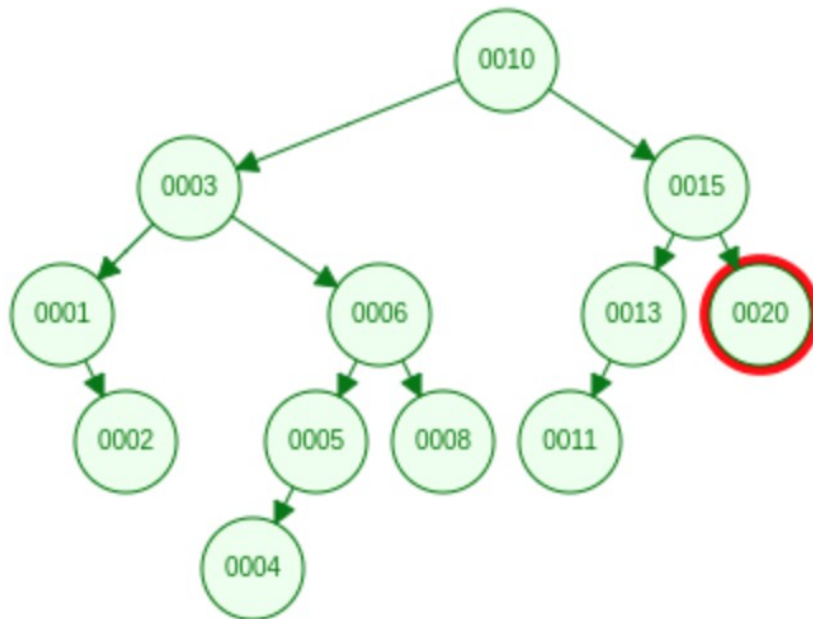


1. 請以圖一為例，說明 Binary Search Tree 如何先後 Delete 20, 6, 15，請敘述過程。

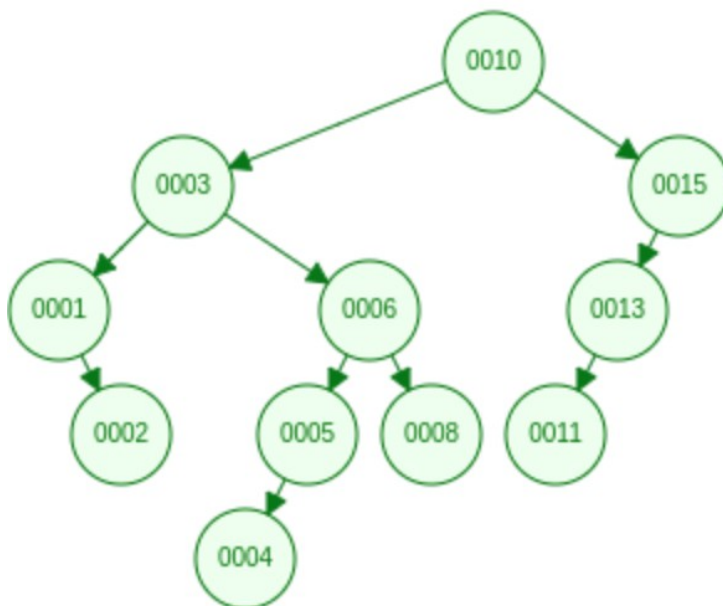
A) 刪除 20

I. 透過 binary search 找到 20

0020 == 0020. Found node to delete



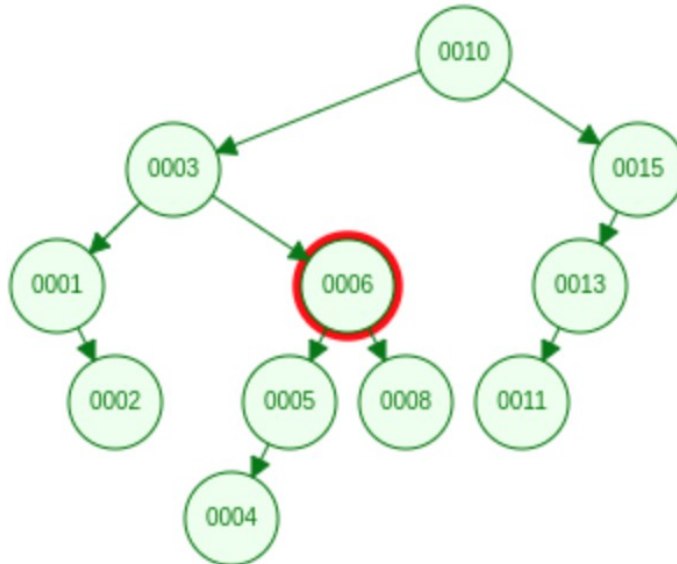
II. 因為 20 是 leaf node，所以直接刪除即可



