

深度优先遍历 二叉树 图类似



广度优先



**Remove Nth Node From End of List**

Given linked list: 1->2->3->4->5->null, and n = 2.

After removing the second node from the end, the linked list becomes 1->2->3->5->null.

简单题， 使用快慢指针解决此题，需要注意最后删除的是否为头节点。让快指针先走n步，直至快指针走到终点，找到需要删除节点之前的一个节点，改变node->next域即可。

class Solution {

public:

ListNode\* removeNthFromEnd(ListNode\* head, int n) {

if (!head->next) return NULL;

ListNode \*pre = head, \*cur = head;

for (int i = 0; i < n; ++i) cur = cur->next;

if (!cur) return head->next;

while (cur->next) {

cur = cur->next;

pre = pre->next;

}

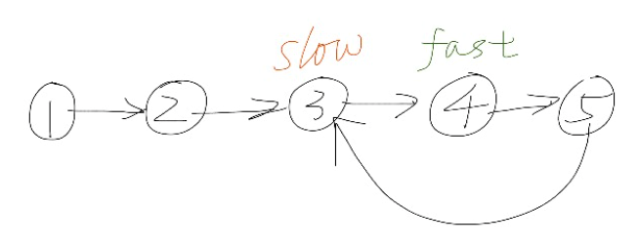
pre->next = pre->next->next;

return head;

}

};

链表环：快慢指针



public class Solution {

public boolean hasCycle(ListNode head) {

if (head == null || head.next == null) {

return false;

}

ListNode slow = head;

ListNode fast = head;

while (fast.next != null && fast.next.next != null) {

slow = slow.next;

fast = fast.next.next;

if (slow == fast) {

return true;

}

}

return false;

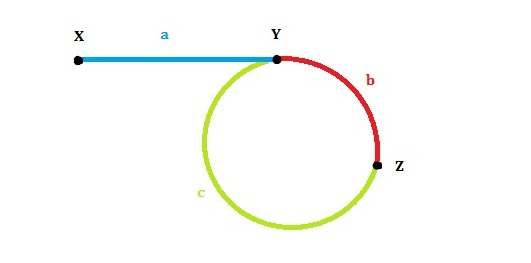
}

}

**扩展问题https://www.cnblogs.com/hiddenfox/p/3408931.html**

在网上搜集了一下这个问题相关的一些问题，思路开阔了不少，总结如下：

1. 环的长度是多少？



第一次相遇时slow走过的距离：a+b，fast走过的距离：a+b+c+b。

**因为fast的速度是slow的两倍，所以fast走的距离是slow的两倍，有 2(a+b) = a+b+c+b，可以得到a=c（这个结论很重要！）。**

也就是说，从一开始到二者第一次相遇，慢指针的步数就等于环的长度。

1. 如何找到环中第一个节点（即Linked List Cycle II）？

我们已经得到了结论a=c，那么让两个指针分别从X和Z开始走，每次走一步，那么正好会在Y相遇！也就是环的第一个节点。

1. 如何将有环的链表变成单链表（解除环）？

将c段中Y点之前的那个节点与Y的链接切断即可。

4. 如何判断两个单链表是否有交点？如何找到第一个相交的节点？

首先我们看下面这张图：

**判断两个单链表是否相交并求出相交的第一结点**

将一个链表首尾相接，然后判断另外一个链表是否有环，如果有环，则两个链表相交。那么求第一个交点则求出有环的的那个链表的环结点即是。

Given a singly linked list *L*: *L\_0→\_L\_1→…→\_L\_\_n*-1→*L\_n,  
reorder it to: \_L\_0→\_L\_\_n*→*L\_1→\_L\_\_n*-1→*L\_2→\_L\_\_n*-2→…

You must do this in-place without altering the nodes' values.

For example,  
Given {1,2,3,4}, reorder it to {1,4,2,3}.

