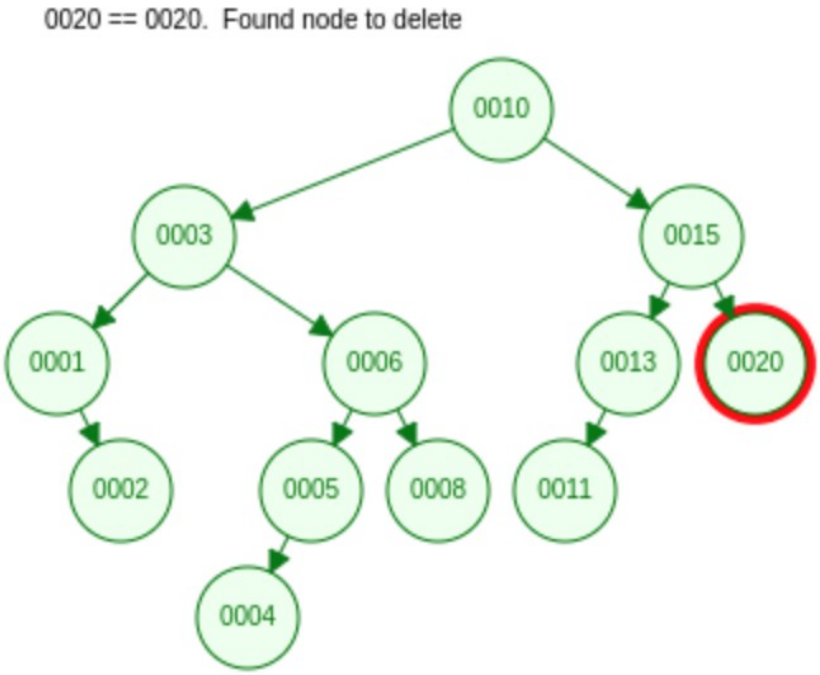
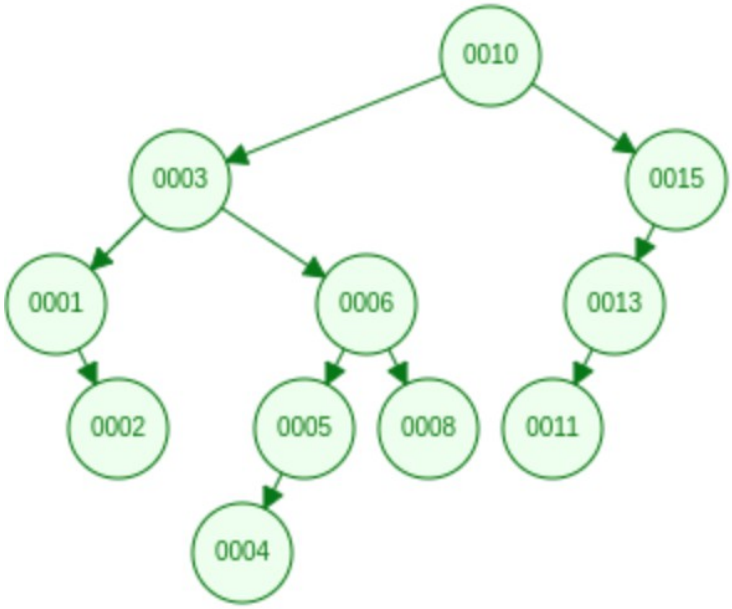
110753201 資科碩一 曹昱維

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。
   A) 刪除 20
      I. 透過 binary search 找到 20