## B) 接著刪除 6

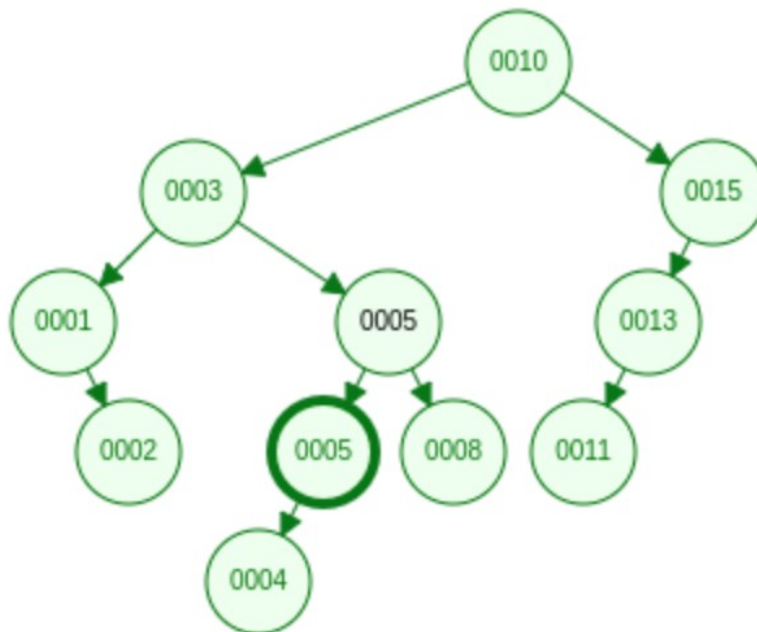
### I. 透過 binary search 找到 6

0006 == 0006. Found node to delete



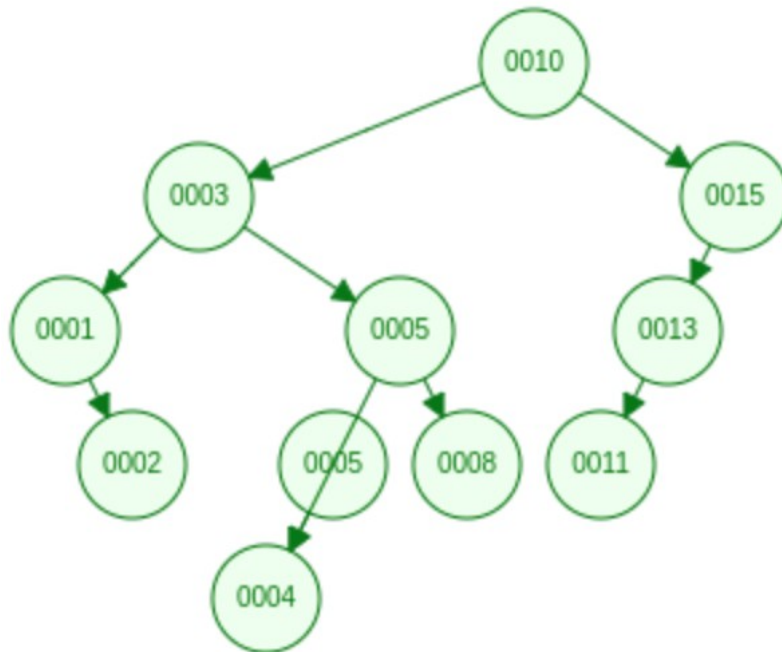
### II. 因為 6 有兩個 children，可以使用左子樹的最大節點或者右子樹的最小節點來取代

Copy largest value of left subtree into node to delete.

