树的陨落篇

**树的题目大致可以分为以下几种：树的遍历，其中包括DFS和BFS两种；使用Divide & Conquer算法解的题目；以及BST相关的题目。本篇将分别进行讲解。**

**Part 1 – DFS**

**在LeetCode中使用DFS对树进行遍历的题目有：**

**1. Binary Tree Inorder Traversal**

**2. Binary Tree Preorder Traversal**

**3. Binary Tree Postorder Traversal**

**使用DFS对树进行遍历的题目，一般有4种解法：**

**1. Recursion 2. Divide & Conquer 3. Iteration 4. Morris**

**下面题目中将分别用4种方法来解答，但重点掌握Recursion和Iteration即可，每种都有模板，一定要牢记模板，并根据题目在模板上做相应的修改。**

**1. Binary Tree Inorder Traversal，Recursion和Divide & Conquer两种方法最容易实现，Iteration方法可以会被问道，也需熟悉，而Morris了解思想即可。**

// Solution 1 – Recursion

public List<Integer> inorderTraversal(TreeNode root) {

List<Integer> res = new ArrayList<Integer>();

helper(root, res);

return res;

}

private void helper(TreeNode root, List<Integer> res){

if(root == null){

return;

}

helper(root.left, res);

res.add(root.val);

helper(root.right, res);

}

// Solution 2 – Divide & Conquer

public List<Integer> inorderTraversal(TreeNode root) {

List<Integer> res = new ArrayList<Integer>();

if(root == null){

return res;

}

List<Integer> leftList = inorderTraversal(root.left);

List<Integer> rightList = inorderTraversal(root.right);

res.addAll(leftList);

res.add(root.val);

res.addAll(rightList);

return res;

}

// Solution 3 – Iteration

public List<Integer> inorderTraversal(TreeNode root) {

List<Integer> res = new ArrayList<Integer>();

if(root == null){

return res;

}

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;

}

// Solution 4 – Morris

public List<Integer> inorderTraversal(TreeNode root) {

List<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;

}

**2. Binary Tree Preorder Traversal，解法与Inorder相同。Recursion和Divide & Conquer两种方法最容易实现，Iteration方法可以会被问道，也需熟悉，而Morris了解思想即可。**

// Solution 1 – Recursion

public List<Integer> preorderTraversal(TreeNode root) {

List<Integer> res = new ArrayList<Integer>();

helper(root, res);

return res;

}

private void helper(TreeNode root, List<Integer> res){

if(root == null){

return;

}

res.add(root.val);

helper(root.left, res);

helper(root.right, res);

}

// Solution 2 – Divide & Conquer

public List<Integer> preorderTraversal(TreeNode root) {

List<Integer> res = new ArrayList<Integer>();

if(root == null){

return res;

}

List<Integer> leftList = preorderTraversal(root.left);

List<Integer> rightList = preorderTraversal(root.right);

res.add(root.val);

res.addAll(leftList);

res.addAll(rightList);

return res;

}

// Solution 3 – Iteration

public List<Integer> preorderTraversal(TreeNode root) {

List<Integer> res = new ArrayList<Integer>();

if(root == null){

return res;

}

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;

}

// Solution 4 – Morris

public List<Integer> preorderTraversal(TreeNode root) {

List<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;

}

**3. Binary Tree Postorder Traversal，与Inorder和Preorder的解法基本相同。Recursion和Divide & Conquer两种方法最容易实现，Iteration方法可以会被问道，也需熟悉，而Morris了解思想即可。后续遍历比先序和中序要稍微麻烦一点Iteration方法需要添加判断，Morris需要添加一个倒序的方法。**

// Solution 1 – Recursion

public List<Integer> postorderTraversal(TreeNode root) {

List<Integer> res = new ArrayList<Integer>();

helper(root, res);

return res;

}

private void helper(TreeNode root, List<Integer> res){

if(root == null){

return;

}

helper(root.left, res);

helper(root.right, res);

res.add(root.val);

