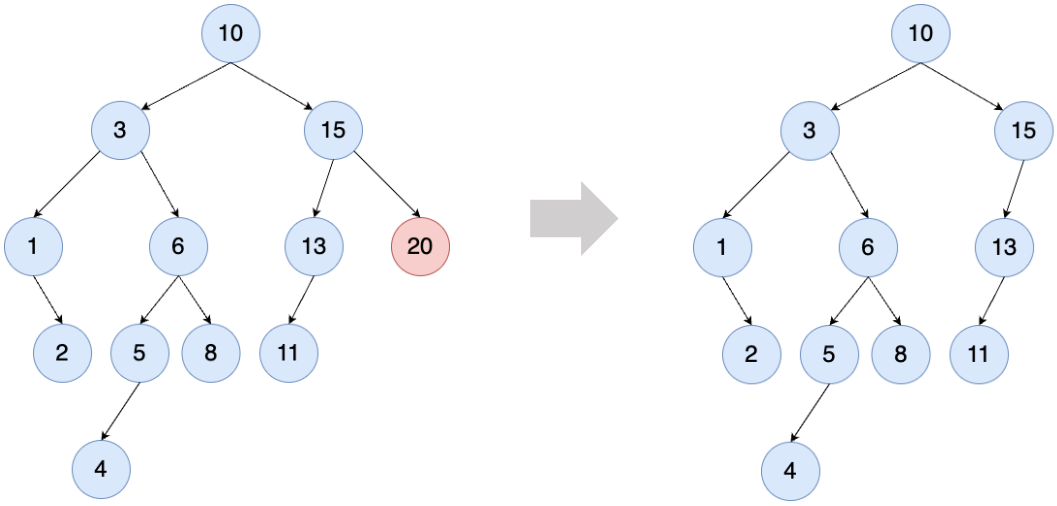
演算法 HW1

姓名：陳品伃 / 學號：112753204

1. 請以圖一為例，說明 Binary Search Tree 如何先後 Delete 20, 6, 15，請敘述過程。

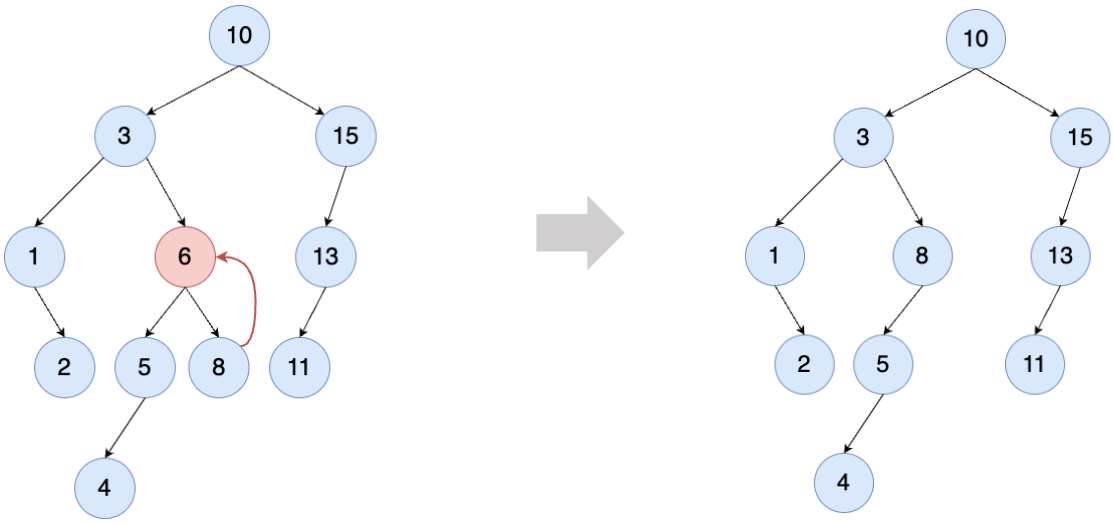
[Delete 20]