**题解 - 反转链表后归并**

1. 找中点：我在九章算法模板的基础上增加了对head->next的异常检测，增强了鲁棒性。
2. 反转：非常精炼的模板，记牢！
3. 合并：也可使用九章提供的模板，思想是一样的，需要注意left, right和dummy三者的赋值顺序，不能更改任何一步。

public void reorderList(ListNode head) {

if (head == null || head.next == null) return;

// 找中点

ListNode slow = head, fast = head.next;

while (fast != null && fast.next != null) {

slow = slow.next;

fast = fast.next.next;

}

ListNode rHead = slow.next;

slow.next = null;

// 从中点开始，反转后半段

ListNode prev = null;

while (rHead != null) {

ListNode temp = rHead.next;

rHead.next = prev;

prev = rHead;

rHead = temp;

}

// merge two list

rHead = prev;

ListNode lHead = head;

while (lHead != null && rHead != null) {

ListNode temp1 = lHead.next;

lHead.next = rHead;

rHead = rHead.next;

lHead.next.next = temp1;

//连续两个next看着累，最好再定义一个前驱节点看着清楚

lHead = temp1;

}

}

[Sort List](https://leetcode.com/problems/sort-list/) 方法类似

**归并排序(快慢指针求中间节点)**

除了遍历链表求得总长外，还可使用看起来较为巧妙的技巧如「快慢指针」，快指针每次走两步，慢指针每次走一步，最后慢指针所指的节点即为中间节点。

public ListNode sortList(ListNode head) {

// write your code here

if (head == null || head.next == null) return head;

ListNode mid = findMid(head);

ListNode head1 = head;

ListNode head2 = mid.next;

mid.next = null;

ListNode left = sortList(head1);

ListNode right = sortList(head2);

return merge(left, right);

}

// find mid

public ListNode findMid(ListNode head) {

if (head == null) return null;

ListNode fast = head.next;

ListNode slow = head;

while (fast != null && fast.next != null) {

fast = fast.next.next;

slow = slow.next;

}

return slow;

}

// merge

public ListNode merge(ListNode head1, ListNode head2) {

ListNode dummy = new ListNode(0);

ListNode head = dummy;

while (head1 != null || head2 != null) {

int a = head1 == null ? Integer.MAX\_VALUE : head1.val;

int b = head2 == null ? Integer.MAX\_VALUE : head2.val;

if (a < b) {

head.next = new ListNode(a);

if (head1 != null) head1 = head1.next;

} else {

head.next = new ListNode(b);

if (head2 != null) head2 = head2.next;

}

head = head.next;

}

return dummy.next;

}

}

**前序遍历二叉树**

def preorderTraversal(self, root):

if root == None:

return []

return [root.val] + self.preorderTraversal(root.left) \

+ self.preorderTraversal(root.right)

层次遍历二叉树广搜的基础题，使用一个队列保存每层的节点即可。出队和将子节点入队的实现使用 for 循环，将每一轮的节点输出。

public class Solution {

public List<List<Integer>> levelOrder(TreeNode root) {

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

if (root == null) return result;

Queue<TreeNode> q = new LinkedList<TreeNode>();

q.offer(root);

while (!q.isEmpty()) {

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

int qSize = q.size();

for (int i = 0; i < qSize; i++) {

TreeNode node = q.poll();

list.add(node.val);

// push child node into queue

if (node.left != null) q.offer(node.left);

if (node.right != null) q.offer(node.right);

}

result.add(new ArrayList<Integer>(list));

}

return result;

}

}

**Maximum Depth of Binary Tree**

public int maxDepth(TreeNode root) {

// write your code here

if (root == null) {

return 0;

}

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

}

**平衡树**的定义是两子树的深度差最大不超过1，显然使用递归进行分析较为方便。既然使用递归，

public class Solution {

public boolean isBalanced(TreeNode root) {

return maxDepth(root) != -1;

}

private int maxDepth(TreeNode root) {

if (root == null) return 0;

int leftDepth = maxDepth(root.left);

int rightDepth = maxDepth(root.right);

if (leftDepth == -1 || rightDepth == -1 ||

Math.abs(leftDepth - rightDepth) > 1) {

return -1;

}

return 1 + Math.max(leftDepth, rightDepth);

}

}

# Binary Tree Maximum Path Sum

For example: Given the below binary tree,

1

/ \

2 3

Return 6.

