**前序遍历**

|  |
| --- |
| class Solution {  public:  vector<int> preorderTraversal(TreeNode\* root) {  vector<int> res;  stack<TreeNode\*> s;  TreeNode\* p = root;  while(!s.empty() || p){  if(p != nullptr){  res.push\_back(p->val);  s.push(p);  p = p->left;  }else {  p = s.top();  s.pop();  p = p->right;  }  }  return res;  }  }; |

**中序遍历**

|  |
| --- |
| class Solution {  public:  vector<int> inorderTraversal(TreeNode\* root) {  vector<int> res;  stack<TreeNode\*> s;  TreeNode\* p = root;  while(!s.empty() || p){  if(p != nullptr){  s.push(p);  p = p->left;  }else{  p = s.top();  s.pop();  res.push\_back(p->val);  p = p->right;  }  }  return res;  }  }; |

**后序遍历**

|  |
| --- |
| class Solution {  public:  vector<int> postorderTraversal(TreeNode\* root) {  vector<int> res;  stack<TreeNode\*> s;  TreeNode\* p = root;  TreeNode\* r = nullptr;  while(!s.empty() || p){  if(p){  s.push(p);  p = p->left;  }else{  p = s.top();  if(p->right == nullptr || p->right == r){  res.push\_back(p->val);  s.pop();  r = p;  p = nullptr;  }else  p = p->right;  }  }  return res;  }  }; |

**98. 验证二叉搜索树**

给定一个二叉树，判断其是否是一个有效的二叉搜索树。

假设一个二叉搜索树具有如下特征：

节点的左子树只包含小于当前节点的数。

节点的右子树只包含大于当前节点的数。

所有左子树和右子树自身必须也是二叉搜索树。

输入: 输入:

2 5

/ \ / \

1 3 1 4

输出: true / \

3 6

输出: false

解释: 输入为: [5,1,4,null,null,3,6]。

根节点的值为 5 ，但是其右子节点值为 4

|  |
| --- |
| class Solution {  public boolean isValidBST(TreeNode root) {  return validate(root, Long.MIN\_VALUE, Long.MAX\_VALUE);  }  public boolean validate(TreeNode node, long min, long max) {  if (node == null) {  return true;  }  if (node.val <= min || node.val >= max) {  return false;  }  return validate(node.left, min, node.val) && validate(node.right, node.val, max);  }  }  检查是否是合法二叉搜索树。对于所有结点，检查是否满足大于左子树，小于右子树。 |

**315. 计算右侧小于当前元素的个数**

给定一个整数数组 nums，按要求返回一个新数组 counts。数组 counts 有该性质： counts[i] 的值是 nums[i] 右侧小于 nums[i] 的元素的数量。

输入：nums = [5,2,6,1]

输出：[2,1,1,0]

解释：

5 的右侧有 2 个更小的元素 (2 和 1)

2 的右侧仅有 1 个更小的元素 (1)

6 的右侧有 1 个更小的元素 (1)

1 的右侧有 0 个更小的元素

|  |
| --- |
| class Solution {  public:  vector<int> countSmaller(vector<int>& nums) {  if (nums.empty()) {  return {};  }  vector<int> res;  vector<int> sortedNums;  for (int i = nums.size() - 1; i >= 0; i--) {  auto iter = lower\_bound(sortedNums.begin(), sortedNums.end(), nums[i]);  int pos = iter - sortedNums.begin();  res.push\_back(pos);  sortedNums.insert(iter, nums[i]);  }  reverse(res.begin(), res.end());  return res;  }  }; |

当一个队伍，插入一个新同学的时候，如何知道有几个人身高比他矮呢？

如果这个队伍的人并不是按照身高从低到高的顺序排列的话，那没办法，只能一个一个数了。如果这个队伍是有序的呢，只要找到插入的位置，就能直接计算出来比他矮的同学的个数了。

在这个题目里，要想知道一个数后面比它小的数有多少个，只要找到新数字插入的位置就能判断出有几个比它小的（假设比它后面的数字都已经排好序了）。来一个例子。

[1,3,6,1,2,3]

input 3, output: [3] -> 3 左边有 0 个数

input 2, output: [2,3] -> 2 左边有 0 个数

input 1, output: [1,2,3] -> 1 左边有 0 个数

input 6, output: [1,2,3,6] -> 6 左边有 3 个数

input 3', output: [1,2,3',3,6] -> 3' 左边有 2 个数

input 1', output: [1',1,2,3',3,6] -> 1' 左边有 0 个数

可以看到，在不断插入的过程中，能根据插入的位置判断出比它小的数有多少个。虽然插入的位置查找速度是 logn 的，但是插入过程却要移动元素，复杂度是 n，这个成本非常高。

如果有一种办法，查找速度很快 logn，插入的速度也很快，O(1)，那多好。链表不行，虽然插入是 O(1)，但是查找却无法做到 log(n).

