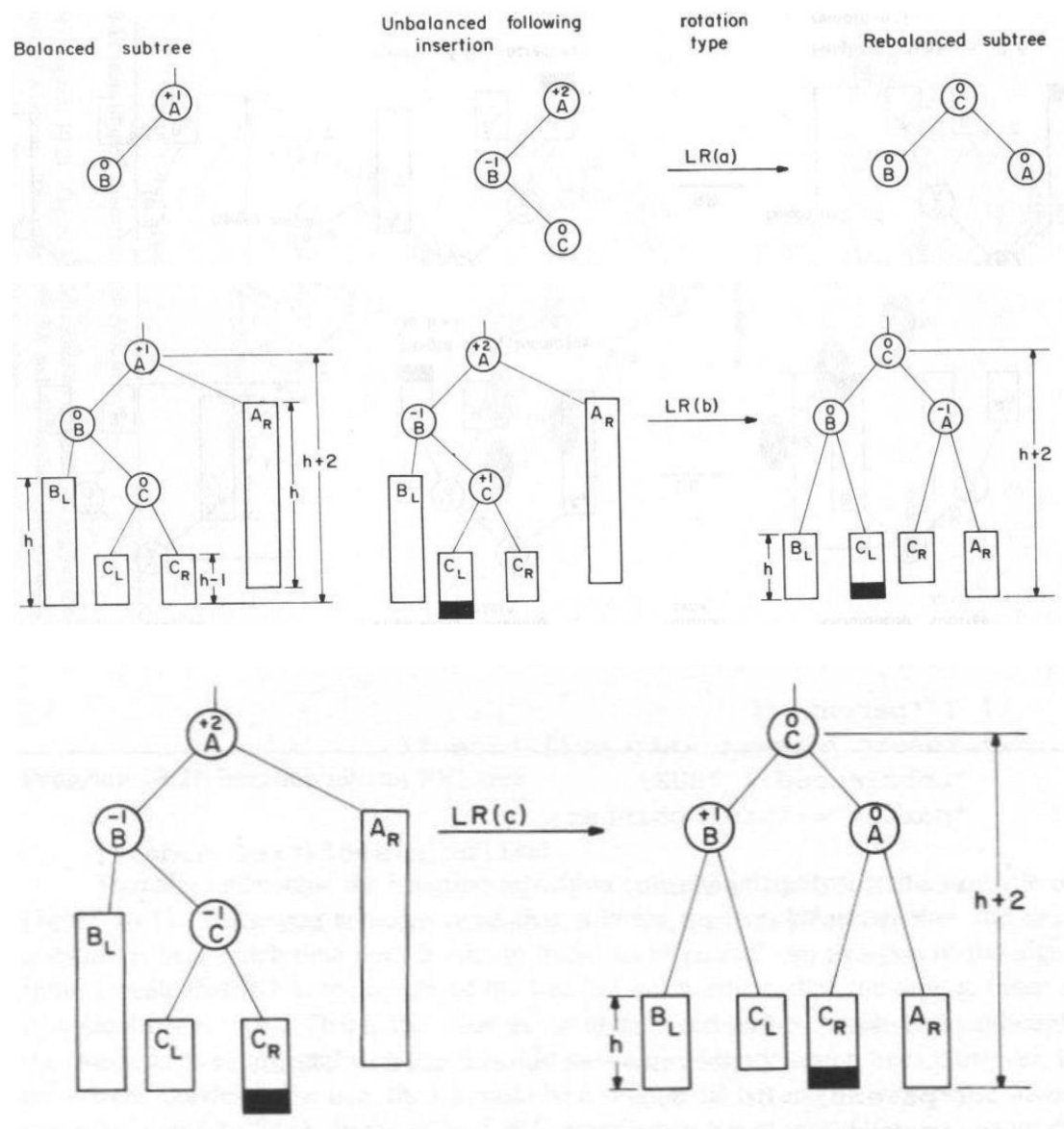
(*parent)->bf=recalculate=0;//最後 parent 的 bf 一定都是 0，並且因為調整

```

過了所以整棵樹可以達到平衡

}

以上為 LR rotation，grandchild(C)會被提上去，其左子樹(Cl)既要在 grandchild(C)的左邊，又要在 child(B)的右邊，所以 child(B)的 rtree(Br)會接它(Cl)；而 grandchild 的右子樹(Cr)要在 grandchild(C)的右邊，又會比 parent 小，所以它會在 parent(A)的左邊，也就是 parent(A)的 ltree(Br)會接它(Cr)；最後，grandchild(C)的 rtree 接 parent(A)，使得名義上的 parent 變成 child(B)。



## Parameter

**parent** -發生不平衡的 node，在 leftrotation 裡 bf 均為 +2

**child** -parent 的 child，因為討論的是 leftrotation，所以 child 都是 parent 的 ltree

**grandchild** -在 LL 時可以不用用到，在 LR 時是代表 parent 的 ltree 的 rtree

## Return value

這個 function 沒有 return value



**rightrotation** –當有 node 為-2 時，要做右邊子樹的旋轉，做法和 left rotation 大同小異(基本上註解也和原本的差不多，只有改相對位置)

```
void rightrotation(nodeptr *parent)
{
    nodeptr grandchild, child;
    child = (*parent) -> rtree; // 右邊要旋轉，所以要討論的 child 為 parent 的右邊
    if (child -> bf == -1) { // 檢查發生+2 的 node 的 lchild 的 bf，若是-1 則為 RR，若是+1 則為 RL
        (*parent) -> rtree = child -> ltree; // child 會被提上去，其左邊既要在 child 的左邊，又要在 parent 的右邊，所以 parent 的 rtree 會接它
        child -> ltree = *parent; // 而 child 的 ltree 接 parent
        (*parent) -> bf = 0; // 如此修正，原本發生-2 的 node 則變得左右平衡，bf=0
        (*parent) = child;
    }
    else {
        grandchild = child -> ltree; // 不論是 RR 或 RL，child 相對於 parent 的位置都一樣，而在 RL 中，grandchild 為 parent 的右邊的左邊
        child -> ltree = grandchild -> rtree; // grandchild 會被提上去，其右子樹要在 grandchild 的右邊，也要在 child 的左邊，所以 child 的 ltree 會接住
        grandchild -> rtree = child; // 這樣 grandchild 的右邊就可以接 child
        (*parent) -> rtree = grandchild -> ltree; // grandchild 的左子樹要在 grandchild 的左邊，又會比 parent 大，所以它會在 parent 的右邊
        grandchild -> ltree = *parent; // grandchild 的左邊接 parent
        switch (grandchild -> bf) { // 重新計算 bf
            case 1:
                (*parent) -> bf = 0;
                child -> bf = -1;
                break;
            case 0:
                (*parent) -> bf = child -> bf = 0;
                break;
            case -1:
                (*parent) -> bf = 1;
                child -> bf = 0;
                break;
        }
        *parent = grandchild; // grandchile 在正中間
    }
}
```

```
    }  
    (*parent)->bf=recalculate=0; //最後 parent 的 bf 一定都是 0，並且因為調整  
    過了所以整棵樹可以達到平衡  
}
```

### Parameter

**parent** –發生不平衡的 node，在 rughtrotation 裡 bf 均為-2

**child** –parent 的 child，因為討論的是 rightrotation，所以 child 都是 parent 的 rtree

**grandchild** –在 RR 時可以不用用到，在 RL 時是代表 parent 的 rtree 的 ltree

### Return value

這個 function 沒有 return value