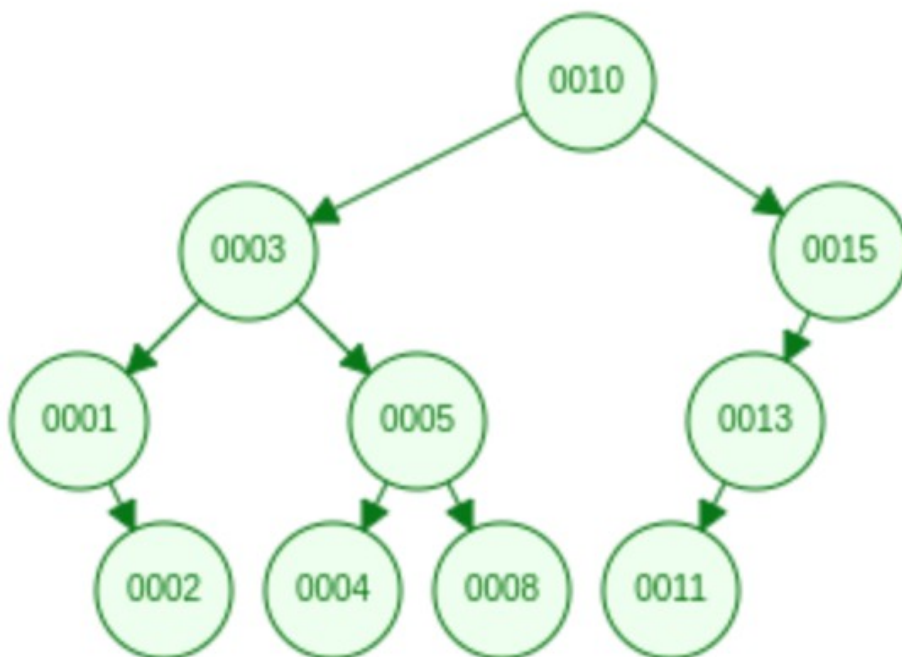


### III. 取代完之後，重新連接

Remove node whose value we copied.