}

// Solution 2 – Divide & Conquer

public List<Integer> postorderTraversal(TreeNode root) {

List<Integer> res = new ArrayList<Integer>();

if(root == null){

return res;

}

List<Integer> leftList = postorderTraversal(root.left);

List<Integer> rightList = postorderTraversal(root.right);

res.addAll(leftList);

res.addAll(rightList);

res.add(root.val);

return res;

}

// Solution 3 – Iteration

public List<Integer> postorderTraversal(TreeNode root) {

List<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 && pre != peekNode.right){

root = peekNode.right;

} else {

res.add(peekNode.val);

stack.pop();

pre = peekNode;

}

}

}

return res;

}

// Solution 4 – Morris

public List<Integer> postorderTraversal(TreeNode root) {

List<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 {

pre.right = null;

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);

root = root.right;

}

}

}

return res;

}

private void reverse(TreeNode start, TreeNode end){

if(start == end){

return;

}

TreeNode pre = start;

TreeNode cur = start.right;

while(pre != end){

TreeNode next = cur.right;

cur.right = pre;

pre = cur;

cur = next;

}

}

**Part 2 – BFS**

**在LeetCode中使用BFS对树进行遍历的题目有：**

**1. Binary Tree Level Order Traversal**

**2. Binary Tree Zigzag Level Order Traversal**

**3. Maximum Depth of Binary Tree**

**4. Minimum Depth of Binary Tree**

**5. Symmetric Tree**

**6. Populating Next Right Pointers in Each Node**

**使用BFS对树进行遍历的题目解法只有一种，即选用一种合适的数据结构来遍历当前层结点和记录下一层结点，一层一层的遍历即可，所以也可以称为层序遍历。**

**1. Binary Tree Level Order Traversal，这道题是最基础的层序遍历，套用模板即可。**

// Solution – Classic Model

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;

}

**2. Binary Tree Zigzag Level Order Traversal，这道题与Binary Tree Level Order Traversal的区别有以下几点：根据题意，奇数行从左向右，偶数行从右向左，所以需要一个后进先出的数据结构来存储结点，我们很容易想到用栈。由于队列存入的结点永远遵循一个顺序，所以在上一道题中一个queue即可完成任务，但栈后入先出，如果只使用一个栈就会得到错误的结果，所以要额外使用一个栈来存储下一层的结点。**

// Solution – Classic Model

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 level = 1;

ArrayList<Integer> list = new ArrayList<Integer>();

while(!stack.isEmpty()){

LinkedList<TreeNode> curStack = new LinkedList<TreeNode>();

while(!stack.isEmpty()){

TreeNode cur = stack.pop();

list.add(cur.val);

if((level & 1) == 1){

if(cur.left != null){

curStack.push(cur.left);

}

if(cur.right != null){

curStack.push(cur.right);

}

} else {

if(cur.right != null){

curStack.push(cur.right);

}

if(cur.left != null){

curStack.push(cur.left);

}

}

}

level++;

res.add(list);

list = new ArrayList<Integer>();

stack = curStack;

}

return res;

}

**3. Maximum Depth of Binary Tree，这道题一般使用Divide & Conquer方法来做，下面也会介绍。使用层序遍历，用一个level变量来记录也是可以的。**

// Solution – Classic Model

public int maxDepth(TreeNode root) {

if(root == null){

return 0;

}

LinkedList<TreeNode> queue = new LinkedList<TreeNode>();

queue.offer(root);

int curNum = 1;

int nextNum = 0;

int level = 0;

while(!queue.isEmpty()){

TreeNode cur = queue.poll();

curNum--;

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;

level++;

}

}

return level;

}

**4. Minimum Depth of Binary Tree，这道题一般使用Divide & Conquer方法来做，下面也会介绍。使用层序遍历，用一个level变量来记录也是可以的。与上道题的区别是，这里遇到一个叶子结点的时候，就可以返回level了，不用遍历整个树。**

// Solution – Classic Model

public int minDepth(TreeNode root) {

if(root == null){

return 0;

