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。

```python
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)
```