那就只剩下树了。把上面的数组换成 bst (binary-search-tree)，一切就好办了。只要维护好这棵树就行。这里就不解释太多了，其它的答案都有很详细的说明。

|  |
| --- |
| class Solution {  struct Node {  shared\_ptr<Node> left;  shared\_ptr<Node> right;  int val;  int count = 0; // 左子树节点的个数  Node(int val) : val(val) {}  };  public:  vector<int> countSmaller(vector<int>& nums) {  shared\_ptr<Node> root;  vector<int> res(nums.size());  for (int i = nums.size() - 1; i >= 0; --i) {  root = insert(root, nums[i], res, i);  }  return res;  }  shared\_ptr<Node> insert(shared\_ptr<Node> root, int val, vector<int>& res, int index) {  if (!root) {  return make\_shared<Node>(val);  }  auto& r = res;  if (val <= root->val) {  root->count++;  root->left = insert(root->left, val, res, index);  } else {  r[index] += root->count + 1;  root->right = insert(root->right, val, res, index);  }  return root;  }  }; |

**105. 从前序与中序遍历序列构造二叉树**

根据一棵树的前序遍历与中序遍历构造二叉树。

注意: 你可以假设树中没有重复的元素。

前序遍历 preorder = [3,9,20,15,7]

中序遍历 inorder = [9,3,15,20,7]

返回如下的二叉树：

3

/ \

9 20

/ \

15 7

|  |
| --- |
| class Solution {  public:  TreeNode\* buildTree(vector<int>& preorder, vector<int>& inorder)  {  if (preorder.size() == 0) {  return NULL;  }  TreeNode\* root = new TreeNode(preorder[0]);  // 通过根结点preorder[0]划分中序遍历序列  int pos = find(inorder.begin(), inorder.end(), preorder[0]) - inorder.begin();  // 递归构造左子树  vector<int> prev(preorder.begin() + 1, preorder.begin() + 1 + pos);  vector<int> inv(inorder.begin(), inorder.begin() + pos);  root->left = buildTree(prev, inv);  // 递归构造右子树  prev.assign(preorder.begin() + 1 + pos, preorder.end());  inv.assign(inorder.begin() + pos + 1, inorder.end());  root->right = buildTree(prev, inv);  //返回根结点  return root;  }  }; |

**222. 完全二叉树的节点个数**

给出一个完全二叉树，求出该树的节点个数。

说明：完全二叉树的定义如下：在完全二叉树中，除了最底层节点可能没填满外，其余每层节点数都达到最大值，并且最下面一层的节点都集中在该层最左边的若干位置。若最底层为第 h 层，则该层包含 1~ 2h 个节点。

输入:

1

/ \

2 3

/ \ /

4 5 6

输出: 6

|  |
| --- |
| class Solution {  public:  int countNodes(TreeNode\* root) {  if (root == nullptr) {  return 0;  }  int leftLevel = CountLevel(root->left);  int rightLevel = CountLevel(root->right);  if (leftLevel == rightLevel) {  return (1 << leftLevel) + countNodes(root->right);  } else {  return (1 << rightLevel) + countNodes(root->left);  }  return 0;  }  private:  int CountLevel(TreeNode\* node) {  if (node == nullptr) {  return 0;  }  int level = 0;  while (node != nullptr) {  level++;  node = node->left;  }  return level;  }  }; |

**230. 二叉搜索树中第K小的元素**

给定一个二叉搜索树，编写一个函数 kthSmallest 来查找其中第 k 个最小的元素。

说明：你可以假设 k 总是有效的，1 ≤ k ≤ 二叉搜索树元素个数。

输入: root = [3,1,4,null,2], k = 1

3

/ \

1 4

\

2

输出: 1

输入: root = [5,3,6,2,4,null,null,1], k = 3

5

/ \

3 6

/ \

2 4

/

1

输出: 3

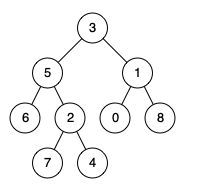
|  |
| --- |
| class Solution {  int count = 0;  int res = Integer.MAX\_VALUE;  public int kthSmallest(TreeNode root, int k) {  help(root, k);  return res;  }  private void help(TreeNode root, int k){  if(root == null){  return;  }  help(root.left, k);  count ++;  if(count == k){  res = root.val;  return;  }  help(root.right, k);  }  }  中序遍历，输出第k个数字。 |