因為 20 這個 node 下面沒有任何的 child，所以可以直接進行刪除。



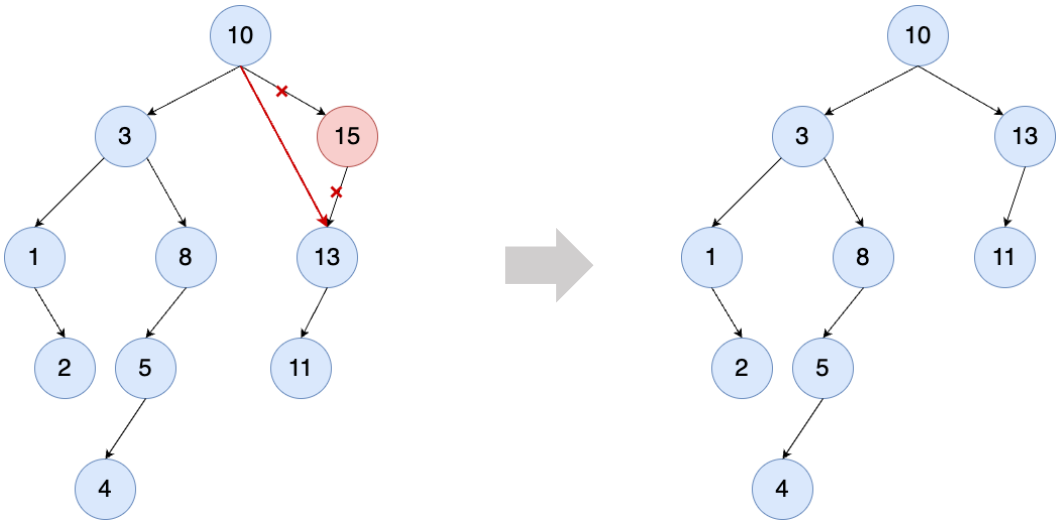
[Delete 6]

因為 6 這個 node 有兩個 children，若要 delete 6 必須將以 6 為 root 的右子樹中最小節點 8 (6 的 rightchild)移至 6 的位置，並將 3 (原 6 的 parent)的 pointer 指向 8。



[Delete 15]

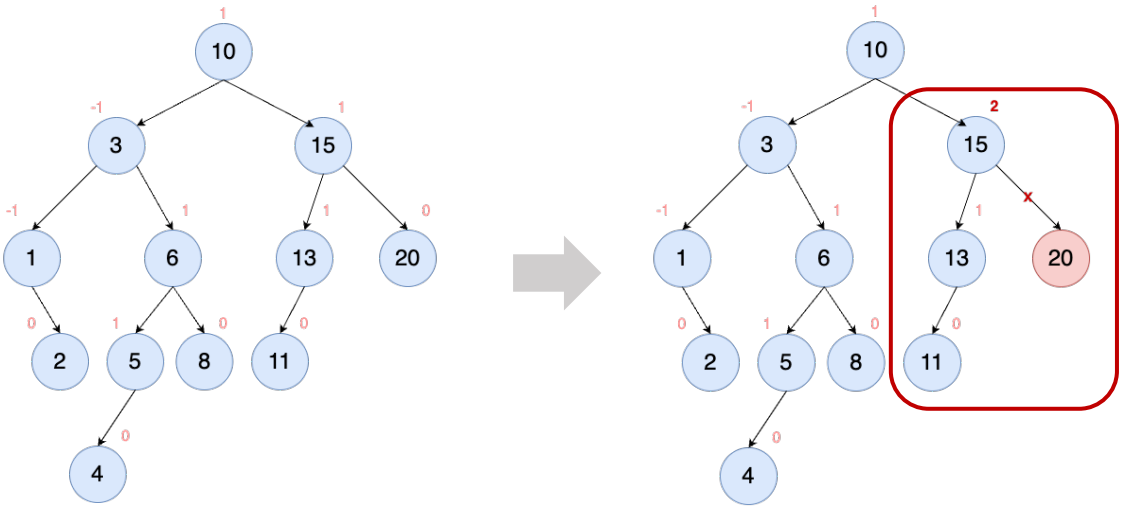
15 這個 node 有一個 child 13，而 13 這個 node 也有一個 leftchild 11，所以選擇將 10 (15 的 parent)的 pointer 直接指向 13，因為 13 原本就在 10 的右子樹中，所以這樣並不會影響 Binary Search Tree 的正確性。



