树的构造篇

这篇总结主要介绍树中比较常见的一类题型 -- 树的构造。其实本质还是用递归的手法来实现，但是这类题目有一个特点，就是它是构建一棵树，而不是给定一棵树，然后进行遍历，所以实现起来思路上有点逆向，还是要练习一下。

LeetCode 中关于树的构造的题目有以下几道：

1. Convert Sorted Array to Binary Search Tree
2. Convert Sorted List to Binary Search Tree
3. Construct Binary Tree from Preorder and Inorder Traversal
4. Construct Binary Tree from Inorder and Postorder Traversal

1. Convert Sorted Array to Binary Search Tree，其实从本质来看，如果把一个数组看成一棵树（也就是以中点为根，左右为左右子树，依次下去）数组就等价于一个二分查找树。所以如果要构造这棵树，那就是把中间元素转化为根，然后递归构造左右子树。递归的写法跟常规稍有不同，就是要把根 root 先 new 出来，然后它的左节点接到递归左边部分的返回值，右节点接到递归右边部分的返回值，最后将 root 返回回去。这个模板在树的构造中非常有用，其他几道题也都是按照这个来实现。时间复杂度还是一次树遍历 O(n)，总的空间复杂度是栈空间 O(logn)加上结果的空间 O(n)，额外空间是 O(logn)，总体是 O(n)。代码如下：

```
public TreeNode sortedArrayToBST(int[] num) {
    return helper(num, 0, num.length - 1);
}

private TreeNode helper(int[] num, int l, int r){
    if(l > r){
        return null;
    }
    int m = (l + r) / 2;
    TreeNode root = new TreeNode(num[m]);
    root.left = helper(num, l, m - 1);
    root.right = helper(num, m + 1, r);
    return root;
}
```

2. Convert sorted List to Binary Search Tree，这个题是二分查找树的题目，要把一个有序链表转换成一棵二分查找树。其实原理还是跟Convert Sorted Array to Binary Search Tree这道题相似，我们需要取中点作为当前函数的根。这里的问题是对于一个链表我们是不能常量时间访问它的中间元素的。这时候就要利用到树的中序遍历了，按照递归中序遍历的顺序对链表结点一个个进行访问，而我们要构造的二分查找树正是按照链表的顺序来的。思路就是先对左子树进行递归，然后将当前结点作为根，迭代到下一个链表结点，最后在递归求出右子树即可。整体过程就是一次中序遍历，时间复杂度是O(n)，空间复杂度是栈空间O(logn)加上结果的空间O(n)，额外空间是O(logn)，总体是O(n)。

代码如下：

```
public TreeNode sortedListToBST(ListNode head) {
    if(head == null){
        return null;
    }
    ListNode cur = head;
    int count = 0;
    while(cur.next != null){
        cur = cur.next;
        count++;
    }
    ArrayList<ListNode> nextRoot = new ArrayList<ListNode>();
    nextRoot.add(head);
    return helper(nextRoot, 0, count);
}

private TreeNode helper(ArrayList<ListNode> nextRoot, int l, int r){
    if(l > r){
        return null;
    }
    int m = (l + r) / 2;
    TreeNode left = helper(nextRoot, l, m - 1);
    TreeNode root = new TreeNode(nextRoot.get(0).val);
    root.left = left;
    nextRoot.set(0, nextRoot.get(0).next);
    root.right = helper(nextRoot, m + 1, r);
    return root;
}
```

3. Construct Binary Tree from Preorder and Inorder Traversal，假设树的先序遍历是12453687，中序遍历是42516837。这里最重要的一点就是先序遍历可以提供根的所在，而根据中序遍历的性质知道根的所在就可以将序列分为左右子树。比如上述例子，我们知道1是根，所以根据中序遍历的结果425是左子树，而6837就是右子树。接下来根据切出来的左右子树的长度又可以在先序便利中确定左右子树对应的子序列（先序遍历也是先左子树后右子树）。根据这个流程，左子树的先序遍历和中序遍历分别是245和425，右子树的先序遍历和中序遍历则是3687和6837，我们重复以上方法，可以继续找到根和左右子树，直到剩下一个元素。可以看出这是一个比较明显的递归过程，对于寻找根所对应的下标，我们可以先建立一个HashMap，以免后面需要进行线行搜索，这样每次递归中就只需要常量操作就可以完成对根的确定和左右子树的分割。算法最终相当于一次树的遍历，每个结点只会被访问一次，所以时间复杂度是O(n)。而空间我们需要建立一个map来存储元素到下标的映射，所以是O(n)。

代码如下：

```
    public TreeNode buildTree(int[] preorder, int[] inorder) {
        HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();
        for(int i = 0; i < inorder.length; i++){
            map.put(inorder[i], i);
        }
        return helper(preorder, inorder, 0, preorder.length - 1, 0, inorder.length - 1,
map);
    }

    private TreeNode helper(int[] preorder, int[] inorder, int preL, int preR, int inL, int
inR, HashMap<Integer, Integer> map){
        if(preL > preR || inL > inR){
            return null;
        }
        TreeNode root = new TreeNode(preorder[preL]);
        int idx = map.get(root.val);
        root.left = helper(preorder, inorder, preL + 1, preL + idx - inL, inL, idx - 1,
map);
        root.right = helper(preorder, inorder, preL + idx - inL + 1, preR, idx + 1, inR,
map);
        return root;
    }
```

4. Construct Binary Tree from Inorder and Postorder Traversal，这道题和Construct Binary Tree from Preorder and Inorder Traversal思路完全一样。区别是要从中序遍历和后序遍历中构造出树，算法还是一样，只是现在取根是从后面取（因为后序遍历根是遍历的最后一个元素）。思想和代码基本都是差不多的，时间复杂度和空间复杂度也还是O(n)。有朋友可能会想根据先序遍历和后序遍历能不能重新构造出树来，答案是否定的。只有中序便利可以根据根的位置切开左右子树，其他两种遍历都不能做到，其实先序遍历和后序遍历是不能唯一确定一棵树的，会有歧义发生，也就是两棵不同的树可以有相同的先序遍历和后序遍历。

代码如下：

```
    public TreeNode buildTree(int[] inorder, int[] postorder) {
        HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();
        for(int i = 0; i < inorder.length; i++){
            map.put(inorder[i], i);
        }
        return helper(inorder, postorder, 0, inorder.length - 1, 0, postorder.length - 1,
map);
    }

    private TreeNode helper(int[] inorder, int[] postorder, int inL, int inR, int postL, int
postR, HashMap<Integer, Integer> map){
        if(inL > inR || postL > postR){
            return null;
        }
        TreeNode root = new TreeNode(postorder[postR]);
        int idx = map.get(root.val);
        root.left = helper(inorder, postorder, inL, idx - 1, postL, postL + (idx - inL) - 1,
map);
        root.right = helper(inorder, postorder, idx + 1, inR, postL + (idx - inL), postR -
1, map);
        return root;
    }
```

总结：

这篇总结主要介绍了 LeetCode 中四个树的构造的题目，比较统一的思路就是在递归中创建根节点，然后找到将元素劈成左右子树的方法，递归得到左右根节点接上创建的根然后返回。方法还是比较具有模板型的。