**236. 二叉树的最近公共祖先**

给定一个二叉树, 找到该树中两个指定节点的最近公共祖先。

百度百科中最近公共祖先的定义为：“对于有根树 T 的两个结点 p、q，最近公共祖先表示为一个结点 x，满足 x 是 p、q 的祖先且 x 的深度尽可能大（一个节点也可以是它自己的祖先）。”

例如，给定如下二叉树: root = [3,5,1,6,2,0,8,null,null,7,4]



输入: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1

输出: 3

解释: 节点 5 和节点 1 的最近公共祖先是节点 3。

输入: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4

输出: 5

解释: 节点 5 和节点 4 的最近公共祖先是节点 5。因为根据定义最近公共祖先节点可以为节点本身。

|  |
| --- |
| public class Solution {//所有的递归的返回值有4种可能性，null、p、q、公共祖先  public TreeNode LowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {  if (root == null) {//当遍历到叶结点后就会返回null  return root;  }  //当找到p或者q的是时候就会返回pq，值得一提的是，如果公共祖先是自己（pq）  //并不要寻找另一个，执行前序遍历会先找上面的，后找下面，直接返回公共祖先  if (root == p || root == q) {  return root;  }  //返回的结点进行保存，可能是null,也可能是pq，还可能是公共祖先  TreeNode left = LowestCommonAncestor(root.left, p, q);  TreeNode right = LowestCommonAncestor(root.right, p, q);  //如果左右都存在，就说明pq都出现了，这就是公共祖先  // 此时不用考虑公共祖先是自己的情况，因为上面已经做过判断了  if (left != null && right != null) {  return root;  //否则我们返回已经找到的那个值（存储在left，与right中），p或者q  //还有一种可能就是，由下面返回的公共祖先，并将这个值一路返回到最表层  } else if (left != null) {  return left;  } else if (right != null) {  return right;  }  return null;  }  } |

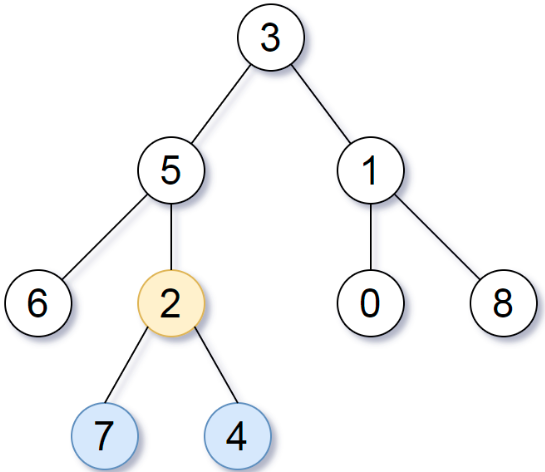
**865/1123. 最深叶节点的最近公共祖先**

给你一个有根节点的二叉树，找到它最深的叶节点的最近公共祖先。

叶节点 是二叉树中没有子节点的节点

树的根节点的 深度 为 0，如果某一节点的深度为 d，那它的子节点的深度就是 d+1

如果我们假定 A 是一组节点 S 的 最近公共祖先，S 中的每个节点都在以 A 为根节点的子树中，且 A 的深度达到此条件下可能的最大值。



输入：root = [3,5,1,6,2,0,8,null,null,7,4]

输出：[2,7,4]

解释：

我们返回值为 2 的节点，在图中用黄色标记。

在图中用蓝色标记的是树的最深的节点。

注意，节点 6、0 和 8 也是叶节点，但是它们的深度是 2 ，而节点 7 和 4 的深度是 3 。

输入：root = [1]

输出：[1]

解释：根节点是树中最深的节点，它是它本身的最近公共祖先。

输入：root = [0,1,3,null,2]

输出：[2]

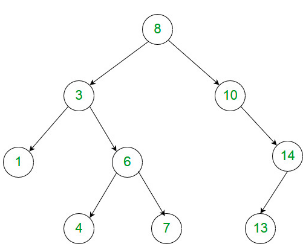
解释：树中最深的叶节点是 2 ，最近公共祖先是它自己。

|  |
| --- |
| class Solution {  int maxLayer\_ {0};  TreeNode\* ans\_ {nullptr};  public:  TreeNode\* lcaDeepestLeaves(TreeNode\* root) {  if (root == nullptr) {  return nullptr;  }  int left = Depth(root->left);  int right = Depth(root->right);  if (left == right) {  return root;  } else if (left > right) {  return lcaDeepestLeaves(root->left);  } else {  return lcaDeepestLeaves(root->right);  }  }  int Depth(TreeNode\* root) {  if (root == nullptr) {  return 0;  }  int left = Depth(root->left);  int right = Depth(root->right);  return 1 + max(left, right);  }  }; |