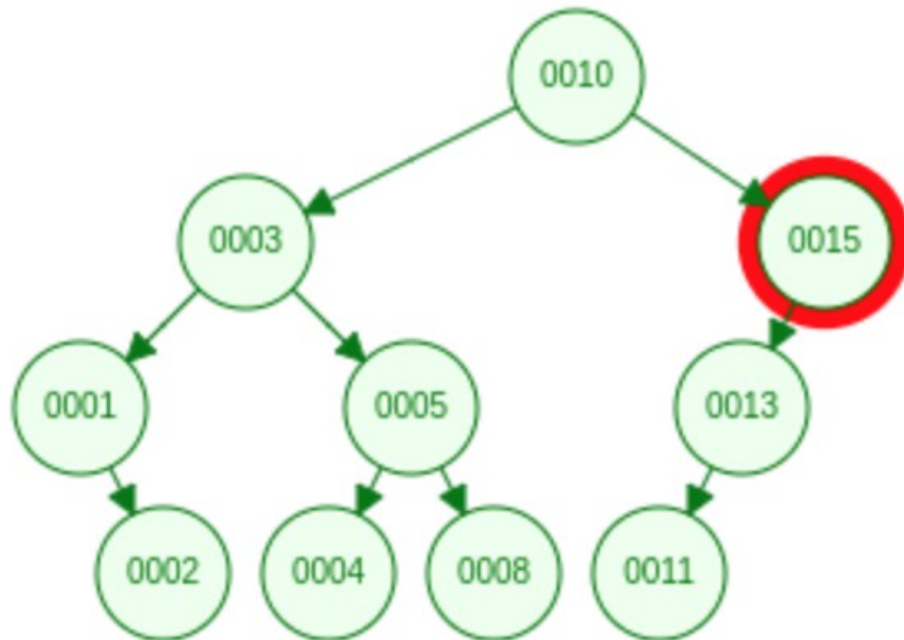
### IV. 刪除多餘資料



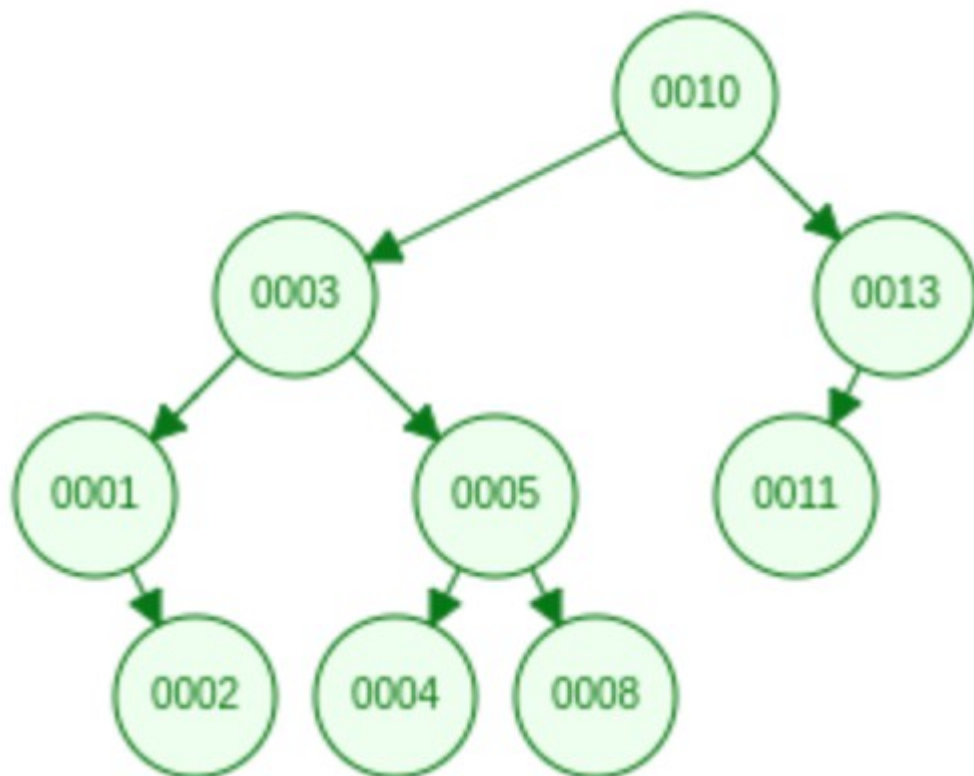
C) 接著刪除 15

I. 透過 binary search 找到 15

0015 == 0015. Found node to delete



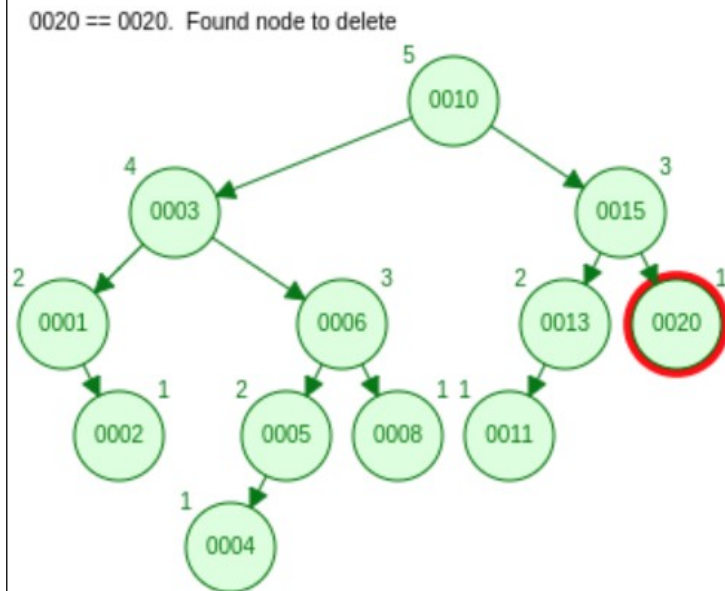
II. 因為 15 只有一個 children，所以調整 children 的 parent，然後刪除 15 即可