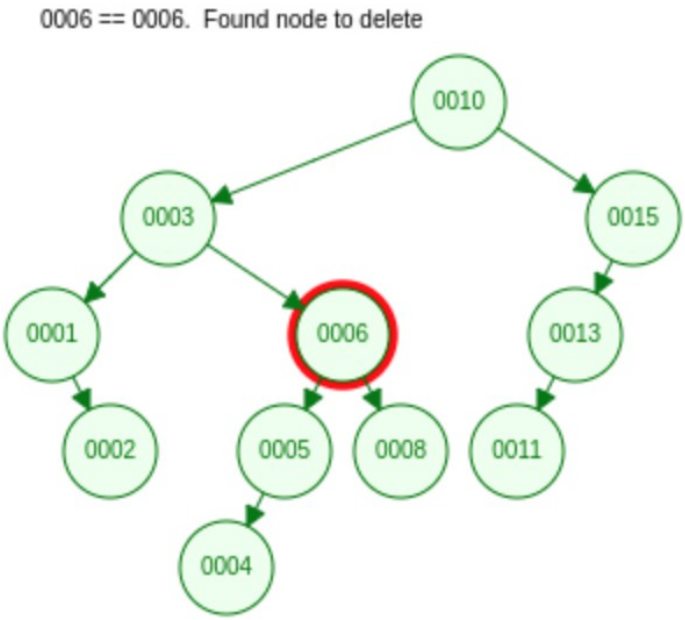



      II. 因為 20 是 leaf node，所以直接刪除即可

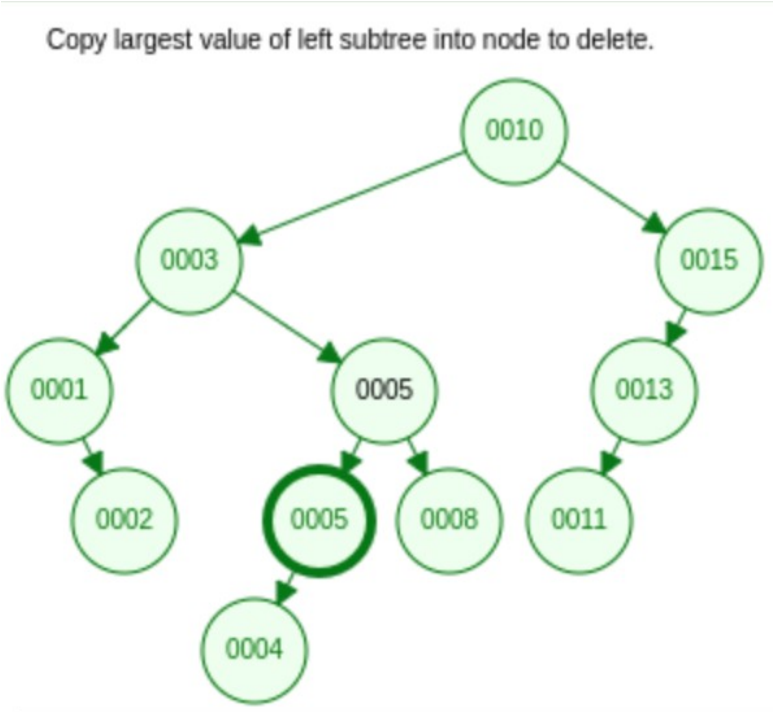


B) 接著刪除 6

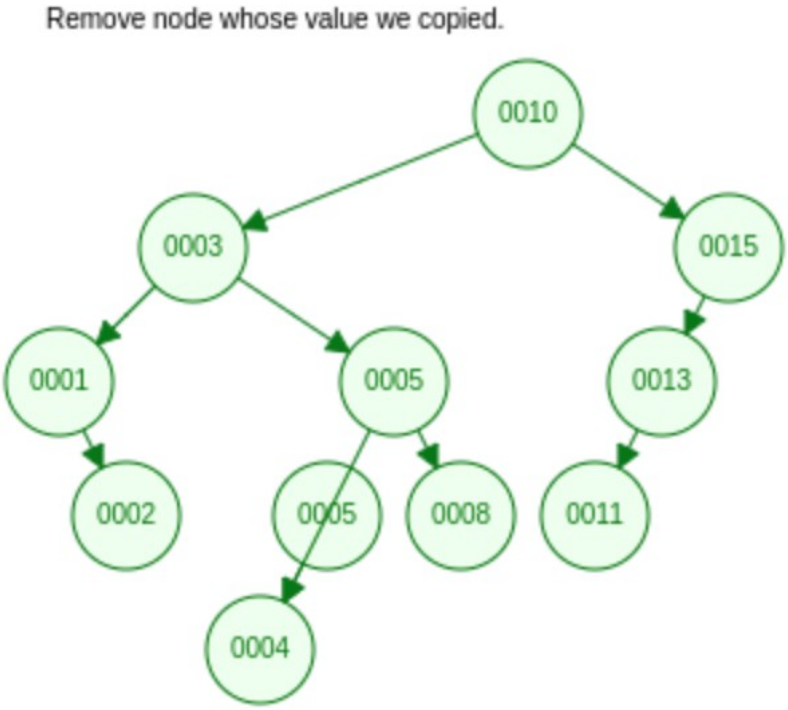
I. 透過 binary search 找到 6