}

LinkedList<TreeNode> queue = new LinkedList<TreeNode>();

queue.offer(root);

int curNum = 1;

int nextNum = 0;

int level = 1;

while(!queue.isEmpty()){

TreeNode cur = queue.poll();

curNum--;

if(cur.left == null && cur.right == null){

return level;

}

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;

level++;

}

}

return level;

}

**5. Symmetric Tree，这道题一般使用Divide & Conquer方法来做，下面也会介绍。使用层序遍历相对比较复杂，但也是可以的。在这里，我们使用2个queue或者2个stack都可以，分别记录每层的左半部分和右半部分，一层层进行比较。**

// Solution – Classic Model

public boolean isSymmetric(TreeNode root) {

if(root == null){

return true;

}

if(root.left == null && root.right == null){

return true;

}

if(root.left == null || root.right == null){

return false;

}

LinkedList<TreeNode> left = new LinkedList<TreeNode>();

LinkedList<TreeNode> right = new LinkedList<TreeNode>();

left.offer(root.left);

right.offer(root.right);

while(!left.isEmpty() && !right.isEmpty()){

TreeNode l = left.poll();

TreeNode r = right.poll();

if(l.val != r.val){

return false;

}

if(l.left != null && r.right == null || l.left == null && r.right != null){

return false;

}

if(l.right != null && r.left == null || l.right == null && r.left != null){

return false;

}

if(l.left != null && r.right != null){

left.offer(l.left);

right.offer(r.right);

}

if(l.right != null && r.left != null){

left.offer(l.right);

right.offer(r.left);

}

}

return true;

}

**6. Populating Next Right Pointers in Each Node，这道题也可以使用BFS方法来做，与之前题目的区别是，这里把下一层的结点用next指针连接起来，所以下层相当于一个存储了所有下层结点的LinkedList，我们只要获得该LinkedList的头即可依次获得下层的结点，所以可以省去存储下层结点的数据结构。**

// Solution – Classic Model

public void connect(TreeLinkNode root) {

TreeLinkNode curHead = root;

TreeLinkNode nextHead = null;

TreeLinkNode pre = null;

while(curHead != null){

TreeLinkNode cur = curHead;

while(cur != null){

if(cur.left != null){

if(nextHead == null){

nextHead = cur.left;

pre = nextHead;

} else {

pre.next = cur.left;

pre = pre.next;

}

}

if(cur.right != null){

if(nextHead == null){

nextHead = cur.right;

pre = nextHead;

} else {

pre.next = cur.right;

pre = pre.next;

}

}

cur = cur.next;

}

curHead = nextHead;

nextHead = null;

}

}

**Part 3 – Divide & Conquer**

**在LeetCode和LintCode中使用Divide & Conquer算法的题目有：**

**1. Maximum Depth of Binary Tree**

**2. Minimum Depth of Binary Tree**

**3. Balanced Binary Tree**

**4. Binary Tree Maximum Path Sum**

**5. Lowest Common Ancestor**

**6. Merge Sort & Quick Sort**

**1. Maximum Depth of Binary Tree，该解法的重点在于Conquer部分，要清楚返回的是什么，以及如何将其Conquer起来返回给上一层。**

public int maxDepth(TreeNode root) {

if(root == null){

return 0;

}

return Math.max(maxDepth(root.left), maxDepth(root.right)) + 1;

}

**2. Minimum Depth of Binary Tree，该题同样使用Divide & Conquer来解。与上题的区别在于对单孩子结点的对待，由于空孩子方向会返回0，直接取min会产生错误结果。**

public int minDepth(TreeNode root) {

if(root == null){

return 0;

}

if(root.left == null){

return minDepth(root.right) + 1;

}

if(root.right == null){

return minDepth(root.left) + 1;

}

return Math.min(minDepth(root.left), minDepth(root.right)) + 1;

}

**3. Balanced Binary Tree，如果有子树已不平衡，直接返回-1，不需要继续进行比较了。**

public boolean isBalanced(TreeNode root) {

return getHeightAndCheck(root) != -1;

}

private int getHeightAndCheck(TreeNode root){

if(root == null){

return 0;

