二叉查找树篇

这篇总结主要介绍一个比较常见的数据结构 -- 二叉查找树。二叉查找树既是一颗树，又带有特别的有序性质，所以考察的方式比较多而且灵活，属于面试题目中的常客。

LeetCode 中关于二叉查找树的题目有以下几道：

1. Validate Binary Search Tree
2. Recover Binary Search Tree
3. Unique Binary Search Trees
4. Unique Binary Search Trees II
5. Convert Sorted Array to Binary Search Tree
6. Convert Sorted List to Binary Search Tree

1. Validate Binary Search Tree，这道题就是判断一个树是不是二叉查找树。二分查找树是非常常见的一种数据结构，因为它可以在O(logn)时间内实现搜索。这里我们介绍两种方法来解决这个问题。

第一种是利用二叉查找树的中序遍历有序的性质，只要对树进行一次中序遍历，而其中的结点都满足有序即可，实现上就是维护一个前驱结点，每次判断前驱结点比当前结点要小。

注意以下代码我们用一个一个变量的数组去保存前驱结点，原因是java没有传引用的概念，如果传入一个变量，它是按值传递的，所以是一个备份的变量，改变它的值并不能影响它在函数外部的值，算是java中的一个小细节。

代码如下：

```
// Solution 1 - Recursion
public boolean isValidBST(TreeNode root) {
    ArrayList<TreeNode> pre = new ArrayList<TreeNode>();
    pre.add(null);
    return helper(root, pre);
}

private boolean helper(TreeNode root, ArrayList<TreeNode> pre){
    if(root == null){
        return true;
    }
```

```
        boolean left = helper(root.left, pre);
        if(pre.get(0) != null && pre.get(0).val >= root.val){
            return false;
        }
        pre.set(0, root);
        return left && helper(root.right, pre);
    }
```

第二种方法是根据二叉查找树的定义来实现，其实就是对于每个结点保存左右界，保证结点满足它的左子树的每个结点比当前结点值小，右子树的每个结点比当前结点值大，实现上就是对于每个结点保存左右界，然后进行递归判断左右界不会违背即可。

代码如下：

```
    // Solution 2 - Min-Max Range
    public boolean isValidBST(TreeNode root) {
        return helper(root, Long.MIN_VALUE, Long.MAX_VALUE);
    }

    private boolean helper(TreeNode root, long min, long max){
        if(root == null){
            return true;
        }
        if(root.val <= min || root.val >= max){
            return false;
        }
        return helper(root.left, min, root.val) && helper(root.right, root.val, max);
    }
```

上述两种方法本质上都是做一次树的遍历，所以时间复杂度是O(n)，空间复杂度是O(logn)。个人其实更喜欢第一种做法，因为思路简单，而且不需要用到整数最大值和最小值这种跟语言相关的量来定义无穷大和无穷小。

做题时的感悟：

1. 因为有if(root == null) return true;的判断，所以开始不用对root进行是否为空的判断。
2. 虽然第一种的思想很简单而且好实现，但第二种的思想很不错，可能在相关题目中可以用到，值得借鉴。

2. Recover Binary Search Tree，这道题目还是利用二叉查找树的主要性质，就是中序遍历有序性。那么如果其中有元素被调换了，意味着中序遍历中必然出现违背有序的情况。主要考虑到就是出现违背的次数问题。这里有两种情况：

（1）如果是中序遍历相邻的两个元素被调换了，很容易想到就只需会出现一次违反情况，只需要把这个两个节点记录下来最后调换值就可以；

（2）如果是不相邻的两个元素被调换了，会发生两次逆序的情况，那么这时候需要调换的元素应该是第一次逆序前面的元素，和第二次逆序后面的元素。

算法只需要一次中序遍历，所以时间复杂度是 O(n)，空间是栈大小 O(logn)。

代码如下：

```
    public void recoverTree(TreeNode root) {
        ArrayList<TreeNode> pre = new ArrayList<TreeNode>();
        pre.add(null);
        ArrayList<TreeNode> res = new ArrayList<TreeNode>();
        helper(root, pre, res);
        if(res.size() > 0){
            int temp = res.get(0).val;
            res.get(0).val = res.get(1).val;
            res.get(1).val = temp;
        }
    }

    private void helper(TreeNode root, ArrayList<TreeNode> pre,
ArrayList<TreeNode> res){
        if(root == null){ return; }
        helper(root.left, pre, res);
        if(pre.get(0) != null && pre.get(0).val > root.val){
            if(res.size() == 0){
                res.add(pre.get(0));
                res.add(root);
            } else {
                res.set(1, root);
                return;
            }
        }
        pre.set(0, root);
        helper(root.right, pre, res);
    }
```

做题时的感悟：

1. 创建完pre的ArrayList，要先向其中添加一个空元素，不然取的时候会出现空指针。
2. 在最后做交换之前，先判断一下res.size() > 0。如果等于0，说明没有违反的情况，不需交换。虽然这道题肯定有违反的情况，但这样写更加严禁，应该学习。
3. 这道题总的思路就是模仿中序遍历，利用中序遍历的有序性来找到违反条件的结点。但具体情况要分析透彻，因为2个结点相邻和不相邻是2种不同的情况，要分别处理。
4. 二叉查找树的性质：
   1. 若任意节点的左子树不空，则左子树上所有结点的值均小于或等于它的根结点的值；
   2. 任意节点的右子树不空，则右子树上所有结点的值均大于它的根结点的值；
   3. 任意节点的左、右子树也分别为二叉查找树。
   4. 没有键值相等的节点（no duplicate nodes）。