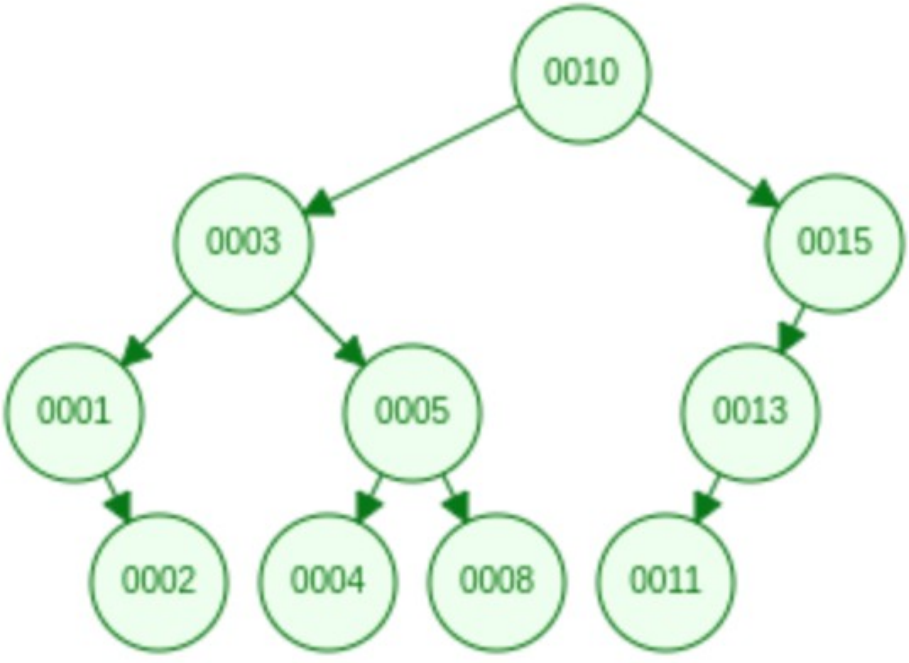

II. 因為 6 有兩個 children，可以使用左子樹的最大節點或者右子樹的最小節點來取代



III. 取代完之後，重新連接

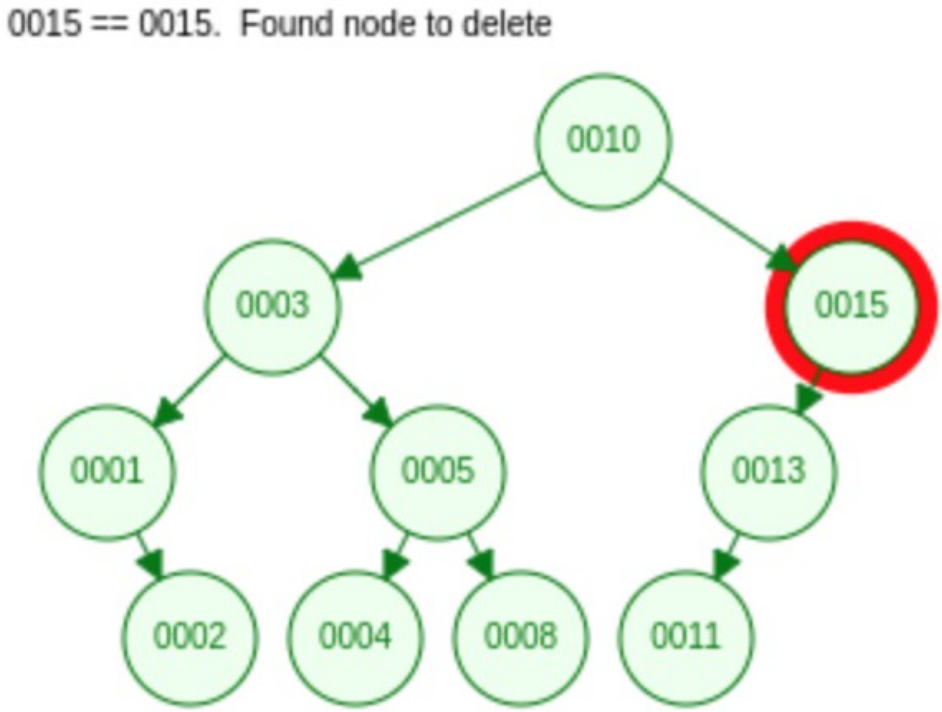



IV. 刪除多餘資料