}

int left = getHeightAndCheck(root.left);

if(left == -1){

return -1;

}

int right = getHeightAndCheck(root.right);

if(right == -1){

return -1;

}

int diff = Math.abs(left - right);

if(diff > 1){

return -1;

}

return Math.max(left, right) + 1;

}

**4. Binary Tree Maximum Path Sum，比较的是双路径，返回的是单路径。**

public int maxPathSum(TreeNode root) {

ArrayList<Integer> res = new ArrayList<Integer>();

res.add(Integer.MIN\_VALUE);

helper(root, res);

return res.get(0);

}

private int helper(TreeNode root, ArrayList<Integer> res){

if(root == null){

return 0;

}

int left = helper(root.left, res);

int right = helper(root.right, res);

int value = Math.max(left, 0) + Math.max(right, 0) + root.val;

if(value > res.get(0)){

res.set(0, value);

}

return Math.max(Math.max(left, right), 0) + root.val;

}

**5. Lowest Common Ancestor，从下向上Conquer，当遇到第一个包含2个所给结点的父结点时，返回该结点即可。**

public TreeNode lowestCommonAncestor(TreeNode root, TreeNode A, TreeNode B) {

if(root == null || root == A || root == B){

return root;

}

TreeNode left = lowestCommonAncestor(root.left, A, B);

TreeNode right = lowestCommonAncestor(root.right, A, B);

if(left != null && right != null){

return root;

}

if(left != null){

return left;

}

if(right != null){

return right;

}

return null;

}

**6. Merge Sort & Quick Sort，这两个排序算法都是利用Divide & Conquer最经典的例子。Merge Sort是先局部有序再整体有序，而Quick Sort是先整体有序再局部有序。由于Merge Sort需要一个拷贝数组的过程，所以速度不及Quick Sort。但两种排序算法中的思想都是非常重要的，在很多题中都会用到，所以在此提及。**

**Merge Sort: 由于是先局部有序再整体有序，所以要先调用两次mergeSort()之后再调用merge()将已排序的两个子数组合并。还需要注意需要一个辅助数组aux[]以及在merge时，对一个数组已经结束时的处理。**

**Quick Sort: 由于是先整体有序再局部有序，所以要先调用partision()根据pivot将原数组化为两个字数组，在调用两次quickSort()对子数组进行排序。我们默认left指针所对应的元素即为pivot元素，注意下标的处理。**

**代码如下：**

public class MergeAndQuickReview {

public static void main(String[] args) {

int[] array = {3, 6, 1, 5, 4, 2, 8, 7};

printArray(array);

mergeSort(array);

printArray(array);

int[] array2 = {3, 6, 1, 5, 4, 2, 8, 7};

printArray(array2);

quickSort(array2);

printArray(array2);

}

private static void mergeSort(int[] array) {

mergeSort(array, 0, array.length - 1);

}

private static void mergeSort(int[] array, int left, int right) {

if(left >= right){

return;

}

int mid = left + (right - left) / 2;

mergeSort(array, left, mid);

mergeSort(array, mid + 1, right);

merge(array, left, mid, right);

}

private static void merge(int[] array, int left, int mid, int right) {

int[] aux = new int[array.length];

for(int k = left; k <= right; k++){

aux[k] = array[k];

}

int subBegin1 = left;

int subBegin2 = mid + 1;

for(int k = left; k <= right; k++){

if(subBegin1 > mid){

array[k] = aux[subBegin2++];

} else if(subBegin2 > right){

array[k] = aux[subBegin1++];

} else if(more(aux[subBegin1], aux[subBegin2])){

array[k] = aux[subBegin2++];

} else {

array[k] = aux[subBegin1++];

}

}

}

private static void quickSort(int[] array) {

quickSort(array, 0, array.length - 1);

}

private static void quickSort(int[] array, int left, int right) {

if(left >= right){

return;

}

int partisionIndex = partision(array, left, right);

quickSort(array, left, partisionIndex - 1);

quickSort(array, partisionIndex + 1, right);