**思路：首先我们分析一下对于指定某个节点为根时，最大的路径和有可能是哪些情况。第一种是左子树的路径加上当前节点，第二种是右子树的路径加上当前节点，第三种是左右子树的路径加上当前节点（相当于一条横跨当前节点的路径），第四种是只有自己的路径。乍一看似乎以此为条件进行自下而上递归就行了，然而这四种情况只是用来计算以当前节点根的最大路径，如果当前节点上面还有节点，那它的父节点是不能累加第三种情况的。所以我们要计算两个最大值，一个是当前节点下最大路径和，另一个是如果要连接父节点时最大的路径和。我们用前者更新全局最大量，用后者返回递归值就行了。**

**Java代码：**

[复制代码](javascript:void(0);)

public class Solution {

private int max = Integer.MIN\_VALUE;

public int maxPathSum(TreeNode root) {

helper(root);

return max;

}

public int helper(TreeNode root) {

if(root == null) return 0;

int left = helper(root.left);

int right = helper(root.right);

//连接父节点的最大路径是一、二、四这三种情况的最大值

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

//当前节点的最大路径是一、二、三、四这四种情况的最大值

int currMax = Math.max(currSum, left + right + root.val);

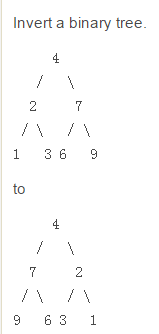
//用当前最大来更新全局最大

max = Math.max(currMax, max);

return currSum;

}

}

[翻转二叉树](http://www.cnblogs.com/grandyang/p/4572877.html) 

TreeNode\* invertTree(TreeNode\* root) {

if (!root) return NULL;

TreeNode \*tmp = root->left;

root->left = invertTree(root->right);

root->right = invertTree(tmp);

return root;

}

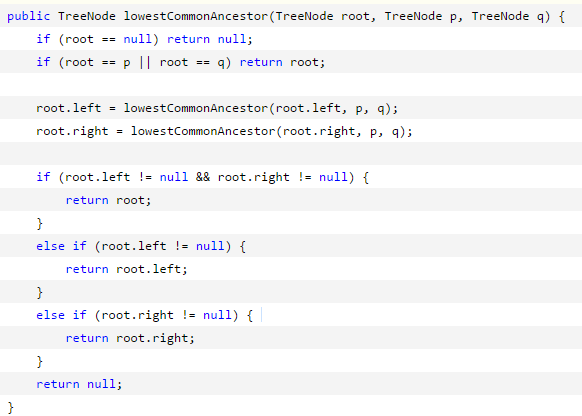
我们需要注意到**最小共有祖先**的特点：

(1)若节点M为节点P,Q的最小共有祖先，那么M的左右后代都不是P,Q的共有祖先；M的祖先都是P,Q的共有祖先；

也就是说，对于节点P,Q，给定树的根节点一定是他们的共有祖先，而且，若M为P,Q共有祖先，则分为两种情况，第一，M不是最小祖先，那么，M的左右儿子中有且只有一个为P,Q的共有祖先；第二，M是最小祖先，后代情况见特点(1)；

很简单的思路就是看两个值在root的哪边：

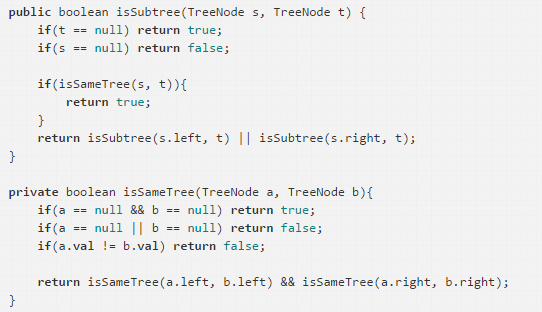
* 两个值都在左边，则LCA在左边
* 两个值都在右边，则LCA在右边
* 一个在左一个在右，则说明LCA就是当前的root节点。



通过一棵二叉树的前序和中序排列来得出它的树形结构。

具体解决方法是：  
通过先序遍历找到第一个点作为根节点，在中序遍历中找到根节点并记录index。  
因为中序遍历中根节点左边为左子树，所以可以记录左子树的长度并在先序遍历中依据这个长度找到左子树的区间，用同样方法可以找到右子树的区间。  
递归的建立好左子树和右子树就好。