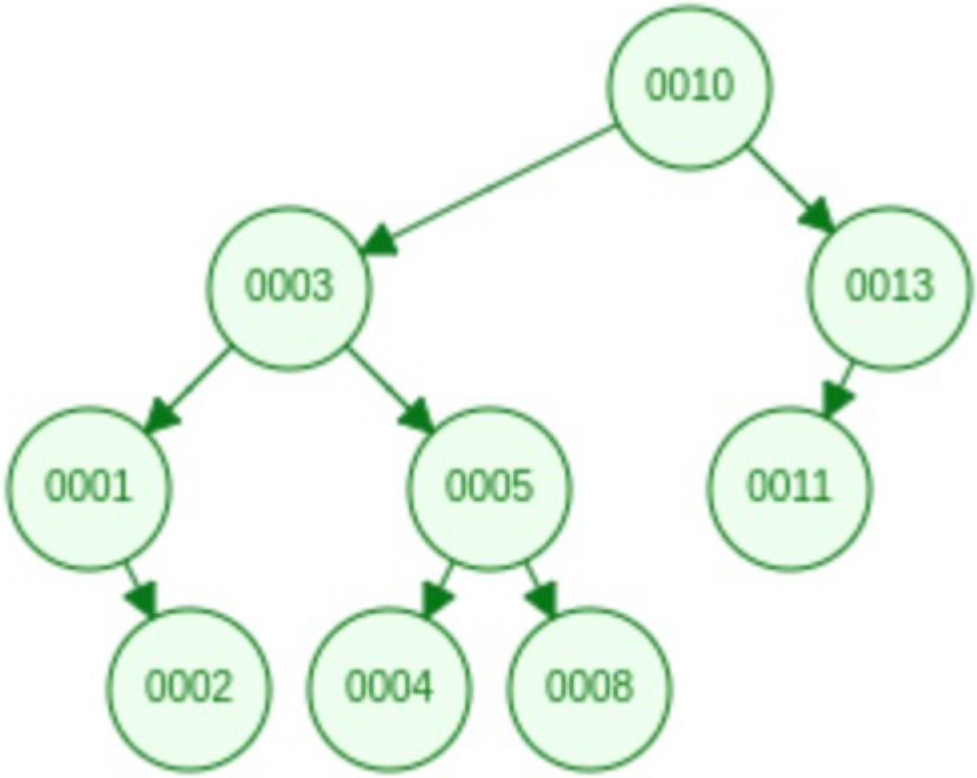


C) 接著刪除 15

I. 透過 binary search 找到 15



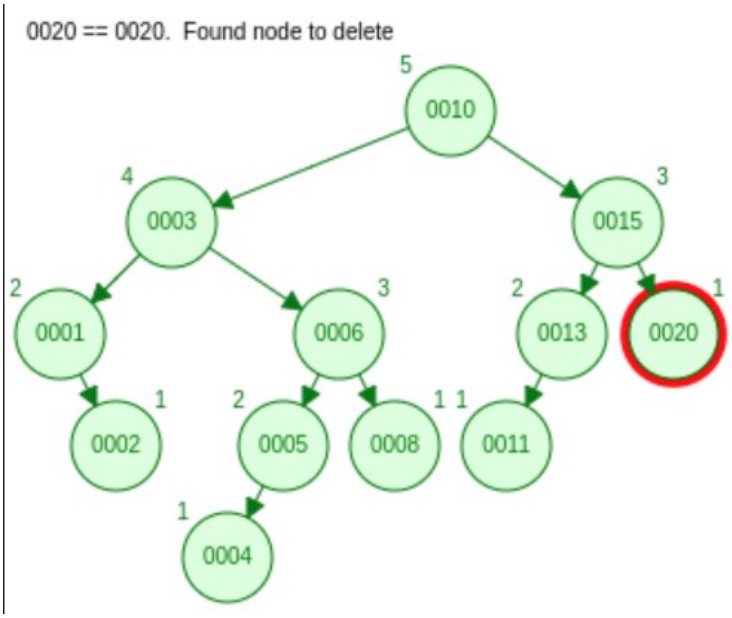

II. 因為 15 只有一個 children，所以調整 children 的 parent，然後刪除 15 即可



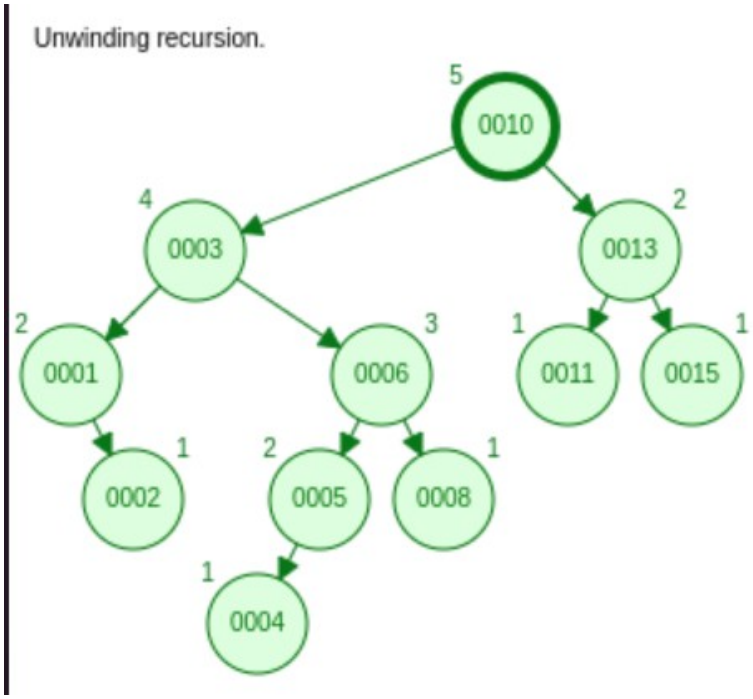