}

private static int partision(int[] array, int left, int right) {

int i = left;

int j = right + 1;

while(true){

while(more(array[left], array[++i])){

if(i == right){

break;

}

}

while(more(array[--j], array[left])){

if(j == left){

break;

}

}

if(i >= j){

break;

}

exchange(array, i, j);

}

exchange(array, left, j);

return j;

}

private static void exchange(int[] array, int i, int j) {

int temp = array[i];

array[i] = array[j];

array[j] = temp;

}

private static boolean more(int i, int j) {

return i > j;

}

private static void printArray(int[] array) {

for(int x : array){

System.out.print(x + " ");

}

System.out.println();

}

}

**Part 4 – BST**

**在LeetCode和LintCode中BST相关的题目有：**

**1. Unique Binary Search Tree**

**2. Unique Binary Search Tree II**

**3. Convert Sorted Array to Binary Search Tree**

**4. Convert Sorted List to Binary Search Tree**

**5. Binary Search Tree Iterator**

**6. Validate Binary Search Tree**

**7. Recover Binary Search Tree**

**8. Insert Node in a Binary Search Tree**

**9. Search Range in Binary Search Tree**

**10. Remove Node in Binary Search Tree**

**切记，基本所有与BST有关的题目，都要用到一个BST的性质，那就是BST中序遍历有序性。**

**1. Unique Binary Search Tree，其实这是一道动态规划的题目，放在这只是保证BST题目的完整性。由于这道题符合卡特兰常数的模型，所以直接使用动态规划计算该卡特兰常数即可，计算时注意下标的Corner Case。**

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];

}

**2. Unique Binary Search Tree II，从1至n中选定一个作为根结点，然后递归处理得到所有左、右子树可能的根，将其做排列组合与根结点相连即可。**

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 k = left; k <= right; k++){

ArrayList<TreeNode> leftList = helper(left, k - 1);

ArrayList<TreeNode> rightList = helper(k + 1, right);

for(int i = 0; i < leftList.size(); i++){

for(int j = 0; j < rightList.size(); j++){

TreeNode root = new TreeNode(k);

root.left = leftList.get(i);

root.right = rightList.get(j);

res.add(root);

}

}

}

return res;

}

**3. Convert Sorted Array to Binary Search Tree，递归每次取中点作为根结点即可。**

  public TreeNode sortedArrayToBST(int[] num) {

if(num == null || num.length == 0){

return null;

}

return helper(num, 0, num.length - 1);

}

private TreeNode helper(int[] num, int left, int right){

if(left > right){

return null;

}

int mid = left + (right - left) / 2;

TreeNode root = new TreeNode(num[mid]);

root.left = helper(num, left, mid - 1);

root.right = helper(num, mid + 1, right);

return root;

}

**4. Convert Sorted List to Binary Search Tree，巧妙的利用了一个nextRoot数组来模拟树的中序遍历过程，只有当一个TreeNode root被创建的时候，nextRoot才会移动。**

public TreeNode sortedListToBST(ListNode head) {

if(head == null){

return null;

}

int count = 0;

ListNode cur = head;

while(cur.next != null){

count++;

cur = cur.next;

}

ArrayList<ListNode> nextRoot = new ArrayList<ListNode>();

nextRoot.add(head);

return helper(nextRoot, 0, count);

}

private TreeNode helper(ArrayList<ListNode> nextRoot, int start, int end){

if(start > end){

return null;

}

int mid = start + (end - start) / 2;

TreeNode left = helper(nextRoot, start, mid - 1);

TreeNode root = new TreeNode(nextRoot.get(0).val);

root.left = left;

nextRoot.set(0, nextRoot.get(0).next);

root.right = helper(nextRoot, mid + 1, end);

return root;

}

**5. Binary Search Tree Iterator，由于每次要使用next()返回下一个最小值，所以相当于中序遍历，我们使用一个stack来模拟中序遍历即可。**

public class BSTIterator {

LinkedList<TreeNode> stack;

