树的遍历篇

Morris Traversal 方法遍历二叉树（非递归，不用栈，O(1)空间）：
http://www.cnblogs.com/AnnieKim/archive/2013/06/15/MorrisTraversal.html

树的遍历题目在 LeetCode 中有以下几个：

1. Binary Tree Inorder Traversal
2. Binary Tree Preorder Traversal
3. Binary Tree Postorder Traversal
4. Binary Tree Level Order Traversal
5. Binary Tree Level Order Traversal II
6. Binary Tree Zigzag Level Order Traversal

遍历树是数据结构中最常见的操作，可以说大部分关于树的题目都是围绕遍历进行变体来解决的。一般来说面试中遇到树的题目是用递归来解决的，不过如果直接考察遍历，那么一般递归的解法就过于简单了，面试官一般还会问更多问题，比如非递归实现，或者空间复杂度分析以及能否优化等等。

树的遍历基本上分成两种类型，下面分别介绍：

（1）第一种是以图的深度优先搜索为原型的遍历，可以是中序，先序和后序三种方式，不过结点遍历的方式是相同的，只是访问的时间点不同而已。对应于Binary Tree Inorder Traversal, Binary Tree Preorder Traversal和Binary Tree Postorder Traversal这三道题。在这种类型中，递归的实现方式是非常简单的，只需要递归左右结点，直到结点为空作为结束条件就可以，哪种序就取决于你访问结点的时间。

不过一般这不能满足面试官的要求，可能会接着问能不能用非递归实现一下，这个说起来比较简单，其实就是用一个栈手动模拟递归的过程，Binary Tree Inorder Traversal和Binary Tree Preorder Traversal比较简单，用一个栈来保存前驱的分支结点（相当于图的深度搜索的栈），然后用一个结点来记录当前结点就可以了。而Binary Tree Postorder Traversal则比较复杂一些，保存栈和结点之后还得根据情况来判断当前应该走的方向（往左， 往右或者回溯）。

有时候非递归还是不能满足面试官，还会问一问，上面的做法时间和空间复杂度是多少。我们知道，正常遍历时间复杂度是 O(n)，而空间复杂度是则是递归栈(或者自己维护的栈)的大小，也就是 O(logn)。他会问能不能够在常量空间内解决树的遍历问题呢？确实还真可以，这里就要介绍 Morris Traversal 的方法。Morris 遍历方法用了线索二叉树，这个方法不需要为每个节点额外分配指针指向其前驱和后继结点，而是利用叶子节点中的右空指针指向中序遍历下的后继节点就可以了。这样就节省了需要用栈来记录前驱或者后继结点的额外空间，所以可以达到 O（1）的空间复杂度。不过这种方法有一个问题就是会暂时性的改动树的结构，这在程序设计中并不是很好的习惯，这些在面试中都可以和面试官讨论，一般来说问到这里不会需要进行 Morris 遍历方法的代码实现了，只需要知道这种方法和他的主要优劣势就可以了。

1. Binary Tree Inorder Traversal，二叉树的遍历我们介绍三种方法：

第一种是递归，第二种是迭代，第三种是线索二叉树。

递归是最常用的算法，时间复杂度是O(n)，空间复杂度则是递归栈的大小，即O(logn)。

代码如下：

```
// Solution 1 - Recursion
public ArrayList<Integer> inorderTraversal(TreeNode root) {
    ArrayList<Integer> res = new ArrayList<Integer>();
    helper(root, res);
    return res;
}

private void helper(TreeNode root, ArrayList<Integer> res){
    if(root == null){
        return;
    }
    helper(root.left, res);
    res.add(root.val);
    helper(root.right, res);
}
```

栈实现的迭代，其实就是用一个栈来模拟递归的过程。所以算法时间复杂度也是 O(n)，空间复杂度是栈的大小 O(logn)。过程中维护一个 node 表示当前走到的结点（不是中序遍历的那个结点）。

代码如下：

```
// Solution 2 - Iteration
public ArrayList<Integer> inorderTraversal(TreeNode root) {
    ArrayList<Integer> res = new ArrayList<Integer>();
    LinkedList<TreeNode> stack = new LinkedList<TreeNode>();
    while(root != null || !stack.isEmpty()){
        if(root != null){
            stack.push(root);
            root = root.left;
        } else {
            root = stack.pop();
            res.add(root.val);
            root = root.right;
        }
    }
    return res;
}
```