2. 請以圖一為例，說明 AVL Tree 如何先後 Delete 20, 6, 15 (包括必要時 Rebalance 的過程)，請敘述過程

A)刪除 20

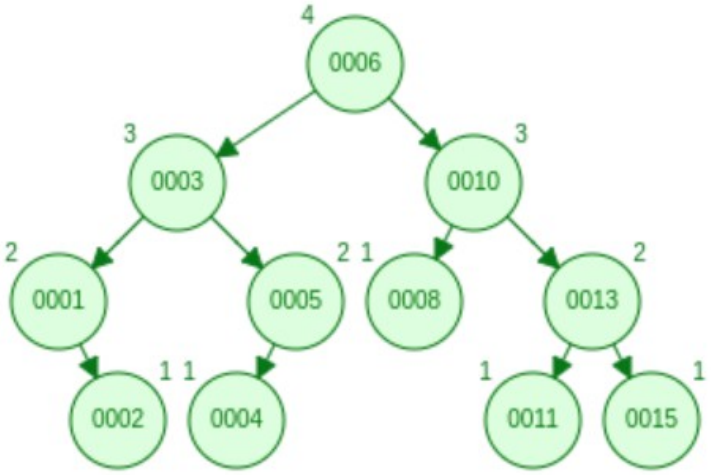
I. 透過 binary search 找到 20，因為 20 是 leaf node，所以直接刪除即可