**1026. 节点与其祖先之间的最大差值**

给定二叉树的根节点 root，找出存在于不同节点 A 和 B 之间的最大值 V，其中 V = |A.val - B.val|，且 A 是 B 的祖先。

（如果 A 的任何子节点之一为 B，或者 A 的任何子节点是 B 的祖先，那么我们认为 A 是 B 的祖先）



输入：[8,3,10,1,6,null,14,null,null,4,7,13]

输出：7

解释：

我们有大量的节点与其祖先的差值，其中一些如下：

|8 - 3| = 5

|3 - 7| = 4

|8 - 1| = 7

|10 - 13| = 3

在所有可能的差值中，最大值 7 由 |8 - 1| = 7 得出。

|  |
| --- |
| class Solution {  public int maxAncestorDiff(TreeNode root) {  int left = maxAncestorDiff(root.left, root.val, root.val);  int right = maxAncestorDiff(root.right, root.val, root.val);  return left > right ? left : right;  }  public int maxAncestorDiff(TreeNode root, int max, int min){  if(root == null){  return 0;  }  if(root.val > max){  max = root.val;  }  else if(root.val < min){  min = root.val;  }  if(root.left == null && root.right == null){  return max - min;  }  int left = maxAncestorDiff(root.left, max, min);  int right = maxAncestorDiff(root.right, max, min);  return left > right ? left : right;  }  } |

|  |
| --- |
| class Solution {  public:  int maxAncestorDiff(TreeNode\* root) {  if (root == nullptr) {  return 0;  }  int val = root->val;  if (root->left != nullptr) {  Dfs(root->left, { val });  }  if (root->right != nullptr) {  Dfs(root->right, { val });  }  return diff\_;  }  void Dfs(TreeNode\* node, vector<int> ancestor) {  if (node == nullptr) {  return;  }  int val = node->val;  for (auto an : ancestor) {  int diff = abs(an - val);  if (diff > diff\_) {  diff\_ = diff;  }  }  ancestor.push\_back(val);  Dfs(node->left, ancestor);  Dfs(node->right, ancestor);  }  private:  int diff\_{0};  }; |

**剑指 Offer 51. 数组中的逆序对**

在数组中的两个数字，如果前面一个数字大于后面的数字，则这两个数字组成一个逆序对。输入一个数组，求出这个数组中的逆序对的总数。

输入: [7,5,6,4]

输出: 5

|  |
| --- |
| class Solution {  public:  int dfs(vector<int>& nums, int l, int r)  {  if (r - l <= 1) {  return 0;  }  int mid = (l + r) / 2;  int res = dfs(nums, l, mid) + dfs(nums, mid, r);  sort(nums.begin() + l, nums.begin() + mid);  for (int i = mid; i < r; ++i) {  // 计算左侧数组中比nums[i]大的数的个数，也就是pos到mid的距离  res += nums.begin() + mid - upper\_bound(nums.begin() + l, nums.begin() + mid, nums[i]);  }  return res;  }  int reversePairs(vector<int>& nums)  {  return dfs(nums, 0, nums.size());  }  }; |

|  |
| --- |
| class Solution {  public:  int dfs(vector<int> & nums,int l,int r){  if(r-l <= 1) return 0;  int mid = (l+r)>>1;  vector<int> tmp;  int res = dfs(nums,l,mid) + dfs(nums,mid,r);  int i = l;  int j = mid;  while(i < mid && j < r){  if(nums[i] > nums[j]){  res += mid - i;  tmp.push\_back(nums[j++]);  }else{  tmp.push\_back(nums[i++]);  }  }  while(i < mid) tmp.push\_back(nums[i++]);  while(j < r) tmp.push\_back(nums[j++]);  copy(tmp.begin(),tmp.end(),nums.begin()+l);  return res;  }  int reversePairs(vector<int>& nums) {  return dfs(nums,0,nums.size());  }  }; |

高级解法：树状数组：

<https://leetcode-cn.com/problems/count-of-smaller-numbers-after-self/solution/shu-zhuang-shu-zu-by-liweiwei1419/>