AVL Tree Deletion

109703040 資科一 洪晙宸

[演算法說明]

輸入為 AVLTree 及 欲刪除值 key

刪除步驟：

1. 尋找 key 是否在 tree 裡，循著 binary tree 左右搜尋，將 subtree 作為根(root)進行遞迴。若找不到對應的 key 便不進行動作。使用遞迴是因為在完成刪除後一步一步 return 回頂點時仍要持續保持 AVL 特性，因此要對每個子節點進行 rebalance。
2. 當找到欲刪除的 key 時，分為三種情況

   (1) 節點的高度差為-1 時(左比右高 1)

   (2) 節點的高度差為 1 時(右比左高 1)

   (3) 節點的高度差為 0

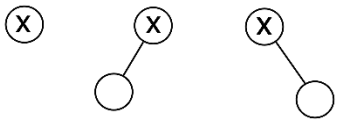
   第 1 種情況中，將左邊 subtree 裡最右邊(max)的數和欲刪除的值交換，繼續向下遞迴尋找換下去的數字。原因有：

   (1) 已知左邊比右邊高，若向左取出某值替代刪除的節點，右方 subtree 和左方 subtree 的高度差將會到 2，違反 AVL 特性。

   (2) 因為節點左邊的數值均比該節點小，因此找左邊最大的數字做為新的根結點可以滿足

狀況 2 同理，尋找右邊 subtree 裡最左邊(min)的數。

狀況 3 則無所謂，本演算法歸類在狀況 2 處理。

3. 當第二次找到目標時，該點一定為 

x 為欲刪除值，0 為原本存在於該點的節點，因為替換時已經保證是子樹的最大值或最小值，故僅可能為上面三種情況的其中一種。

(1) 無子樹：直接將該點捨去

(2) 有一邊子樹：將根節點的值換成子節點的值

同時將該點高度重新定義為 0，結束函式。

4. 因為到達底層的時候是一層一層的傳下去，所以結束每階段函式工作時進行 AVL 判斷：

(1) 從左方返回：此時右方的子樹確定保持 AVL 特性，先檢查右邊子樹和左邊子樹差是否等於 2，若是，檢查左子樹中的子樹高度差，若左邊高度大於右邊，進行 RL rotation，反之進行 RR rotation

(2) 從右方返回：此時左方的子樹確定保持 AVL 特性，先檢查左邊子樹和右邊子樹差是否等於 2，若是，檢查右子樹中的子樹高度差，若左邊高度大於右邊，進行 LL rotation，反之進行 LR rotation

一路傳到頂層，回傳刪除後的樹，結束函式。

[Pseudo code]

```
/*
data structure:

node:{
    key: int,
    height: int,
    left: node,
    rightL node
}

*/

function max_key_in_subtree(subtree){  //尋找該節點中最大的子節點
    while(subtree.right != null){
        subtree = subtree.right
    }
    return subtree;
}
function min_key_in_subtree(subtree){  //尋找該節點中最小的子節點
    while(subtree.left != null){
        subtree = subtree.left
    }
    return subtree;
}
function height(tree){  //尋找節點高度
    if(tree){
        return tree.height
    }
    return 0;
}
function L_L(tree){  //LL旋轉
    var tmp = tree.left;
    tree.left = tmp.right
    tmp.right = tree
    tree.height = max(height(tree.left), height(tree.right)) + 1
    tmp.height = max(height(tmp.left), height(tmp.right)) + 1
    return tmp
}
function R_R(tree){  //RR旋轉
    var tmp = tree.right;
    tree.right = tmp.left
    tmp.left = tree
    tree.height = max(height(tree.left), height(tree.right)) + 1
    tmp.height = max(height(tmp.left), height(tmp.right)) + 1
    return tmp
```

```
}
function R_L(tree){  //RL旋轉
    tree.right = L_L(tree.right)
    return R_R(tree)
}
function L_R(tree){  //LR旋轉
    tree.left = R_R(tree.left)
    return L_L(tree)
}

function delete_node(tree, key){
    if(key < tree.key){
        //欲刪除的值小於節點的值(向左子樹遞迴)
        tree = delete_node(tree.left, key)
        //在遞迴結束後進行avl平衡
        if(height(tree.right) - height(tree.left) == 2){
            if(height(tree.right.left) > height(tree.right.right)){
                tree = R_L(tree)
            }else{
                tree = R_R(tree)
            }
        }
    }else if(key > tree.key){
        //欲刪除的值大於節點的值(向右子樹遞迴)
        tree = delete_node(tree.right, key)
        //在遞迴結束後進行avl平衡
        if(height(tree.left) - height(tree.right) == 2){
            if(height(tree.left.left) > height(tree.left.right)){
                tree = L_L(tree)
            }else{
                tree = L_R(tree)
            }
        }
    }else if(key == tree.key){
        //此節點恰為要刪除的節點
        //第一次遇到時，先假設該節點左右子樹都不為空，開始進行替換
        if(tree.left && tree,right){
            if(tree.left.height > tree.right.height){
                var max_key = max_key_in_subtree(left)
                swap(max_key.key, tree.left.key)
                //目標刪除節點轉至max_key，繼續往下
                delete_node(tree.left, max_key.key)
            }else{
                var min_key = min_key_in_subtree(right)
                swap(min_key.key, tree.right.key)
                //目標刪除節點轉至min_key，繼續往下
```