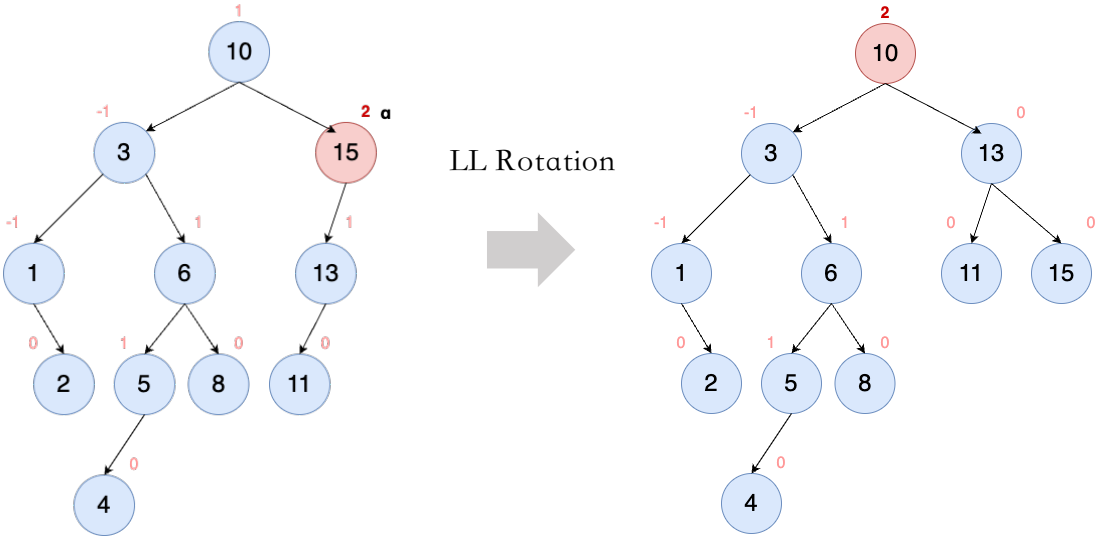
2. 請以圖一為例，說明 AVL Tree 如何先後 Delete 20, 6, 15 (包括必要時 Rebalance 的過程)，請敘述過程。

[Delete 20]

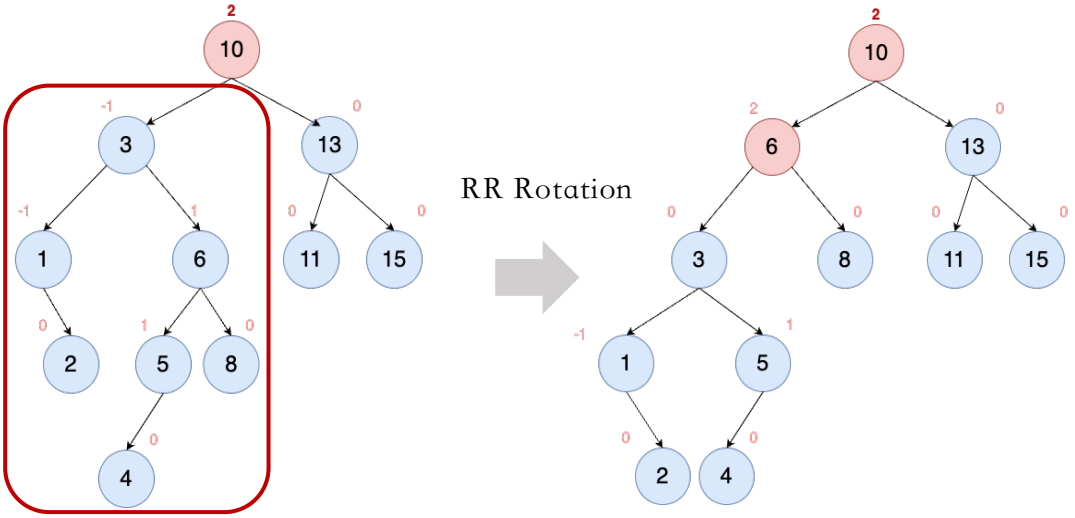
當刪除 20 時，雖然他沒有任何的 child 可以直接刪除，但是會破壞此 AVL Tree 右子樹的平衡，因此需要先將右子樹進行 LL Rotation。