II. 將 20 刪除之後，15 的 balance factor ＝2，以及 13 的 balance factor ＝1，所以要對 15 與 13 執行 LL rotation

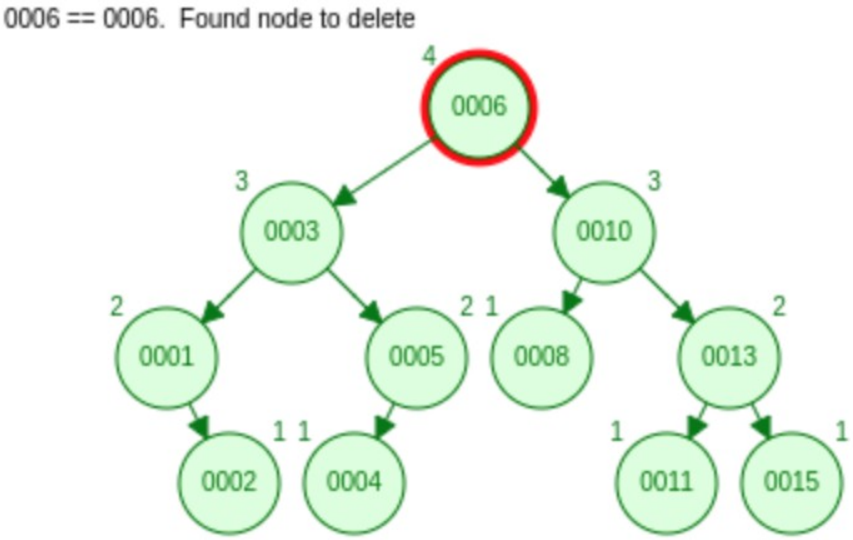


III. 對 15 及 13 執行 LL rotation 之後，10 的 balance factor ＝ 2，以及 3 的 balance factor ＝-1，所以要對 10 與 3 執行 LR rotation

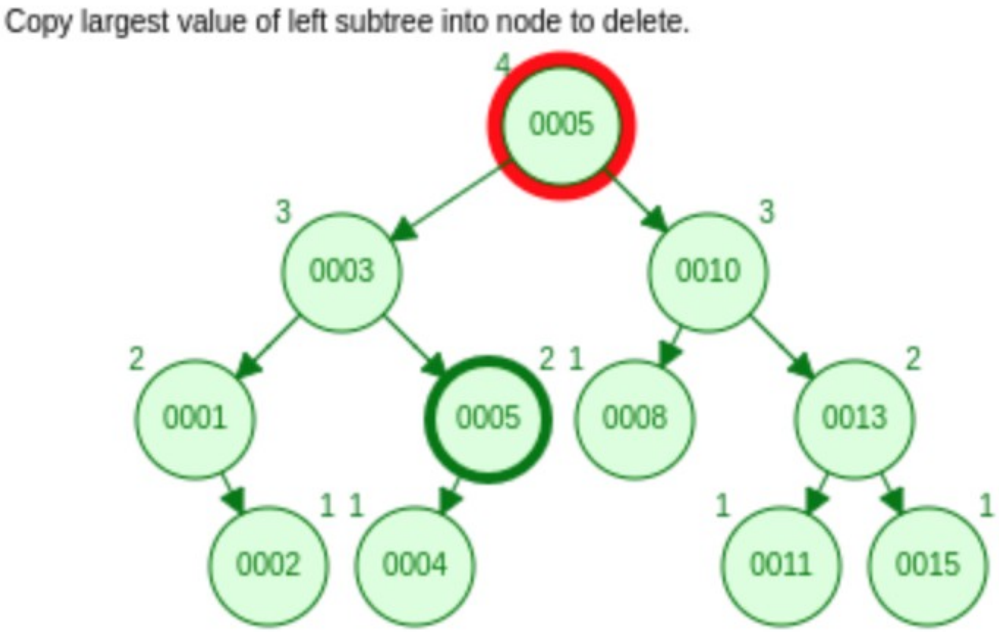



B) 接著刪除 6

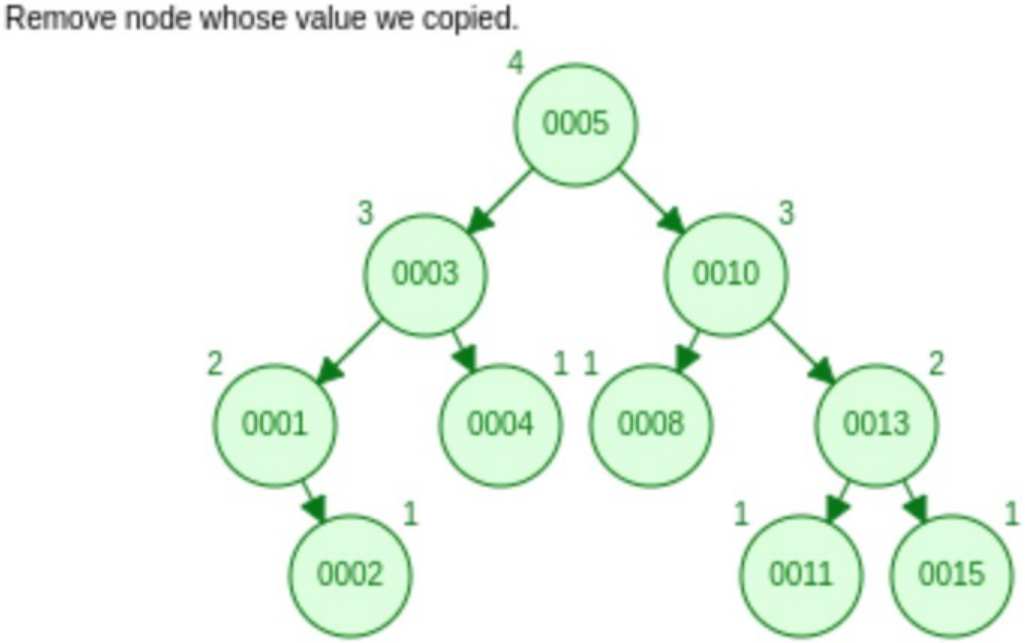
I. 透過 binary search 找到 6



II. 由於 6 有兩個 children，所以可以使用左子樹的最大節點或者右子樹的最小節點來取代，取代完之後，重新連接，並刪除多餘資料

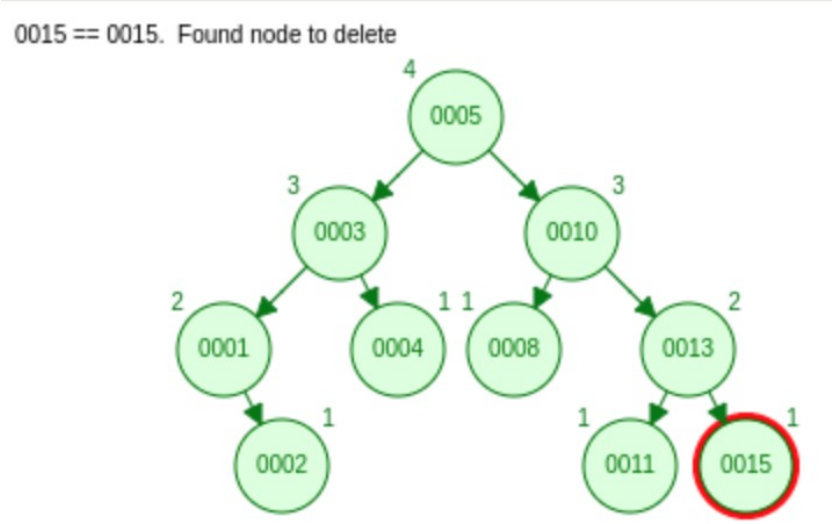



III.檢查 5 與 3 的 balance factor，確認不需要 rebalance

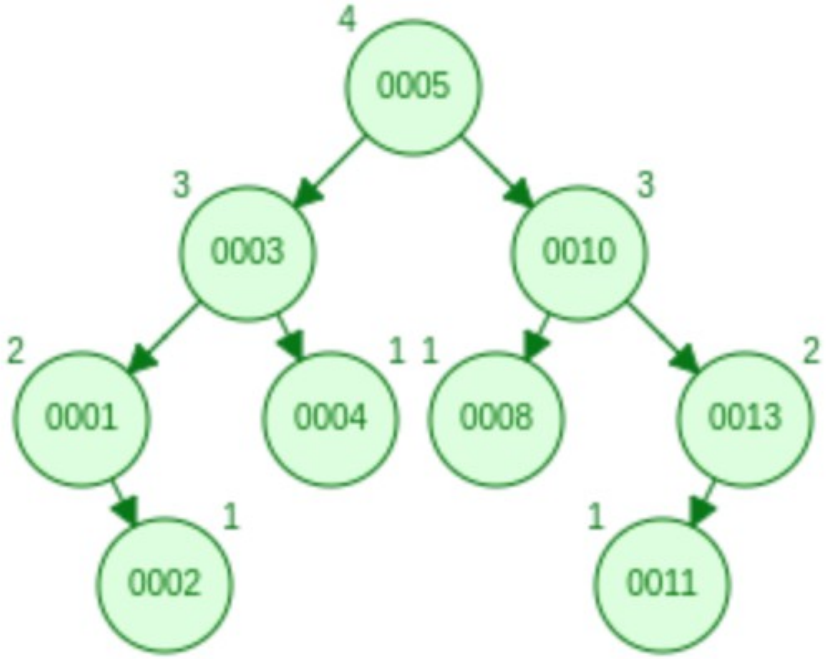


C) 接著刪除 15

I. 透過 binary search 找到 15,因為 15 是 leaf node，所以直接刪除即可




II. 將 15 刪除之後，10 的 balance factor ＝1，以及 13 的 balance factor ＝1，不需要 rebalance



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

```
#ifndef _Tree_H
struct Tree_node;
typedef struct Tree_node *Position;
typedef struct Tree_node *AVL_tree;
AVL_tree delete( ElementType X, AVL_tree T );
#endif

struct Tree_node
{
    ElementType key;
    AVL_tree parent;
    AVL_tree left;
    AVL_tree right;
    int height;
};


// Get height of the tree
int height(AVL_tree t)
{
    if (t == NULL)
        return 0;
    return t->height;
}

// Get Balance factor of node t
int getBalance(AVL_tree t)
{
    if (t == NULL)
        return 0;
    return height(t->left) - height(t->right);
}


```

```c
/*
          y                                x
         / \     Right Rotation           /  \