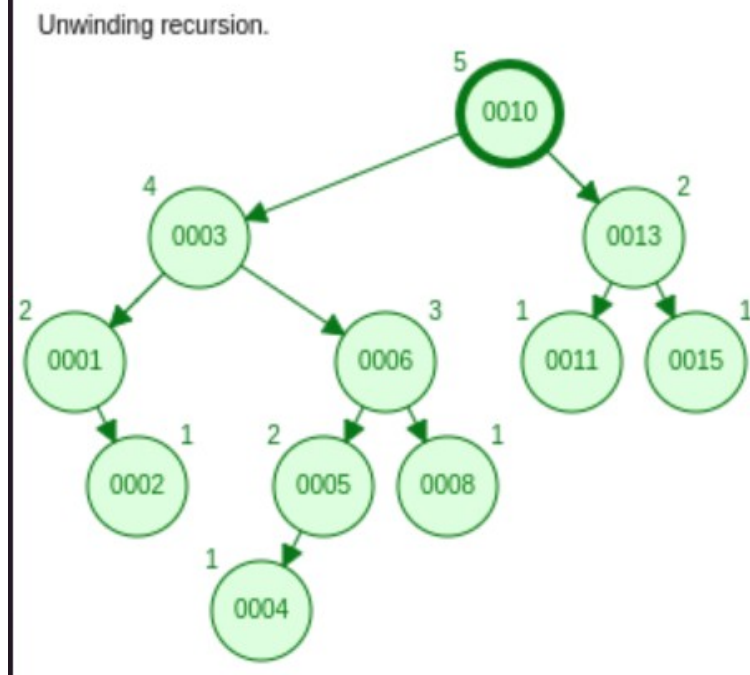
2. 請以圖一為例，說明 AVL Tree 如何先後 Delete 20, 6, 15 (包括必要時 Rebalance 的過程)，請敘述過程

A) 刪除 20

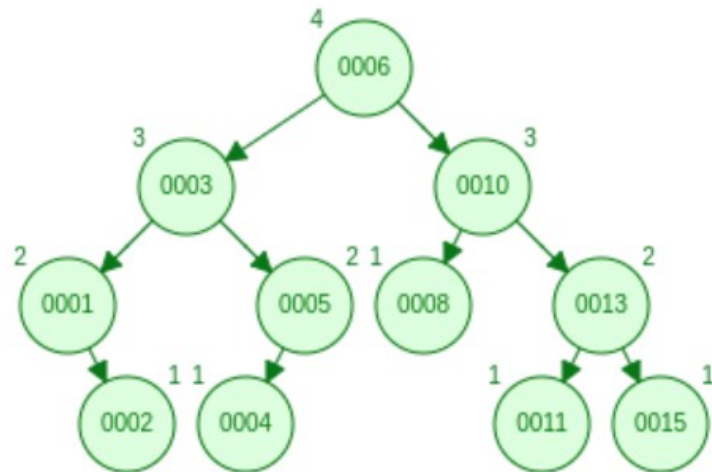
I. 透過 binary search 找到 20，因為 20 是 leaf node，所以直接刪除即可