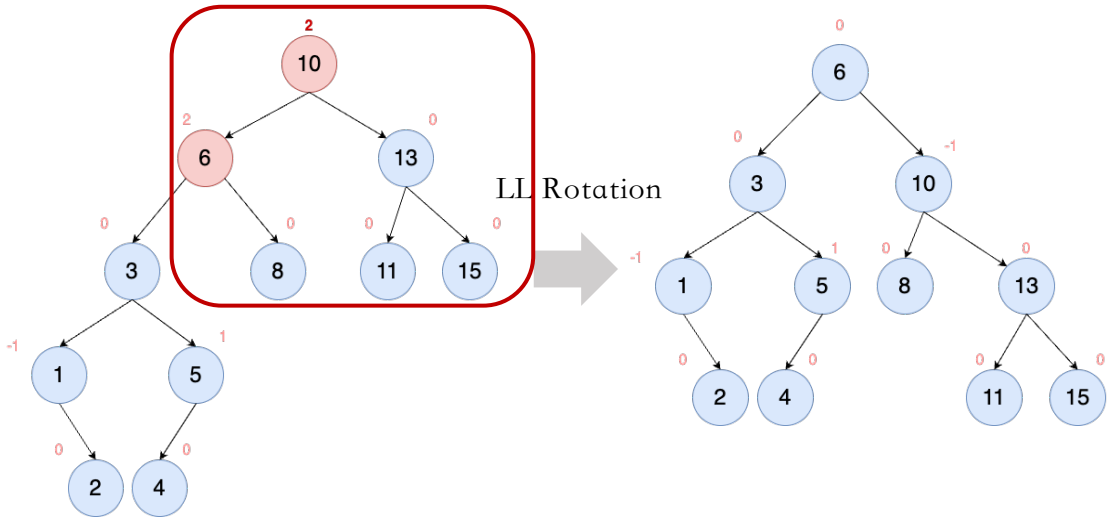


先將距離 delete node 最近的不平衡點 15 當作 α 來進行 LL Rotation，做完 Rotation 後會如下圖，變成 root 10 不平衡，因此需要再做一次 LR Rotation 才能將此 AVL Tree 平衡。



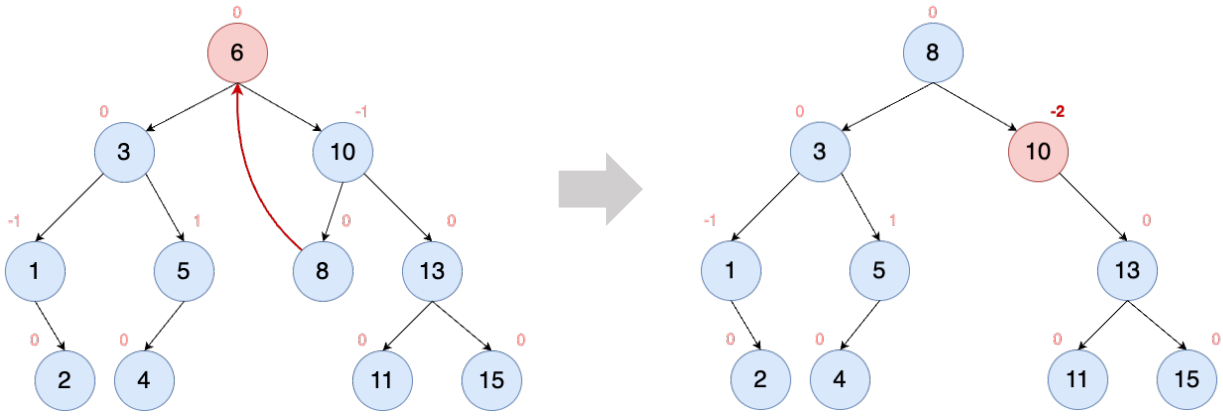
以 3 當作 α 先進行 RR Rotation，接著，再以 10 為 α 進行 LL Rotation，就完成了 LR Rotation，此 AVL Tree 也平衡了。



