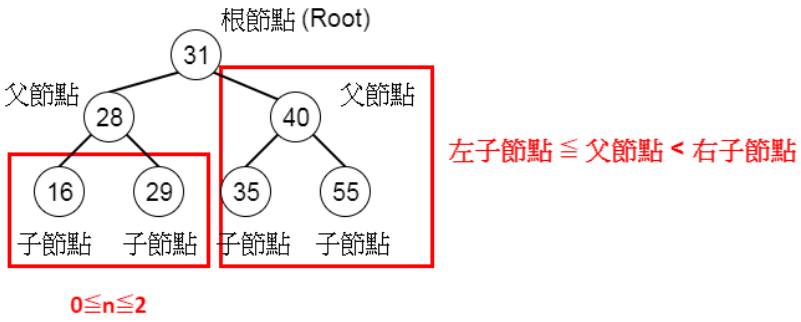
Binary Search Tree (BST) 原理、流程圖、學習歷程

一、 BST 原理 & 流程圖

A. 結構

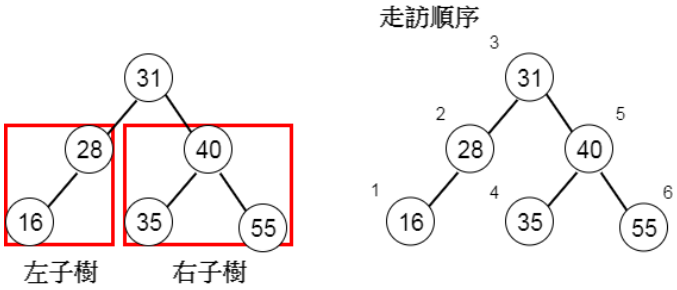
1. 為樹狀結構，由一個或多個節點組合而成的有限集合。
2. 樹不可以為空，至少有一個特殊的節點稱為「根節點」(Root)。
3. 根節點之下的節點為 0 ≦ n ≦ 2 個互斥的子集合 T_1、T_2...T_n，每一個子集合本身也是一棵樹。
4. 每一個節點都會儲存一個值，稱為「鍵值」。
5. 每一個節點的鍵值大於等於左子節點的鍵值；每一個節點的鍵值小於右子節點的鍵值。



B. 走訪

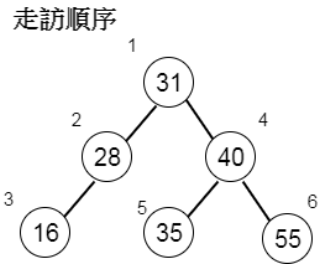
1. 中序走訪 (In-order)

   走訪順序為: 「左子樹 > 樹根 > 右子樹」



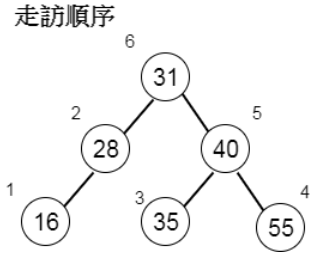
2. 前序走訪 (Pre-order)

   走訪順序為: 「樹根 > 左子樹 > 右子樹」



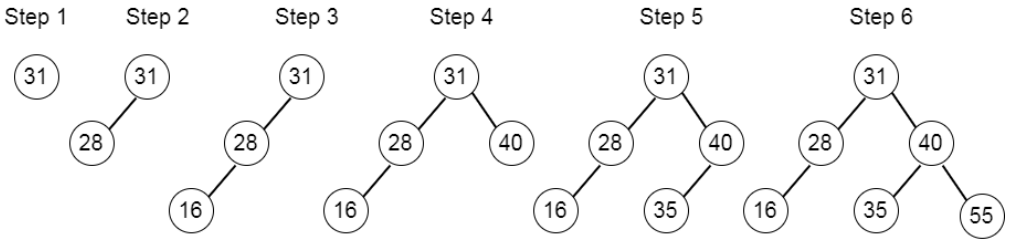
3. 後序走訪 (Post-order)