Morris Traversal - Threaded Binary Tree（线索二叉树），若要用O(1)空间进行遍历，因为不能用栈作为辅助空间来保存父节点的信息，重点在于当访问到子节点的时候如何重新回到父节点（当然这里是指没有父节点指针，如果有其实就比较好办，一直找遍历的后驱结点即可）。Morris遍历方法用了线索二叉树，这个方法不需要为每个节点额外分配指针指向其前驱和后继结点，而是利用叶子节点中的右空指针指向中序遍历下的后继节点即可。

算法具体分情况如下：

1. 如果当前结点的左孩子为空，则输出当前结点并将其当前节点赋值为右孩子。
2. 如果当前节点的左孩子不为空，则寻找当前节点在中序遍历下的前驱节点（也就是当前结点左子树的最右孩子）。

接下来分两种情况：

a）如果前驱节点的右孩子为空，将它的右孩子设置为当前节点（做线索使得稍后可以重新返回父结点）。然后将当前节点更新为当前节点的左孩子。

b）如果前驱节点的右孩子为当前节点，表明左子树已经访问完，可以访问当前节点。将它的右孩子重新设为空（恢复树的结构）。输出当前节点。当前节点更新为当前节点的右孩子。

时间、空间复杂度：

整个过程中每条边最多只走 2 次，一次是为了定位到某个节点，另一次是为了寻找上面某个节点的前驱节点，而 n 个结点的二叉树中有 n-1 条边，所以时间复杂度是 O(2*n)=O(n)，仍然是一个线性算法。空间复杂度的话我们分析过了，只是两个辅助指针，所以是 O(1)。

代码如下：

```
// Solution 3 – Morris Traversal
public ArrayList<Integer> inorderTraversal(TreeNode root) {
    ArrayList<Integer> res = new ArrayList<Integer>();
    while(root != null){
        if(root.left == null){
            res.add(root.val);
            root = root.right;
        } else {
            TreeNode pre = root.left;
            while(pre.right != null && pre.right != root){
                pre = pre.right;
            }
            if(pre.right == null){
                pre.right = root;
                root = root.left;
            } else {
                pre.right = null;
                res.add(root.val);
                root = root.right;
            }
        }
    }
    return res;
}
```

做题时的感悟：

1. 迭代做法的时候，可以直接使用root作为当前走到的结点，不用新建一个cur结点。同时，注意循环条件，root非空或者stack中还有元素，都可以继续循环。

2. 线索二叉树同迭代一样，可以直接使用root作为当前走到的结点，不用新建一个cur结点。分清楚3种条件对应的处理方法即可。

2. Binary Tree Preorder Traversal，解法与inorder基本相同，不再赘述。

递归：

代码如下：

```
// Solution 1 - Recursion
public ArrayList<Integer> preorderTraversal(TreeNode root) {
    ArrayList<Integer> res = new ArrayList<Integer>();
    helper(root, res);
    return res;
}

private void helper(TreeNode root, ArrayList<Integer> res){
    if(root == null){
        return;
    }
    res.add(root.val);
    helper(root.left, res);
    helper(root.right, res);
}
```

栈实现的迭代 ：

代码如下：

```
// Solution 2 - Iteration
public ArrayList<Integer> preorderTraversal(TreeNode root) {
    ArrayList<Integer> res = new ArrayList<Integer>();
    LinkedList<TreeNode> stack = new LinkedList<TreeNode>();
    while(root != null || !stack.isEmpty()){
        if(root != null){
            stack.push(root);
            res.add(root.val);
            root = root.left;
        } else {
            root = stack.pop();
            root = root.right;
        }
    }
    return res;
}
```

Morris Traversal - Threaded Binary Tree（线索二叉树）：

代码如下：

```
// Solution 3 – Morris Traversal
public ArrayList<Integer> preorderTraversal(TreeNode root) {
    ArrayList<Integer> res = new ArrayList<Integer>();
    while(root != null){
        if(root.left == null){
            res.add(root.val);
            root = root.right;
        } else {
            TreeNode pre = root.left;
            while(pre.right != null && pre.right != root){
                pre = pre.right;
            }
            if(pre.right == null){
                pre.right = root;
                res.add(root.val);
                root = root.left;
            } else {
                pre.right = null;
                root = root.right;
            }
        }
    }
    return res;
}
```

做题时的感悟：

1. 迭代做法与inorder的唯一不同就是存储root是时机，inorder是在pop出栈是时候才存储root，而preorder在push进栈的时候就要存储root。