判断 T2是否是 T1的子树，首先应该在 T1中找到 T2的根节点，找到根节点后两棵子树必须完全相同。



二叉搜索树的定义及简介在 [Binary Search Trees](http://algorithm.yuanbin.me/zh-hans/basics_data_structure/binary_search_tree.html) 中已经有所介绍。简单来说就是当前节点的值大于等于左子结点的值，而小于右子节点的值。

**Insert Node in a Binary Search Tree**

public TreeNode insertNode(TreeNode root, TreeNode node) {

if (root == null) {

return node;

}

if (root.val > node.val) {

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

} else {

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

}

return root;

}

**Minimum Absolute Difference in BST**

题意为找**任意**两个节点间绝对值差的最小值，根据二叉搜索树的特性，中序遍历即得有序数组，找出相邻两数的最小差值即为解。

private int min = Integer.MAX\_VALUE;

private TreeNode prev = null;

public int getMinimumDifference(TreeNode root) {

if (root == null) return min;

getMinimumDifference(root.left);

if (prev != null) {

min = Math.min(min, root.val - prev.val);

}

prev = root;

getMinimumDifference(root.right);

return min;

}

判断是否二叉搜索树

public boolean isValidBST(TreeNode root) {

if (root == null) return true;

return helper(root, Long.MIN\_VALUE, Long.MAX\_VALUE);

}

private boolean helper(TreeNode root, long lower, long upper) {

if (root == null) return true;

// System.out.println("root.val = " + root.val + ", lower = " + lower + ", upper = " + upper);

// left node value < root node value < right node value

if (root.val >= upper || root.val <= lower) return false;

boolean isLeftValidBST = helper(root.left, lower, root.val);

boolean isRightValidBST = helper(root.right, root.val, upper);

return isLeftValidBST && isRightValidBST;

}

方法2 中序遍历以后应该是有序数组。判断是否符合

**Convert Sorted Array to Binary Search Tree**

public TreeNode sortedArrayToBST(int[] A) {

if (A == null || A.length == 0) return null;

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

}

private TreeNode helper(int[] nums, int start, int end) {

if (start > end) return null;

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

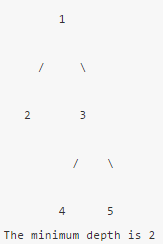
TreeNode root = new TreeNode(nums[mid]);

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

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

return root;

}

**Minimum Depth of Binary Tree** 

public int minDepth(TreeNode root) {

if (root == null) return 0;

int leftDepth = minDepth(root.left);

int rightDepth = minDepth(root.right);

// current node is not leaf node

if (root.left == null) {

return 1 + rightDepth;

} else if (root.right == null) {

return 1 + leftDepth;

}

return 1 + Math.min(leftDepth, rightDepth);

}

『最小』栈，要求在栈的基础上实现可以在 O(1)*O*(1) 的时间内找出最小值，一般这种 O(1)*O*(1)的实现往往就是哈希表或者哈希表的变体，这里简单起见可以另外克隆一个栈用以跟踪当前栈的最小值。

这道最小栈跟原来的栈相比就是多了一个功能，可以返回该栈的最小值。使用两个栈来实现，一个栈来按顺序存储push进来的数据，另一个用来存出现过的最小值。

**Maximum Subarray**求最大子数组和，即求区间和的最大值，

**动态规划1(区间和)**

求最大/最小这种字眼往往都可以使用动态规划求解，此题为单序列动态规划。我们可以先求出到索引 i 的子数组和，然后用子数组和的最大值减去最小值，最后返回最大值即可。用这种动态规划需要注意初始化条件和求和顺序。

public int maxSubArray(ArrayList<Integer> nums) {

// -1 is not proper for illegal input

if (nums == null || nums.isEmpty()) return -1;

int sum = 0, minSum = 0, maxSub = Integer.MIN\_VALUE;

for (int num : nums) {

minSum = Math.min(minSum, sum);

sum += num;

maxSub = Math.max(maxSub, sum - minSum);

}

return maxSub;

}