走訪順序為: 「左子樹 > 右子樹 > 樹根」



C. 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。



i. 新增 31: 先設根節點，31 為其鍵值。

ii. 新增 28: 28 比根節點小，所以設為左子節點。

iii. 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。

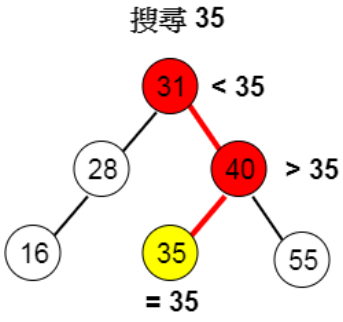
iv. 新增 40: 40 比根節點大，所以設為右子節點。

v. 新增 35: 35 比根節點大，比 40 小，所以設為右子數 40 的左節點。

vi. 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。

D. 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。

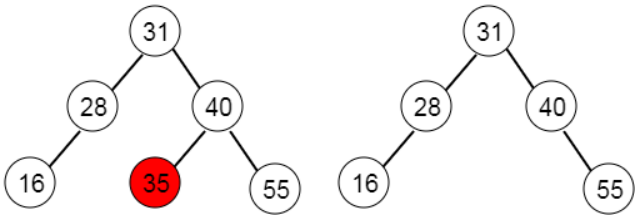


E. 刪除

1. 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。

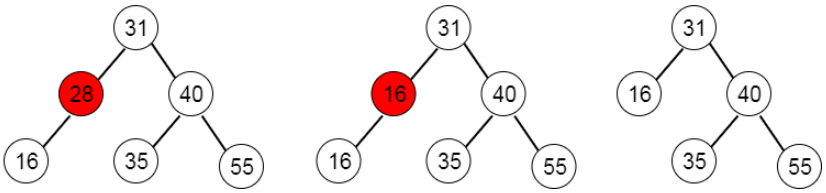
   刪除 35



2. 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。

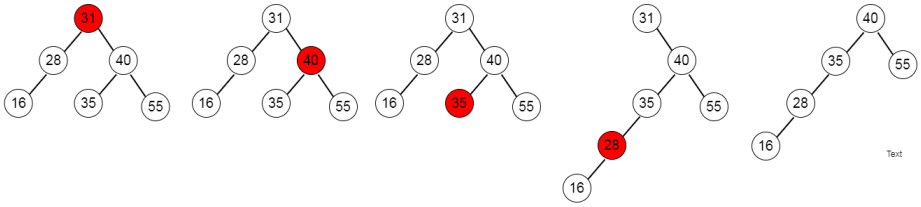
   刪除 28



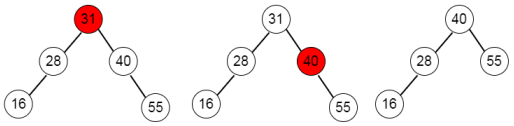
3. 欲刪除的節點有兩個子節點:

   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。

   刪除 31



   刪除 31



二、 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)

A. 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。

```
class Node:
    def __init__(self,val,depth):
        self.val=val
        self.depth=depth
        self.parent=None
        self.leftchild=None
        self.rightchild=None

    def insert(self,val,direction):
        if direction=='l':
            self.leftchild=Node(val,self.depth+1)
            self.leftchild.parent=self
        elif direction=='r':
            self.rightchild=Node(val,self.depth+1)
            self.rightchild.parent=self
class Tree:
    def __init__(self,data):
        self.root=Node(data,1)
        self.s=[]

    def search(self,node,data):
        if node!=None:
            if node.val==data:
                self.s.append(node)
            self.search(node.leftchild,data)
            self.search(node.rightchild,data)
        if len(self.s)!=0:
            min_depth=999
            min_node=None
            for i in self.s:
                if i.depth <= min_depth:
                    min_depth=i.depth
                    min_node=i
            return min_node
        else:
            return 'the data is not in the tree'

    def replace(self,node,target,data):
        if node!=None:
            if node.val==target:
                node.val=data
            self.replace(node.leftchild,target,data)
            self.replace(node.rightchild,target,data)
    def delete(self,node,target):
        self.s=[]
        while self.search(node,target)!='the data is not in the tree':
            self.delete_once(node,target)
            self.s=[]

    def delete_once(self,node,target):
        if node!=None:
            if node.val==target:
                cur=node
                cur_parent=node.parent
                while cur.leftchild!=None or cur.rightchild!=None:
                    if cur.leftchild!=None:
                        cur_parent=cur
                        cur=cur.leftchild
                    elif cur.rightchild!=None:
                        cur_parent=cur
                        cur=cur.rightchild
                if cur.parent==None:
                    cur.val=None
                else:
                    node.val=cur.val
                    if cur.parent.leftchild==cur:
                        cur_parent.leftchild=None
                        cur.parent=None
                    elif cur.parent.rightchild==cur:
                        cur_parent.rightchild=None
                        cur.parent=None
            self.delete_once(node.leftchild,target)
            self.delete_once(node.rightchild,target)
```