        x   T3   - - - - - - - >        T1   y
       / \       < - - - - - - -            / \
      T1  T2     Left Rotation            T2  T3
*/
AVL_tree rightRotate(AVL_tree y)
{
    AVL_tree x = y->left;
    AVL_tree T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = max(height(y->left),
                    height(y->right)) + 1;
    x->height = max(height(x->left),
                    height(x->right)) + 1;

    // Return new root
    return x;
}

AVL_tree leftRotate(AVL_tree x)
{
    AVL_tree y = x->right;
    AVL_tree T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    // Update heights
    x->height = max(height(x->left),
                    height(x->right)) + 1;
    y->height = max(height(y->left),
                    height(y->right)) + 1;

    // Return new root
    return y;
}

// binary search node
Position find( ElementType x, AVL_tree t )
{
    if( t == NULL)
        return NULL;
    if( x < t->key )
        return find( x, t->left );
    if( x > t->key )
        return find( x, t->right );
    else
        return t;
}
```

```
AVL_tree delete( ElementType x, AVL_tree t ){
    // binary search for x
    Position p = find(x);
    if (p == NULL){
        printf("Can't find data");
        return NULL;
    }

    // BST deletion
    if ( x < t->key )
        delete( x, t->left );
    else if ( x > t->key )
        delete( x, t->right );
    else if (t->right!=NULL && t->left!=NULL){       // for two children node