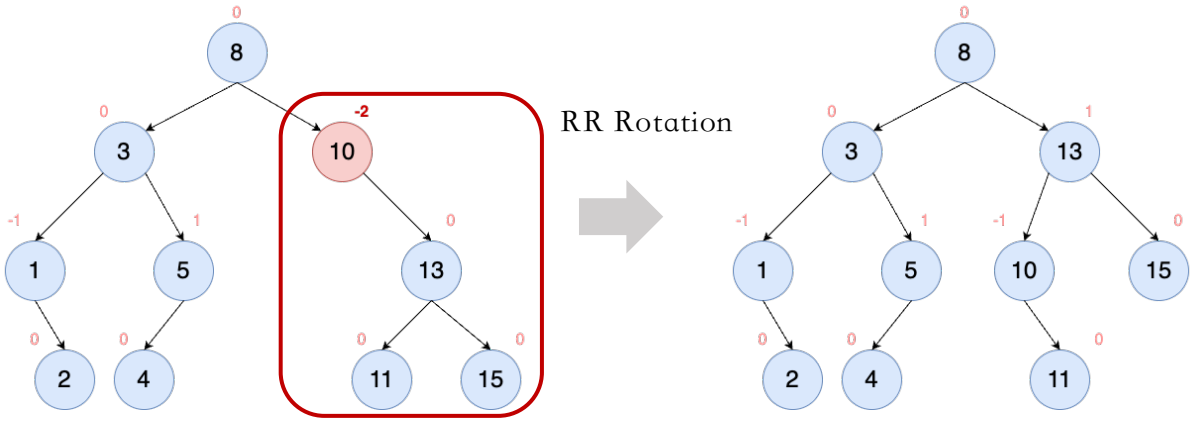


[Delete 6]

因為 6 有兩個 children，若要刪除 6 就如同 Binary Search Tree 的方法，需要從它的右子樹中尋找最小的 node 來遞補。完成後就會如下圖，在 10 這個節點的地方出現了不平衡，因此需要透過 RR Rotation 來進行平衡。