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



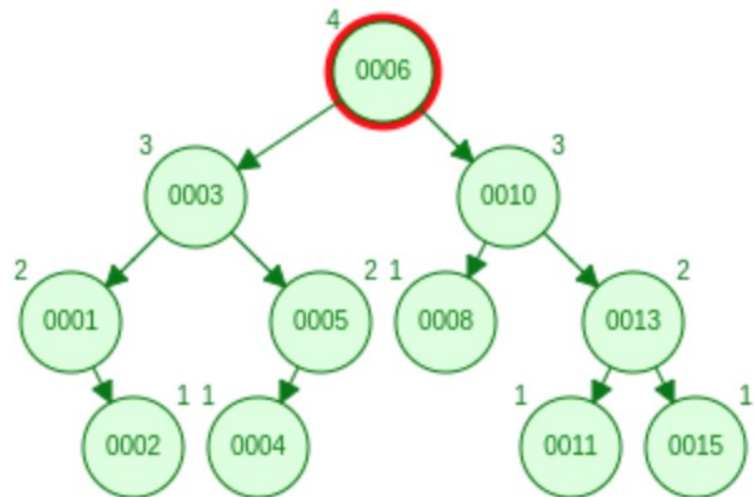
III. 對 15 及 13 執行 LL rotation 之後, 10 的 balance factor = 2, 以及 3 的 balance factor = -1, 所以要對 10 與 3 執行 LR rotation