3. Unique Binary Search Trees，这道题要求可行的二叉查找树的数量，其实二叉查找树可以任意取根，只要满足中序遍历有序的要求就可以。从处理子问题的角度来看，选取一个结点为根，就把结点切成左右子树，以这个结点为根的可行二叉树数量就是左右子树可行二叉树数量的乘积，所以总的数量是将以所有结点为根的可行结果累加起来。写成表达式如下：

$$C_0 = 1 \quad \text{and} \quad C_{n+1} = \sum_{i=0}^{n} C_i\, C_{n-i} \quad \text{for } n \geq 0.$$

这其实是一个卡特兰数的模型，所以按照公式进行实现就可以。而Unique Binary Search Trees II则不能用卡特兰数，因为要求出所有结果，所以还是得走递归遍历的过程，然后把生成树来的树接上。时间上每次求解i个结点的二叉查找树数量的需要一个i步的循环，总体要求n次，所以总时间复杂度是O(1+2+...+n)=O(n^2)。空间上需要一个数组来维护，并且需要前i个的所有信息，所以是O(n)。

代码如下：

```
public int numTrees(int n) {
    if(n <= 0){
        return 0;
    }
    int[] res = new int[n + 1];
    res[0] = 1;
    for(int i = 1; i <= n; i++){
```

```
            for(int j = 0; j < i; j++){
                res[i] += res[j] * res[i - 1 - j];
            }
        }
        return res[n];
    }
```

做题时的感悟：

1. 注意数组长度，因为要求到n，所以数组长度一定要为n + 1。
2. 注意循环的边界，根据公式推出。
3. 这道题还可以用卡特兰数的通项公式来求解，这样时间复杂度就可以降低到O(n)。

4. Unique Binary Search Trees II，这道题是求解所有可行的二叉查找树，从Unique Binary Search Trees中我们已经知道，可行的二叉查找树的数量是相应的卡特兰数，不是一个多项式时间的数量级，所以我们要求解所有的树，自然是不能多项式时间内完成的了。算法上还是用求解NP问题的方法来求解，也就是N-Queens中介绍的在循环中调用递归函数求解子问题。思路是每次一次选取一个结点为根，然后递归求解左右子树的所有结果，最后根据左右子树的返回的所有子树，依次选取然后接上（每个左边的子树跟所有右边的子树匹配，而每个右边的子树也要跟所有的左边子树匹配，总共有左右子树数量的乘积种情况），构造好之后作为当前树的结果返回。

当然我们也可以像在Unique Binary Search Trees中那样存储所有的子树历史信息，然后进行拼合，虽然可以省一些时间，但是最终还是逃不过每个结果要一次运算，时间复杂度还是非多项式的，并且要耗费大量的空间，这样的意义就不是很大了。

代码如下：

```
    public ArrayList<TreeNode> generateTrees(int n) {
        return helper(1, n);
    }

    private ArrayList<TreeNode> helper(int left, int right){
        ArrayList<TreeNode> res = new ArrayList<TreeNode>();
        if(left > right){
            res.add(null);
            return res;
        }
```

```
        for(int i = left; i <= right; i++){
            ArrayList<TreeNode> leftList = helper(left, i - 1);
            ArrayList<TreeNode> rightList = helper(i + 1, right);
            for(int j = 0; j < leftList.size(); j++){
                for(int k = 0; k < rightList.size(); k++){
                    TreeNode root = new TreeNode(i);
                    root.left = leftList.get(j);
                    root.right = rightList.get(k);
                    res.add(root);
                }
            }
        }
        return res;
    }
```

做题时的感悟:

1. NP问题 —> 循环递归 —> 考虑循环什么，递归什么。

循环中，以每一个数字新建一个TreeNode作为根结点，然后递归左右子树，获得左右子树所有可能的根节点。

2. 以当前结点作为根，结合 m * n 种所有左右子树的组合，将当前根存入 res 结果中。

5 & 6. Convert Sorted Array to Binary Search Tree 和 Convert Sorted List to Binary Search Tree 则是属于二叉查找树的构造问题，针对两种不同数据结构数组和链表进行构造。其实方法都是一样，就是递归对窗口进行圈限，然后用中间的结点作为当前根，再递归生成左右子树。链表的构造要稍微绕一些，因为要通过中序遍历走到第一个结点，然后递进链表。这两道题已经在树的构造篇中提及，不再赘述。

做题时的感悟:

1. 递归一般都会需要l和r，这样才能确定递归区域。所以该题要先扫描一遍求出链表长度。

2. 递归过程中的代码有顺序要求，不能颠倒。我们需要模拟一个中序遍历的过程。所以先递归左子树，注意在这阶段nextRoot中存储的结点也是不断更新的，所以保证其中存的一定是左子树根的后续结点。然后，更新nextRoot中的后续结点，再递归右子树即可。