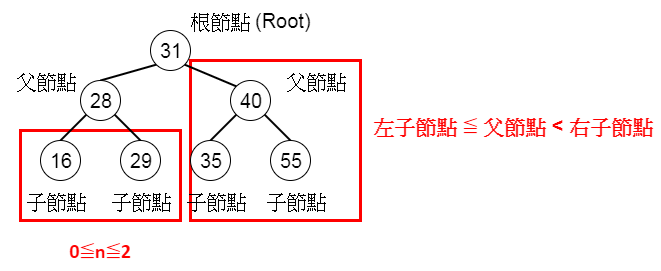
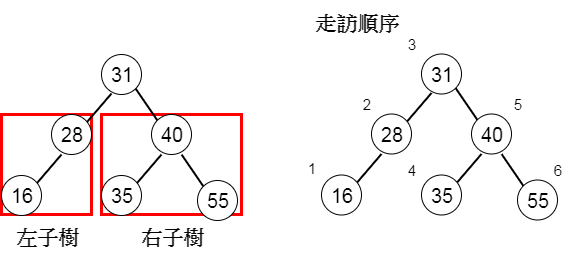
**Binary Search Tree (BST) 原理、流程圖、學習歷程**

1. **BST 原理 & 流程圖**
2. **結構**
   1. 為樹狀結構，由一個或多個節點組合而成的有限集合。
   2. 樹不可以為空，至少有一個特殊的節點稱為「根節點」(Root)。
   3. 根節點之下的節點為 0 ≦ n ≦ 2 個互斥的子集合 T1、T2…Tn，每一個子集合本身也是一棵樹。
   4. 每一個節點都會儲存一個值，稱為「鍵值」。
   5. 每一個節點的鍵值大於等於左子節點的鍵值；每一個節點的鍵值小於右子節點的鍵值。



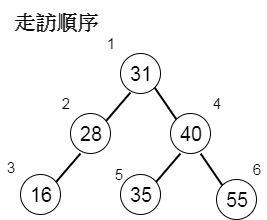
1. **走訪**
2. 中序走訪 (In-order)

走訪順序為: 「左子樹 > 樹根 > 右子樹」



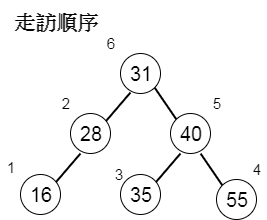
1. 前序走訪 (Pre-order)

走訪順序為: 「樹根 > 左子樹 > 右子樹」



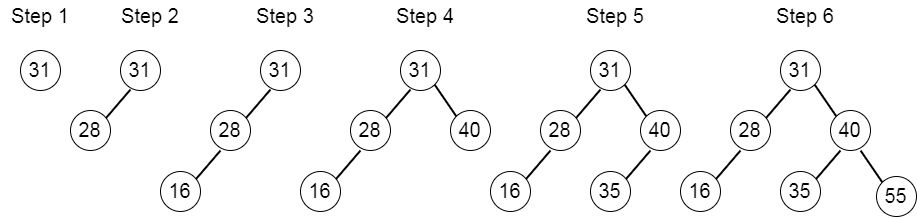
1. 後序走訪 (Post-order)

走訪順序為: 「左子樹 > 右子樹 > 樹根」



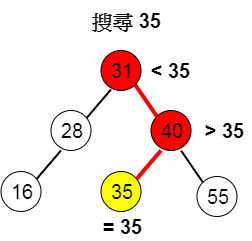
1. **新增**

由於BST有每一個節點的鍵值大於等於左子樹和小於右子樹所有鍵值的特性，因此新增的節點也必須符合這樣的特性。另外，因為根節點之下的節點為 0 ≦ n ≦ 2，所以新增的節點僅能新增在子節點尚未達到二的節點之下。以下將一組資料 31, 28, 16, 40, 35, 55 依照順序新增到一顆二元搜尋樹。輸入的資料相同但順序不同會出現不同的搜尋樹。



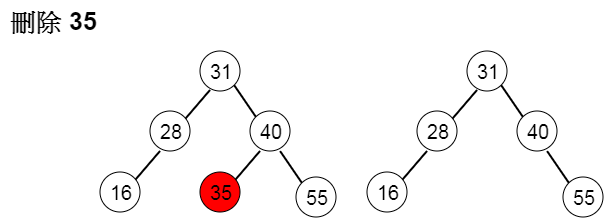
1. 新增31: 先設根節點，31為其鍵值。
2. 新增28: 28比根節點小，所以設為左子節點。
3. 新增16: 16比根節點小，也比28小，所以設為左子數28的左節點。
4. 新增40: 40比根節點大，所以設為右子節點。
5. 新增35: 35比根節點大，比40小，所以設為右子數40的左節點。
6. 新增55: 55比根節點大，也比40大，所以設為右子數40的右節點。
7. **搜尋**