B) 接著刪除 6

I. 透過 binary search 找到 6

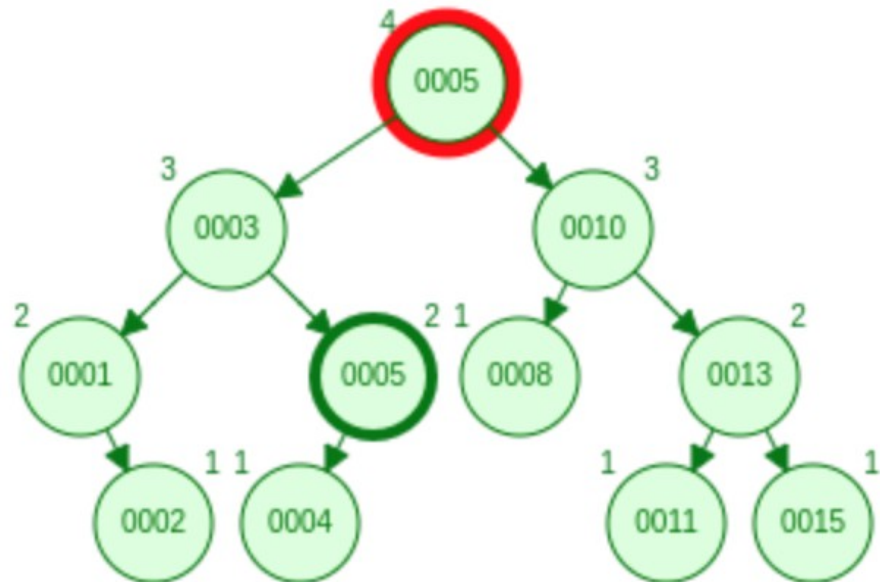
0006 == 0006. Found node to delete





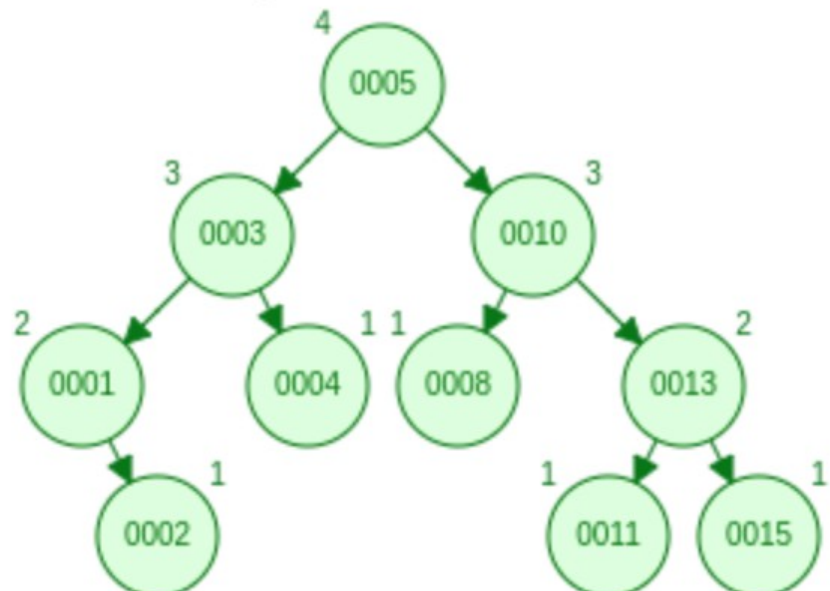
- II. 由於 6 有兩個 children，所以可以使用左子樹的最大節點或者右子樹的最小節點來取代，取代完之後，重新連接，並刪除多餘資料

Copy largest value of left subtree into node to delete.



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

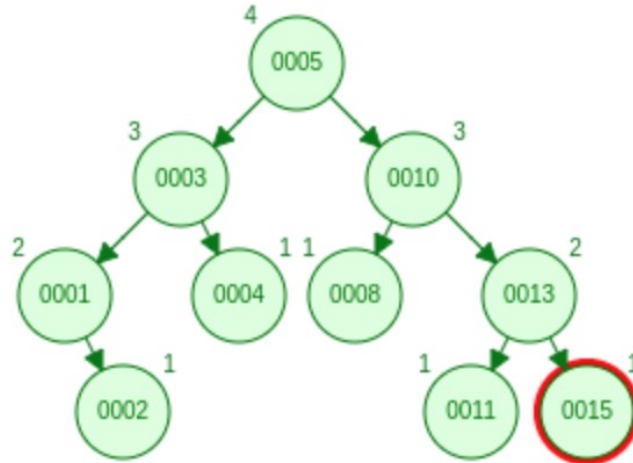
Remove node whose value we copied.