        Find the minimun m of T's right subtree ;
        Replace t.key with m;
        free(m);
    }
    else {                                          // for sigle children node or leaf node
        Adjusts t's parent a pointer to bypass the node
        free(t);
    }


    // update height and balance
    t->height = 1 + max(height(t->left),height(t->right));
    int balance = getBalance(t);

    // deletion might cause arbitrary frequency rotations
    while (balance>1 || balance<-1){

        // Left Left Case
        if (balance > 1 && getBalance(t->left)==1)
            return rightRotate(t);

        // Right Right Case
        if (balance < -1 && getBalance(t->right)==-1)
            return leftRotate(t);

        // Left Right Case
        if (balance > 1 && getBalance(t->left)==-1)
        {
            t->left = leftRotate(t->left);
            return rightRotate(t);
        }

        // Right Left Case
        if (balance < -1 && getBalance(t->right)==1)
        {
            t->right = rightRotate(t->right);
            return leftRotate(t);
        }

        // update height and balance
        t->height = 1 + max(height(t->left),height(t->right));
        int balance = getBalance(t);
    }
    return t;
}
```

4. 為什麼 AVL Tree Deletion 的 Rebalance 與 Insertion 有差異?
    - Deletion 需要另外考量刪除的節點是在樹上的那一個位置，而 Insertion 時資料都是在葉節點上
    - Deletion 有可能造成不固定次數的 rotation，而 Insertion 時最多只需要兩次 rotation