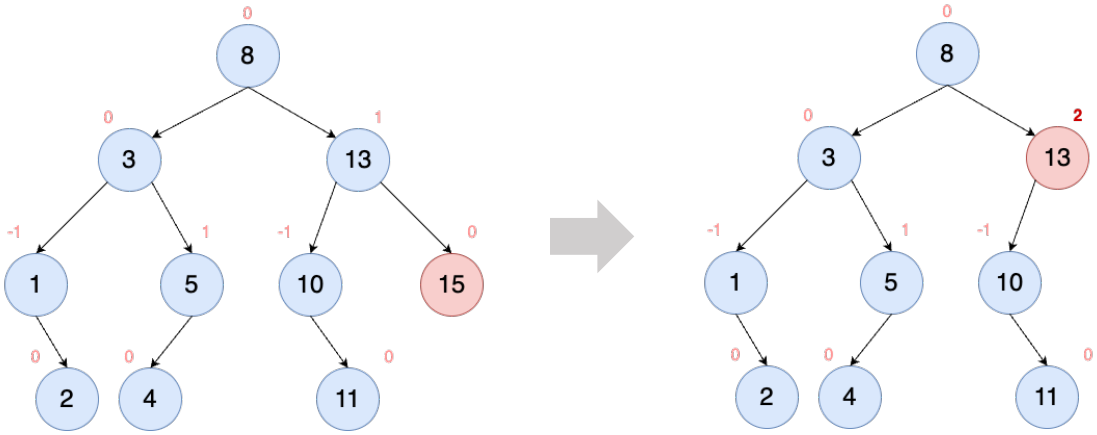


針對以 10 為 α 的子樹來進行 RR Rotation，最後就平衡了這個 AVL Tree。

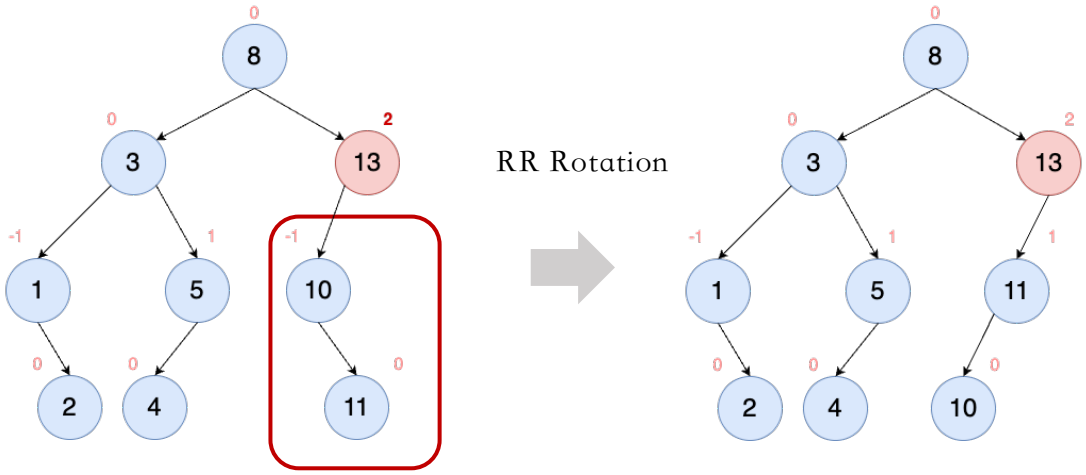


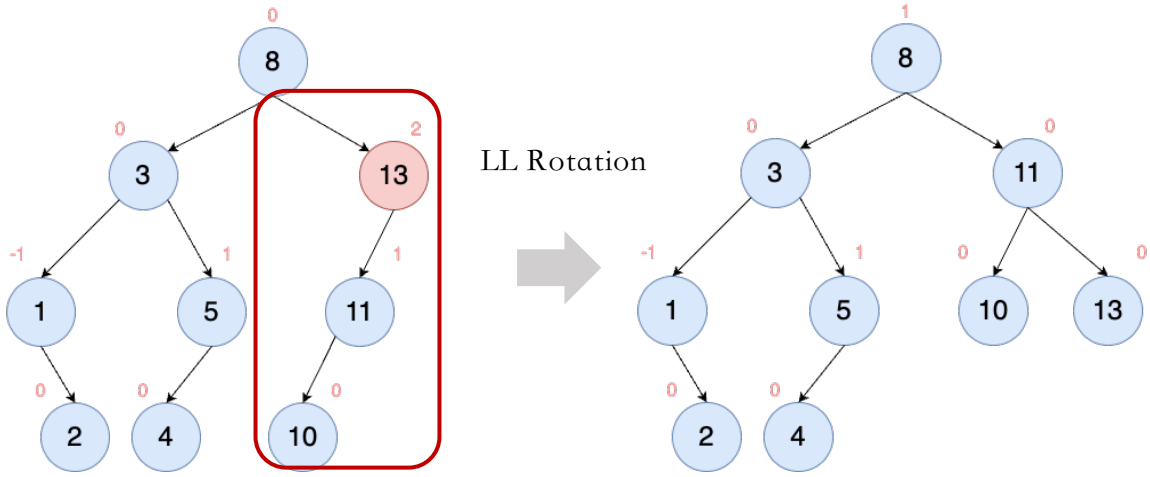
[Delete 15]

在刪除 15 後，會造成右子樹的不平衡，因此需要使用 LR Rotation 來平衡。



首先，以 10 為 α 進行 RR Rotation，調整完之後再以 13 為 α 的子樹做 LL Rotation，也就完成了此 AVL Tree 的平衡。





3. 寫出 AVL Tree Deletion 的演算法（參考投影片 Binary Search Tree Deletion 及 AVL Tree Insertion 演算法的寫法）