2. 线索二叉树也一样，与inorder的唯一不同就是存储root的时机，inorder是在左子树都处理完毕，到最后一个结点（即root的前驱结点）的时候，发现pre的右指针指向root，此时存储root。而preorder在找到root的前驱结点的时候就将root储存。

3. Binary Tree Postorder Traversal，递归与inorder基本相同。但迭代和线索二叉树情况要复杂许多。

递归：

代码如下：

```
// Solution 1 - Recursion
public ArrayList<Integer> postorderTraversal(TreeNode root) {
    ArrayList<Integer> res = new ArrayList<Integer>();
    helper(root, res);
    return res;
}

private void helper(TreeNode root, ArrayList<Integer> res){
    if(root == null){
        return;
    }
    helper(root.left, res);
    helper(root.right, res);
    res.add(root.val);
}
```

栈实现的迭代：

介绍一种跟Binary Tree Inorder Traversal和Binary Tree Preorder Traversal非常类似的解法，容易统一进行记忆，思路可以参考其他两种，区别是最下面在弹栈的时候需要分情况讨论一下：

1）如果当前栈顶元素的右结点存在并且还没访问过（也就是右结点不等于上一个访问结点），那么就把当前结点移到右结点继续循环；

2）如果栈顶元素右结点是空或者已经访问过，那么说明栈顶元素的左右子树都访问完毕，应该访问自己继续回溯了。

代码如下：

```
public ArrayList<Integer> postorderTraversal(TreeNode root) {
    ArrayList<Integer> res = new ArrayList<Integer>();
    if(root == null){
        return res;
    }
    LinkedList<TreeNode> stack = new LinkedList<TreeNode>();
```

```
        TreeNode pre = null;
        while(root != null || !stack.isEmpty()){
            if(root != null){
                stack.push(root);
                root = root.left;
            } else {
                TreeNode peekNode = stack.peek();
                if(peekNode.right != null && peekNode.right != pre){
                    root = peekNode.right;
                } else {
                    res.add(peekNode.val);
                    stack.pop();
                    pre = peekNode;
                }
            }
        }
        return res;
    }
```

Morris Traversal - Threaded Binary Tree（线索二叉树）：

在这里，我们需要创建一个临时的根节点 dummy，把它的左孩子设为树的根 root。同时还需要一个 subroutine 来倒序输出一条右孩子路径上的结点。

代码如下：

```
    // Solution 4 – Morris Traversal
    public ArrayList<Integer> postorderTraversal(TreeNode root) {
        ArrayList<Integer> res = new ArrayList<Integer>();
        TreeNode dummy = new TreeNode(0);
        dummy.left = root;
        root = dummy;
        while(root != null){
            if(root.left == null){
                root = root.right;
            } else {
                TreeNode pre = root.left;
                while(pre.right != null && pre.right != root){
                    pre = pre.right;
                }
                if(pre.right == null){
```

```
                    pre.right = root;
                    root = root.left;
                } else {
                    reverse(root.left, pre);
                    TreeNode temp = pre;
                    while(temp != root.left){
                        res.add(temp.val);
                        temp = temp.right;
                    }
                    res.add(temp.val);
                    reverse(pre, root.left);
                    pre.right = null;
                    root = root.right;
                }
            }
        }
        return res;
    }

    private void reverse(TreeNode start, TreeNode end){
        if(start == end){
            return;
        }
        TreeNode pre = start;
        TreeNode cur = start.right;
        TreeNode next;
        while(pre != end){
            next = cur.right;
            cur.right = pre;
            pre = cur;
            cur = next;
        }
    }
```

做题时的感悟:

1. 新迭代解法与inorder和preorder的迭代解法更相近，更容易记忆。主要是要明白弹栈时的分情况讨论的2种情况。

2. 线索二叉树与 inorder 和 preorder 也非常相近，只是存储结点只在倒序输出时进行。

另一种是以图的广度优先搜索为原型的，在树中称为层序遍历，LeetCode中有三种自顶向下层序，自底向上层序和锯齿层序遍历，对应于Binary Tree Level Order Traversal，Binary Tree Level Order Traversal II和Binary Tree Zigzag Level Order Traversal。

4. Binary Tree Level Order Traversal，这道题要求实现树的层序遍历，其实本质就是把树看成一个有向图，然后进行一次广度优先搜索。这里同样是维护一个队列，只是对于每个结点我们知道它的邻接点只有可能是左孩子和右孩子。算法的复杂度是就结点的数量O(n)，空间复杂度是一层的结点数，也是O(n)。