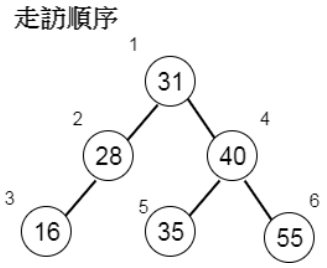
搜尋方面，採用 Pre-order 走訪，Pre-order 的寫法是參考教科書的寫法，用遞迴的方式訪尋每個節點，當發現欲搜尋的值則將該節點 append 到__init__中的 list 中，因為搜尋的節點需離根節點最近，所以在 Node 的物件中有設 depth 的變數，每當新增一個節點，該節點的 depth 就會比其父節點的 depth 多 1，因此當 append 完所數值相同的節點後再 return depth 最小的節點。

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。

B. 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。




```
def pre_order(self,root):
    if root!=None:
        print(root.val,'',end='')
        self.pre_order(root.leftchild)
        self.pre_order(root.rightchild)

In [138]: T.pre_order(T.root)
31 28 16 40 35 55
```

C. 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。

```
15    def insert(self,root,data):
16        if self.root==None:
17            self.root=Node(data,1)
18        else:
19            cur=self.root
20            cur_parent=self.root
21            while cur!=None:
22                if data <= cur.val:
23                    cur_parent=cur
24                    cur=cur.leftchild
25                elif data > cur.val:
26                    cur_parent=cur
27                    cur=cur.rightchild
28            if data <= cur_parent.val:
29                depth=cur_parent.depth+1
30                cur_parent.leftchild=Node(data, depth)
31                cur_parent.leftchild.parent=cur_parent
32            elif data > cur_parent.val:
33                depth=cur_parent.depth+1
34                cur_parent.rightchild=Node(data, depth)
35                cur_parent.rightchild.parent=cur_parent
```

D. 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。

```
    def search(self,root,target):
        self.node_search=[]
        return self.search_once(root,target)

    def search_once(self,root,target):
        if root!=None:
            if target==root.val:
                self.node_search.append(root)
                root=root.leftchild
                self. search_once(root,target)
            else:
                if target < root.val:
                    root=root.leftchild
                    self. search_once(root,target)
                elif target > root.val:
                    root=root.rightchild
                    self. search_once(root,target)

        if len(self.node_search)==0:
            return 'The target is not in the tree.'
```

E. 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。

```
    def rebuild(self,root,root_original):
        if root.val==None:
            return
        if self.max_height(root) <= 2:
            return
        l=self.pop(root)
        self.m=[]
        HH=self.max_height(root_original)
        H=999
        L=None
        for k in range(20):
            l=self.sampling(l)
            n=TreeNode(l[0])
            for i in l[1::]:
                self.insert(n,i)
            h=self.max_height(n)
            if h <= HH:
                L=l
                break
            if h <= H:
                H=h
                L=l
        for i in l:
            self.delete_once(root,i)
        for i in L:
            self.insert(root,i)
        return root

    def max_height(self,root):
        num=0
        self.count_height(root,num)
        maximum=max(self.h)
        self.h=[]
        return maximum

    def count_height(self,root,num):
        if root!=None:
            num+=1
            self.count_height(root.left,num)
            self.count_height(root.right,num)
        else:
            self.h.append(num)

    def pop(self,root):
        if root!=None:
            self.m.append(root.val)
            self.pop(root.left)
            self.pop(root.right)
        ll=self.m
        return ll

    def sampling(self,l):
        import random
        return random.sample(l,len(l))
```

三、 參考資料

1. 使用資料結構 Python 第 7 章 7-20