AVLpractice.cpp > main()

```cpp
#include <stdio.h>
#include <stdlib.h>

struct node
{
    int key;
    struct node *left;
    struct node *right;
    int height = 1;
};

// 在右子樹中尋找最小的node來遞補被delete的node
struct node *FindMin(node *node)
{
    struct node *current = node;
    while (current->left == NULL)
    {
        current = current->left;
    }
    return current;
}

int node_height(struct node *root)
{
    if (root == NULL)
    {
        return 0;
    }
    else
    {
        return root->height;
    }
}

int node_balance(struct node *root)
{
    if (root == NULL)
    {
        return 0;
    }
    else
    {
        return node_height(root->left) - node_height(root->right);
    }
}
```

```c
int Max(int root_left, int root_right)
{
    if (root_left > root_right)
    {
        return root_left;
    }
    else
    {
        return root_right;
    }
}

// turn to left要做RR rotation
struct node *RR_rotation(struct node *root)
{
    struct node *root_right = root->right;
    root->right = root_right->left;
    root_right->left = root;

    root->height = Max(node_height(root->left), node_height(root->right)) + 1;
    root_right->height = Max(node_height(root_right->left), node_height(root_right->right)) + 1;
    return root_right;
}

// turn to right要做LL rotation
struct node *LL_rotation(struct node *root)
{
    struct node *root_left = root->left;
    root->left = root_left->right;
    root_left->right = root;

    root->height = Max(node_height(root->left), node_height(root->right)) + 1;
    root_left->height = Max(node_height(root_left->left), node_height(root_left->right)) + 1;
    return root_left;
}

```

```c
struct node *Delete(int X, struct node *root)
{
    struct node *TmpCell = NULL;
    // 找不到任一node 此AVL Tree為空
    if (root == NULL)
    {
        printf("Element not found");
        return NULL;
    }
    else if (X < root->key)
    {
        root->left = Delete(X, root->left);
    }
    else if (X > root->key)
    {
        root->right = Delete(X, root->right);
    }
    // 找到要刪除的node後，判斷它是否有child
    // 當這個node有兩個children時，找到右子樹中最小的node來遞補
    else if (root->left && root->right)
    {
        TmpCell = FindMin(root->right);
        root->key = TmpCell->key;
        root->right = Delete(root->key, root->right);
    }
    else
    {
        TmpCell = root;
        if (root->left == NULL && root->right != NULL)
        {
            root = root->right;
        }
        else if (root->right == NULL && root->left != NULL)
        {
            root = root->left;
        }
        free(TmpCell);
    }

    root->height = Max(node_height(root->left), node_height(root->right)) + 1;

    int bal = node_balance(root);

```

```
126     if (bal > 1)
127     {
128         if (node_balance(root->left) < 0)
129         {
130             root->left = RR_rotation(root->left);
131             return LL_rotation(root);
132         }
133         else if (node_balance(root->left) >= 0)
134         {
135             return LL_rotation(root);
136         }
137     }
138     else if (bal < -1)
139     {
140         if (node_balance(root->right) > 0)
141         {
142             root->right = LL_rotation(root->right);
143             return RR_rotation(root);
144         }
145         else if (node_balance(root->right) <= 0)
146         {
147             return RR_rotation(root);
148         }
149     }
150
151     return root;
152 }
153
154 int main()
155 {
156     struct node *base = NULL;
157     base = Delete(10, base);
158     base = Delete(15, base);
159 }
```

4. AVL Tree Deletion 時的 Rebalance 與 Insertion 時的 Rebalance 有何不同？為什麼？

Insertion

在做 Rebalance 時只要隨著依循而來的路徑進行 rebalance 就好，至於如何決定做什麼樣的 Rotation，也是依循著一開始從 root 走到新節點的路徑來判斷是要做 LL/RR/LR/RL Rotation 哪一種旋轉方式。

Deletion

Deletion 與 Insertion 的不同之處在於它**需要整棵樹重新計算高度並進行 rebalance**。Deletion 在檢查的過程中一但發現有不平衡的節點就要立即做 LL/RR/RL/LR Rotation，並且在旋轉完後要再從頭計算一次每個節點的高度。

之所以 Insertion 及 Deletion 的 Rebalance 會有所不同是因為在 Insert 時除了 Rotation 之外，不會動到其他節點的位置。而在 Delete 時有可能需要移動部分節點位置，因此會需要重新計算整棵樹的高度。