代码如下：

```
public ArrayList<ArrayList<Integer>> levelOrder(TreeNode root) {
    ArrayList<ArrayList<Integer>> res = new ArrayList<ArrayList<Integer>>();
    if(root == null){
        return res;
    }
    LinkedList<TreeNode> queue = new LinkedList<TreeNode>();
    queue.offer(root);
    ArrayList<Integer> list = new ArrayList<Integer>();
    int curNum = 1;
    int nextNum = 0;
    while(!queue.isEmpty()){
        TreeNode cur = queue.poll();
        curNum--;
        list.add(cur.val);
        if(cur.left != null){
            queue.offer(cur.left);
            nextNum++;
        }
        if(cur.right != null){
            queue.offer(cur.right);
            nextNum++;
        }
        if(curNum == 0){
            res.add(list);
            list = new ArrayList<Integer>();
            curNum = nextNum;
            nextNum = 0;
        }
```

```
        }
        return res;
    }
```

5. Binary Tree Level Order Traversal II, 进行Binary Tree Level Order Traversal中的遍历，然后对结果进行一次reverse。时间上和空间上仍是O(n)。基本和前一道题没区别。

代码如下：

```
    public ArrayList<ArrayList<Integer>> levelOrderBottom(TreeNode root) {
        ArrayList<ArrayList<Integer>> res = new ArrayList<ArrayList<Integer>>();
        if(root == null){
            return res;
        }
        LinkedList<TreeNode> queue = new LinkedList<TreeNode>();
        queue.offer(root);
        int curNum = 1;
        int nextNum = 0;
        ArrayList<Integer> list = new ArrayList<Integer>();
        while(!queue.isEmpty()){
            TreeNode cur = queue.poll();
            curNum--;
            list.add(cur.val);
            if(cur.left != null){
                queue.offer(cur.left);
                nextNum++;
            }
            if(cur.right != null){
                queue.offer(cur.right);
                nextNum++;
            }
            if(curNum == 0){
                curNum = nextNum;
                nextNum = 0;
                res.add(list);
                list = new ArrayList<Integer>();
            }
        }
        Collections.reverse(res);
        return res;
}
```

6. Binary Tree Zigzag Level Order Traversal，这道题其实还是树的层序遍历Binary Tree Level Order Traversal，不过这里稍微做了一点变体，就是在遍历的时候偶数层自左向右，而奇数层自右向左。

在 Binary Tree Level Order Traversal 中我们是维护了一个队列来完成遍历，而在这里为了使每次都倒序出来，我们很容易想到用栈的结构来完成这个操作。有一个区别是这里我们需要一层一层的来处理（原来可以按队列插入就可以，因为后进来的元素不会先处理），所以这里同时维护新旧两个栈，一个来读取，一个存储下一层结点。总体来说还是一次遍历完成，所以时间复杂度是 O(n)，空间复杂度最坏是两层的结点，所以数量级还是 O(n)（满二叉树最后一层的结点是 n/2 个）。

代码如下：

```
public ArrayList<ArrayList<Integer>> zigzagLevelOrder(TreeNode root) {
    ArrayList<ArrayList<Integer>> res = new ArrayList<ArrayList<Integer>>();
    if(root == null){
        return res;
    }
    LinkedList<TreeNode> stack = new LinkedList<TreeNode>();
    stack.push(root);
    int curNum = 1;
    int nextNum = 0;
    int level = 0;
    ArrayList<Integer> list = new ArrayList<Integer>();
    while(!stack.isEmpty()){
        LinkedList<TreeNode> curStack = new LinkedList<TreeNode>();
        while(!stack.isEmpty()){
            TreeNode cur = stack.pop();
            curNum--;
            list.add(cur.val);
            if((level & 1) == 0){
                if(cur.left != null){
                    curStack.push(cur.left);
                    nextNum++;
                }
                if(cur.right != null){
                    curStack.push(cur.right);
                    nextNum++;
                }
            } else {
```

```
                    if(cur.right != null){
                        curStack.push(cur.right);
                        nextNum++;
                    }
                    if(cur.left != null){
                        curStack.push(cur.left);
                        nextNum++;
                    }
                }
                if(curNum == 0){
                    curNum = nextNum;
                    nextNum = 0;
                    level++;
                    res.add(list);
                    list = new ArrayList<Integer>();
                }
            }
            stack = curStack;
        }
        return res;
    }
```

做题时的感悟:

所有对于2的n次方数做模运算，都可以用对2的n次方减1做与运算来代替，而且效率更高。

例如：level % 2 == 0 可以写成 (level & 1) == 0. 同理，n % 8 可以写成 n & 7.