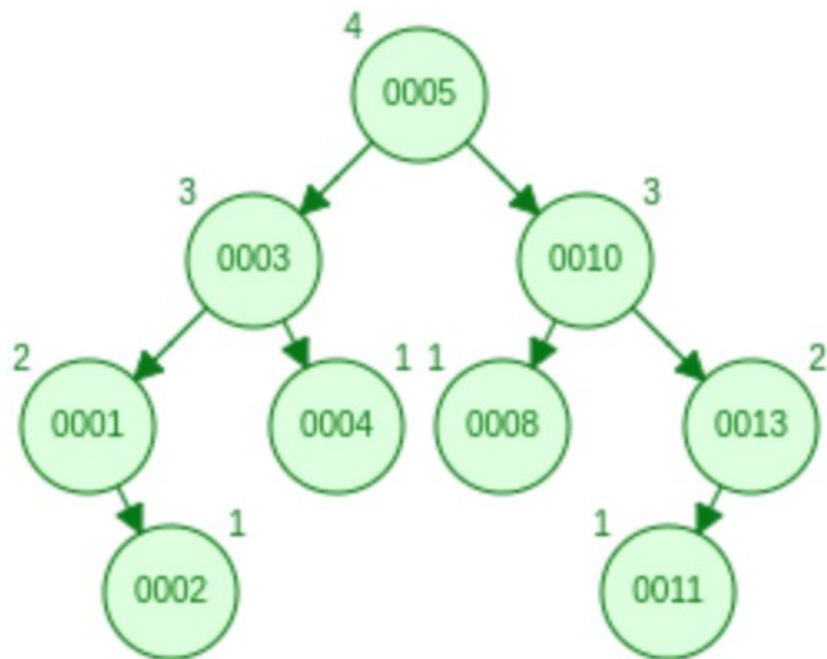
C) 接著刪除 15

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

0015 == 0015. Found node to delete



- 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)。

```
1  #ifndef _Tree_H
2  struct Tree_node;
3  typedef struct Tree_node *Position;
4  typedef struct Tree_node *AVL_tree;
5  AVL_tree delete( ElementType X, AVL_tree T );
6  #endif
7
8  struct Tree_node
9  {
10     ElementType key;
11     AVL_tree parent;
12     AVL_tree left;
13     AVL_tree right;
14     int height;
15 };
16
17
18 // Get height of the tree
19 int height(AVL_tree t)
20 {
21     if (t == NULL)
22         return 0;
23     return t->height;
24 }
25
26 // Get Balance factor of node t
27 int getBalance(AVL_tree t)
28 {
29     if (t == NULL)
30         return 0;
31     return height(t->left) - height(t->right);
32 }
33
34
```

```

35  /*
36      |       |       |       y
37      |       |       |      / \   Right Rotation
38      |       |       |     x   T3  - - - - - - - - >
39      |       |       |    / \   < - - - - - - - -
40      |       |       |   T1  T2   Left Rotation
41  */
42  AVL_tree rightRotate(AVL_tree y)
43  {
44      AVL_tree x = y->left;
45      AVL_tree T2 = x->right;
46
47      // Perform rotation
48      x->right = y;
49      y->left = T2;
50
51      // Update heights
52      y->height = max(height(y->left),
53                    height(y->right)) + 1;
54      x->height = max(height(x->left),
55                    height(x->right)) + 1;
56
57      // Return new root
58      return x;
59  }
60
61  AVL_tree leftRotate(AVL_tree x)
62  {
63      AVL_tree y = x->right;
64      AVL_tree T2 = y->left;
65
66      // Perform rotation
67      y->left = x;
68      x->right = T2;
69
70      // Update heights
71      x->height = max(height(x->left),
72                    height(x->right)) + 1;
73      y->height = max(height(y->left),
74                    height(y->right)) + 1;
75
76      // Return new root
77      return y;
78  }
79
80  // binary search node
81  Position find( ElementType x, AVL_tree t )
82  {
83      if( t == NULL )
84          return NULL;
85      if( x < t->key )
86          return find( x, t->left );
87      if( x > t->key )
88          return find( x, t->right );
89      else
90          return t;
91  }

```

```

95  AVL_tree delete( ElementType x, AVL_tree t ){
96      // binary search for x
97      Position p = find(x);
98      if (p == NULL){
99          printf("Can't find data");
100         return NULL;
101     }
102
103     // BST deletion
104     if ( x < t->key )
105         delete( x, t->left );
106     else if ( x > t->key )
107         delete( x, t->right );
108     else if (t->right!=NULL && t->left!=NULL){          // for two children node
109         Find the minimum m of T's right subtree ;
110         Replace t.key with m;
111         free(m);
112     }
113     else {                                              // for single children node or leaf node
114         Adjusts t's parent a pointer to bypass the node
115         free(t);
116     }
117
118
119     // update height and balance
120     t->height = 1 + max(height(t->left),height(t->right));
121     int balance = getBalance(t);
122
123     // deletion might cause arbitrary frequency rotations
124     while (balance>1 || balance<-1){
125
126         // Left Left Case
127         if (balance > 1 && getBalance(t->left)==1)
128             return rightRotate(t);
129
130         // Right Right Case
131         if (balance < -1 && getBalance(t->right)==-1)
132             return leftRotate(t);
133
134         // Left Right Case
135         if (balance > 1 && getBalance(t->left)==-1)
136         {
137             t->left = leftRotate(t->left);
138             return rightRotate(t);
139         }
140
141         // Right Left Case
142         if (balance < -1 && getBalance(t->right)==1)
143         {
144             t->right = rightRotate(t->right);
145             return leftRotate(t);
146         }
147
148         // update height and balance
149         t->height = 1 + max(height(t->left),height(t->right));
150         int balance = getBalance(t);
151     }
152     return t;
153 }

```

#### 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