TreeNode current;

public BSTIterator(TreeNode root) {

stack = new LinkedList<TreeNode>();

current = root;

}

/\*\* @return whether we have a next smallest number \*/

public boolean hasNext() {

return current != null || !stack.isEmpty();

}

/\*\* @return the next smallest number \*/

public int next() {

while(current != null){

stack.push(current);

current = current.left;

}

TreeNode node = stack.pop();

current = node.right;

return node.val;

}

}

**6. Validate Binary Search Tree，第一种解法同样根据BST中序遍历有序的特性，来判断该BST是否为合法的BST。第二种使用分支定界法，根据BST左子树的所有值必须小于root结点值，右子树的所有值必须大于root结点值的性质来进行判断。**

// 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);

}

// 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);

}

**7. Recover Binary Search Tree，同样根据中序遍历有序性找到违反规则的结点进行修正。**

public void recoverTree(TreeNode root) {

ArrayList<TreeNode> res = new ArrayList<TreeNode>();

ArrayList<TreeNode> pre = new ArrayList<TreeNode>();

pre.add(null);

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);

}

}

pre.set(0, root);

helper(root.right, pre, res);

}

**8. Insert Node in a Binary Search Tree，分为递归和非递归两种插入方法。**

**递归方法：root == null说明找到插入点，直接返回node；根据node.val和root.val之间的关系，选择连接在左面还是右面；为找到null结点前，返回root，不影响原始结构。**

**非递归方法：首先找到合适插入位置的父结点pre，然后根据node.val和pre.val之间的关系，将结点插入到合适的位置即可。**

// Solution 1 - Recursion

public TreeNode insertNode(TreeNode root, TreeNode node) {

if(root == null){

return node;

}

if(node.val < root.val){

root.left = insertNode(root.left, node);

} else {

root.right = insertNode(root.right, node);

}

return root;

}

public TreeNode insertNode(TreeNode root, TreeNode node) {

if(root == null){

root = node;

return root;

}

TreeNode cur = root;

TreeNode pre = null;

while(cur != null){

pre = cur;

if(node.val < cur.val){

cur = cur.left;

} else {

cur = cur.right;

}

}

if(pre != null){

if(node.val < pre.val){

pre.left = node;

} else {

pre.right = node;

}

}

return root;

}

**9. Search Range in Binary Search Tree，该题很简单，递归即可。两个判断可提高效率。**

public ArrayList<Integer> searchRange(TreeNode root, int k1, int k2) {

ArrayList<Integer> res = new ArrayList<Integer>();

helper(root, k1, k2, res);

return res;

}

private void helper(TreeNode root, int k1, int k2, ArrayList<Integer> res){

if(root == null){

return;

}

if(root.val > k1){

helper(root.left, k1, k2, res);

}

if(k1 <= root.val && root.val <= k2){

res.add(root.val);

}

if(root.val < k2){

helper(root.right, k1, k2, res);

}

}

**10. Remove Node in Binary Search Tree，删除情况很复杂，**[**http://www.mathcs.emory.edu/~cheung/Courses/171/Syllabus/9-BinTree/BST-delete.html**](http://www.mathcs.emory.edu/~cheung/Courses/171/Syllabus/9-BinTree/BST-delete.html)**可供参考。**

public TreeNode removeNode(TreeNode root, int value) {

TreeNode dummy = new TreeNode(0);

dummy.left = root;

TreeNode parent = findParent(dummy, root, value);

TreeNode node;

if(parent.left != null && parent.left.val == value){

node = parent.left;

} else if(parent.right != null && parent.right.val == value){

node = parent.right;

} else {

return dummy.left;

}

deleteNode(parent, node);

return dummy.left;

}

private TreeNode findParent(TreeNode parent, TreeNode current, int value){

if(current == null){

return parent;

}

if(current.val == value){

return parent;

}

if(current.val > value){

return findParent(current, current.left, value);

} else {

return findParent(current, current.right, value);

}

}

private void deleteNode(TreeNode parent, TreeNode node){

if(node.right == null){

if(parent.left == node){

parent.left = node.left;