從樹根開始向下搜尋，當欲搜尋的數值小於根節點時就往根節點左方走訪；大於根節點時就往根節點右方走訪。直到找到欲搜尋的數值為止。如已走訪到最後的葉節點但仍未找到數值，則代表欲搜尋的數值並未在二原搜尋樹中。



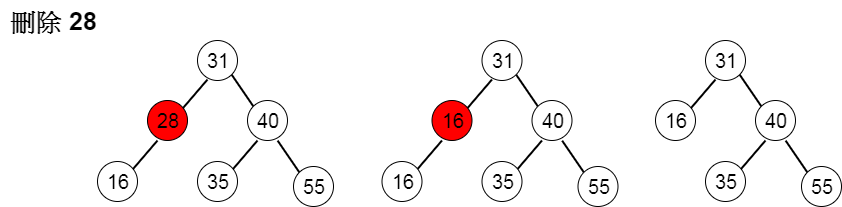
1. **刪除**
2. 欲刪除的節點無任何子節點:

直接將該節點刪除，將其父節點連結該節點的鍊結打斷。



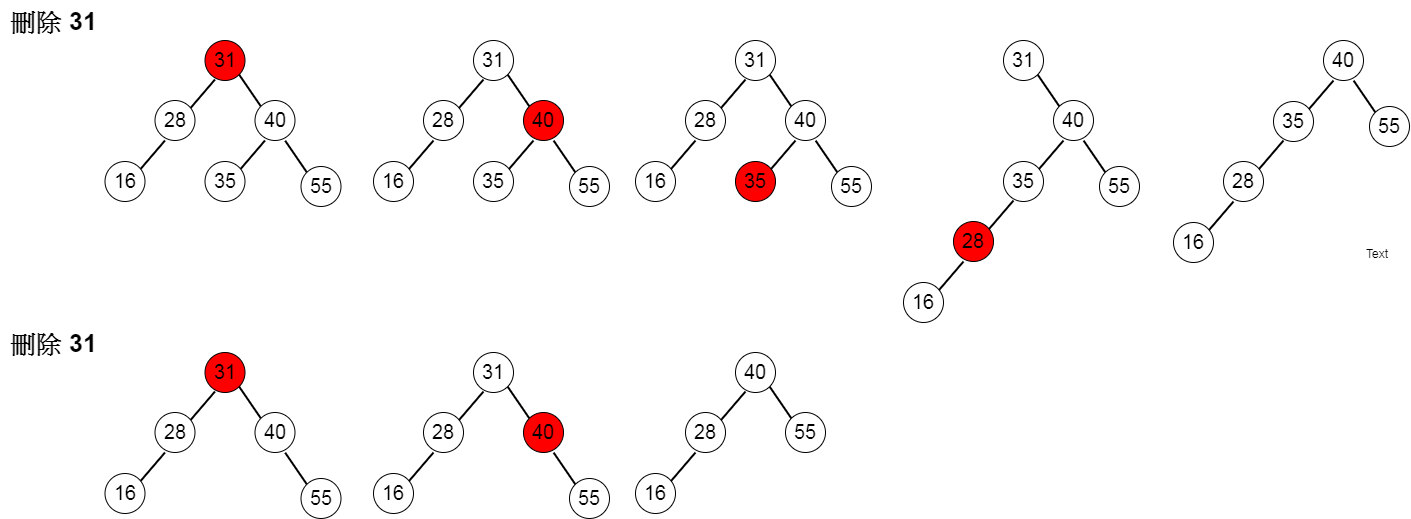
1. 欲刪除的節點僅有一個子節點:

將該節點的子節點的數值取代該節點的數值，再將其子節點刪除，如其子節點有子節點，則將其子節點的子節點接在該節點下。



1. 欲刪除的節點有兩個子節點:

欲刪除的節點有兩個子節點: 先往欲刪除的節點的右邊子節點走訪一格。如果該節點沒有左節點，直接將該節點取代欲刪除的節點，結束。不然就接著不斷往左邊的子節點走訪直到走訪到的節點無左節點為止，然後將欲刪除的節點的左節點接在該節點的左邊，此時欲刪除的節點僅剩右子節點，最後將欲刪除的節點以其柚子節點取代。

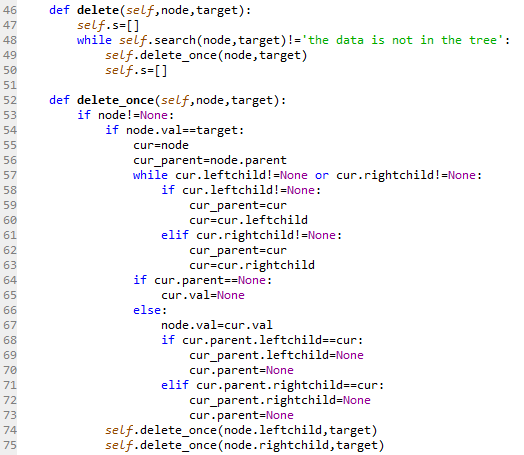
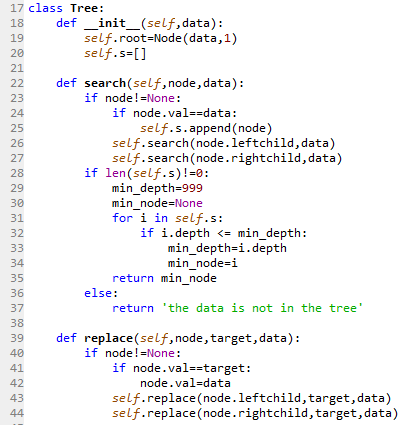
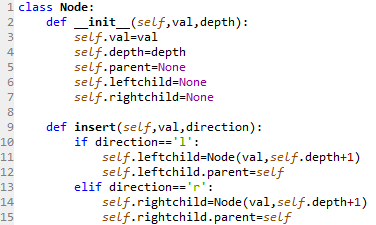


1. **學習歷程**

**(以下程式碼僅Pre-order走訪有參考外部資料，其他均為原創)**

1. **一般二元樹 (非BST) 新增、搜尋、刪除、修改:**

在撰寫BST前有先撰寫一般二元樹，由於結構相對簡單，可以幫助熟悉樹狀結構的演算法。



新增方面，因為不像BST的樹狀結構有規則可循，一般的二元樹的節點可以任意新增。

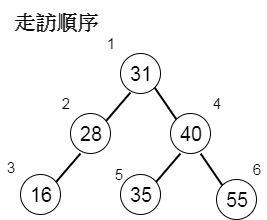
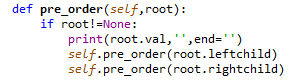
搜尋方面，採用Pre-order走訪，Pre-order的寫法是參考教科書的寫法，用遞迴的方式訪尋每個節點，當發現欲搜尋的值則將該節點append到\_\_init\_\_中的list中，因為搜尋的節點需離根節點最近，所以在Node的物件中有設depth的變數，每當新增一個節點，該節點的depth就會比其父節點的depth多1，因此當append完所數值相同的節點後再return depth最小的節點。

修改的部分，也是用pre-order訪尋，當訪尋到欲修改的節點時則直接修改。

刪除的部分，是採分次刪除的方式，無法一次就刪除所有數值相同的節點。也是用pre-order的方式走訪所有節點，當遇到欲刪除的節點時，則將該節點刪除，其中因為刪除時需呼叫欲刪除節點的父節點，將其父節點的鍊結打斷，但因為走訪是用遞迴的寫法，所以無法在遞迴程式中呼叫所輸入的根節點的父節點，因此在Node的物件中有設parent的變數，知需將node.parent的leftchild或rightchild設為None即可。