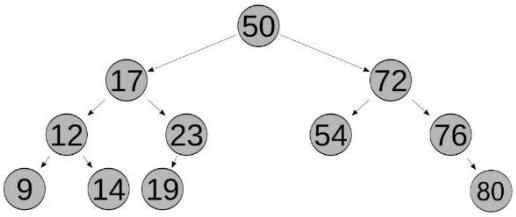
```
82          if(tree.left && tree,right){
83              if(tree.left.height > tree.right.height){
84                  var max_key = max_key_in_subtree(left)
85                  swap(max_key.key, tree.left.key)
86                  //目標刪除節點轉至max_key，繼續往下
87                  delete_node(tree.left, max_key.key)
88              }else{
89                  var min_key = min_key_in_subtree(right)
90                  swap(min_key.key, tree.right.key)
91                  //目標刪除節點轉至min_key，繼續往下
92              }
93          }
94          //第二次遇到或是目標恰在底層，這時他已經位於底層(必定有一邊子樹為空)
95          else{
96              if(tree.left){
97                  tree = tree.left
98              }else if(tree.right){
99                  tree = tree.right
100             }else{
101                 tree = null
102             }
103             tree.height = 0
104         }
105     }
106     return tree
107 }
```

source code in github :

https://gist.github.com/andyjjrt/0f3c90733cbf6566d4c1190ea3bcd2f0

[舉例說明]

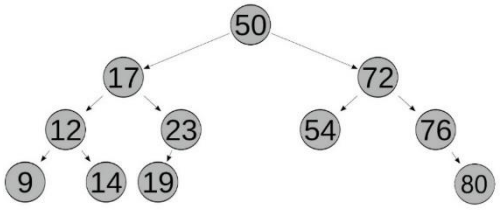
1. 刪除 9 (當刪除點為端點時)：



   1) 從樹頂(根)50 進入。
   2) 9 < 50，向左走進入第二層。
   3) 9 < 17，向左走進入第三層。
   4) 9 < 12，向左走進入第四層。
   5) 9 = 9，且 9 沒有任何子節點，因此直接刪除 9，結束第四層。
   6) 12 的高度差為 1-0 = 1，滿足 AVL，結束第三層。
   7) 17 的高度差為 1-1 = 0，滿足 AVL，結束第二層。
   8) 50 的高度差為 2-2 = 0，滿足 AVL，結束函式，

      傳回刪除 9 後的 AVL tree。

2. 刪除 17 (當欲刪除的點有子節點)：



   1) 從樹頂(根)50 進入。
   2) 17 < 50，向左走進入第二層。
   3) 17 = 17，且有左右節點

      ⇨ 左右子樹高度相同，為了維持 AVL 特性，向右邊子樹尋找最小的值

      為 19

⇨ 將 19 和 17 值對調，繼續向下尋找 17 進行刪除，向右走進入第三層。

4) 17 < 23，向左走進入第四層。

5) 17 = 17，且 17 沒有任何子節點，因此直接刪除 17，結束第四層。

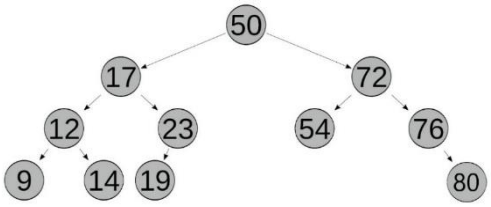
6) 23 的高度差為 0-0 = 0，滿足 AVL，結束第三層。

7) 19 的高度差為 0-1 = -1，滿足 AVL，結束第二層。

8) 50 的高度差為 2-2 = 0，滿足 AVL，結束函式，

傳回刪除 17 後的 AVL tree。

3. 刪除 54 (刪除後不滿足 AVL 特性)：



1) 從樹頂(根)50 進入。

2) 54 > 50，向左走進入第二層。

3) 54 < 72，向左走進入第三層。

4) 54 = 54，且 54 沒有任何子節點，因此直接刪除 54，結束第三層。

5) 72 的高度差為 1-(-1) = 2，不滿足 AVL

⇨ 因為右邊比左邊高 2 且右子樹的右邊比左邊高，進行 RR rotation。

⇨ 滿足 AVL，結束第二層。

6) 50 的高度差為 1-2 = -1，滿足 AVL，結束函式，傳回刪除 54 後的 AVL tree。

[時間複雜度]

對於總結點數為 N 的 AVL Tree，因為高度差最多為 1，因此：

Best case：$\log_2 N$

Worst case：$\log_2 N + 1$

因為使用遞迴，尋找到目標值，刪除。

在刪除時會先尋找其子樹的最大/最小值，他的

Best case：0

Worst case：$\log_2 N$

回傳後進行 AVL 檢查。

總和為：

Best case：$\log_2 N$

Worst case：$2\log_2 N$

因此時間複雜度為 O(logN)。