} else {

parent.right = node.left;

}

} else {

TreeNode temp = node.right;

TreeNode last = node;

while(temp.left != null){

last = temp;

temp = temp.left;

}

if(last.left == temp){

last.left = temp.right;

} else {

last.right = temp.right;

}

if(parent.left == node){

parent.left = temp;

} else {

parent.right = temp;

}

temp.left = node.left;

temp.right = node.right;

}

}

**Part 4 – 杂鱼**

**1. Same Tree，DFS的一种。注意if语句排他性的巧妙应用。**

public boolean isSameTree(TreeNode p, TreeNode q) {

if(p == null && q == null){

return true;

}

if(p == null || q == null){

return false;

}

if(p.val != q.val){

return false;

}

return isSameTree(p.left, q.left) && isSameTree(p.right, q.right);

}

**2. Symmetric Tree，DFS的一种。注意if语句排他性的巧妙应用。**

public boolean isSymmetric(TreeNode root) {

if(root == null){

return true;

}

return helper(root.left, root.right);

}

private boolean helper(TreeNode p, TreeNode q){

if(p == null && q == null){

return true;

}

if(p == null || q == null){

return false;

}

if(p.val != q.val){

return false;

}

return helper(p.left, q.right) && helper(p.right, q.left);

}

**3. Path Sum，注意最后有一边符合要求即可，所以最后return的语句使用“或”的关系。**

public boolean hasPathSum(TreeNode root, int sum) {

if(root == null){

return false;

}

if(root.left == null && root.right == null && root.val == sum){

return true;

}

return hasPathSum(root.left, sum - root.val) || hasPathSum(root.right, sum - root.val);

}

**4. Path Sum II，DFS的一种。非空判断不是在进入递归前，就是在刚进入递归时，总之要有一个对空的判断，不然会有空指针异常。**

  public ArrayList<ArrayList<Integer>> pathSum(TreeNode root, int sum) {

ArrayList<ArrayList<Integer>> res = new ArrayList<ArrayList<Integer>>();

if(root == null){

return res;

}

ArrayList<Integer> item = new ArrayList<Integer>();

helper(root, sum, item, res);

return res;

}

private void helper(TreeNode root, int sum, ArrayList<Integer> item, ArrayList<ArrayList<Integer>> res){

item.add(root.val);

if(root.left == null && root.right == null && root.val == sum){

res.add(new ArrayList<Integer>(item));

}

if(root.left != null){

helper(root.left, sum - root.val, item, res);

}

if(root.right != null){

helper(root.right, sum - root.val, item, res);

}

item.remove(item.size() - 1);

}

**5.** **Sum Root to Leaf Numbers，递归到的结点分2种情况：结点为叶子结点，将总和加入到最终结果中去；结点非叶子结点，继续递归其子结点即可。**

public int sumNumbers(TreeNode root) {

ArrayList<Integer> res = new ArrayList<Integer>();

res.add(0);

if(root == null){

return res.get(0);

}

helper(root, 0, res);

return res.get(0);

}

private void helper(TreeNode root, int sum, ArrayList<Integer> res){

int value = sum \* 10 + root.val;

if(root.left == null && root.right == null){

res.set(0, res.get(0) + value);

return;

}

if(root.left != null){

helper(root.left, value, res);

}

if(root.right != null){

helper(root.right, value, res);

}

}

**6. Flatten Binary Tree to Linked List，相当于一个先序遍历。用pre保存上一个访问的结点，然后对当前结点进行处理。注意要先保存一下right子结点，不然其信息会丢失。**

public void flatten(TreeNode root) {

ArrayList<TreeNode> pre = new ArrayList<TreeNode>();

pre.add(null);

helper(root, pre);

}

private void helper(TreeNode root, ArrayList<TreeNode> pre){

if(root == null){

return;

}

TreeNode right = root.right;

if(pre.get(0) != null){

pre.get(0).left = null;

pre.get(0).right = root;

}

pre.set(0, root);

helper(root.left, pre);

helper(right, pre);

}