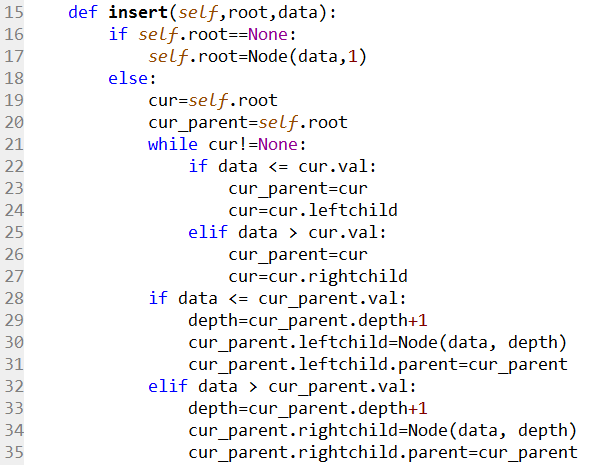
1. **Pre-order走訪心得**

Pre-order 走訪的寫法是參考 <使用資料結構Python 第7章7-20> 的寫法。該方法採用遞迴的寫法，我的理解是，因為愈上面的程式碼會愈先執行，所以self.pre\_order(root.leftchild) 會一直不斷被執行直到最左邊的葉節點為止，印出的也就會是最左邊的葉節點；接著被執行的會是倒數第二層遞迴程式中self.pre\_order(root.rightchild)，印出的會是由左邊數來第二個葉節點。接著以此類推，遞迴程式會由下往上執行。



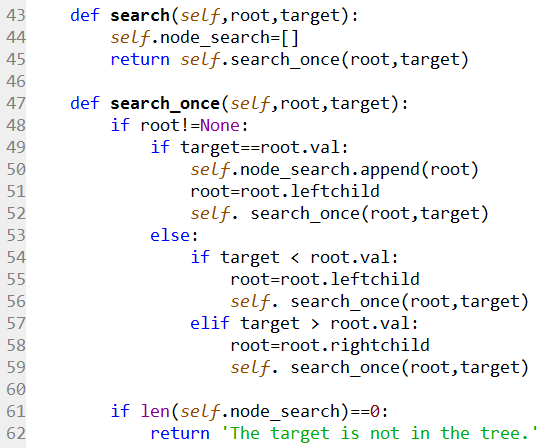
1. **Insert 不同寫法**

採用非遞迴的寫法，像是LinkedList訪尋的方式，先設cur=self.root，再不斷用cur=cur.leftchild或cur=cur.rightchild往下訪尋。

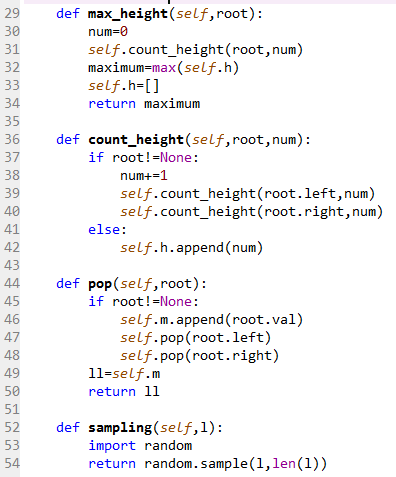
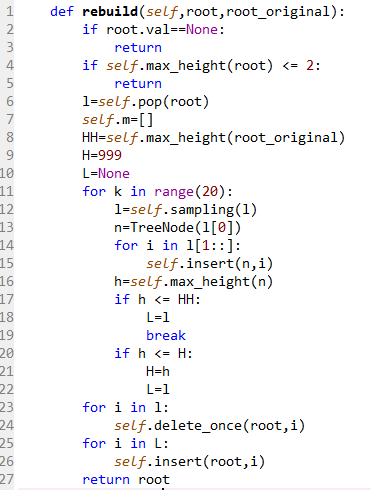


1. **Search不同寫法**

在Node的物件中有設depth的變數，每當新增一個節點，該節點的depth就會比其父節點的depth多1，又因為Search是要回傳離根節點最近的節點，當發現欲搜尋的值則將該節點append到\_\_init\_\_中的list中， append完所數值相同的節點後再return depth最小的節點。



1. **重新整理樹狀結構:**



如果輸入的測值是不符合BST的樹狀結構，就先用Pre-order訪尋所有節點，將這些節點的樹值一一append到\_\_init\_\_裡的list中。為了盡量讓樹狀結構的樹高愈小愈好，所以會將list中的數值做20種隨機排列，按20種不同順序insert進樹狀結構中，最後選一組會讓樹狀結構的樹高最小的排序來建這棵樹，將這個樹狀結構所有的節點刪除後 (根節點的數值設為None)，然後再重新insert 數值進去。其中樹高的計算方式是用Pre-order訪尋所有節點，每走訪一次父節點樹深就加一，當訪尋到最後的葉節點時就append該葉節點的深度到\_\_init\_\_的list中，最後將list中的最大值當作該樹狀結構的樹高。

1. **參考資料**
   1. 使用資料結構Python 第7章7-20