LeetCode Training Day 7 Tree Traversal

Binary Tree Traverse is a common problem in algorithm. The fundamental of the solution is tree traversal. Let’s first review the definition of tree traversal. There are 3 types of tree traversal, preorder, inorder and postorder. For preorder, you need to visit the current node first then visit the children, this allow you to carry the information from parent to children. Post order is that you have to visit all the children first, then carry the information from children, then based on this information and the current node to determine the next action. Most of the tree problems can be resolved by post order traversal. For middle order, you visit the left subtree first, the current node then the right subtree.

**DFS**

To process the preorder or post, normally we build a recursive function call. The call will go deeper if we have a deeper level of children not visited and return when all the children are empty or after we visited both left and right children. We carry the information to the upper-level function by using reference to variables or return tuple. For C++, reference to variable is preferred and for C# return tuple is preferred.

Such recursive call is also known as DFS (deep first search).

**BFS**

In some case we would like to traverse the tree by level, which is to say we first traverse root, then its two children (if available) then the four grand children, and so on. In this case we will do BFS, which is that we build a queue, put all n the node in the current level in the queue, then process these n nodes to generate next level of children.

**BST**

Please also pay attention to the Binary Search tree, for which, all the nodes in left subtree are less than the parent and all the nodes in right subtree are greater than the parent. For Binary Search Tree, the search may be LOG(N) since based on the value in the parent, we can skip the whole subtree.

**Path and Iterator**

When we traverse along the Binary Tree, the path is a list of nodes from root to current node. This path can also help us to build an iterator to travel to next node. The next node will be in the following cases:

1. If the current node has a right subtree, then the left most node in the right subtree is the next node.
2. If current node does not have right subtree, if the current node is a left node of its parent, then the parent is the next node.
3. If current node is the right child of parent and no right subtree, we need recursively find a parent with #1 or #2 is satisfied.

## 110. Balanced Binary Tree

Easy

Given a binary tree, determine if it is height-balanced.

For this problem, a height-balanced binary tree is defined as:

a binary tree in which the left and right subtrees of *every* node differ in height by no more than 1.

**Example 1:**

A picture containing text, clipart

Description automatically generated

**Input:** root = [3,9,20,null,null,15,7]

**Output:** true

**Example 2:**

A picture containing watch

Description automatically generated

**Input:** root = [1,2,2,3,3,null,null,4,4]

**Output:** false

**Example 3:**

**Input:** root = []

**Output:** true

**Constraints:**

* The number of nodes in the tree is in the range [0, 5000].
* -104 <= Node.val <= 104

### Analysis:

By definition, for each node you need to know the depth of left node and right node, and if any node is not balanced then the final result is false. Because you need to make decision on every node based on their children, the DFS with post order is best.

/// <summary>

/// Leet code #110. Balanced Binary Tree

/// </summary>

bool LeetCodeTree::isBalanced(TreeNode\* root, int& depth)

{

if (root == nullptr)

{

depth = 0;

return true;

}

else

{

int left\_depth = 0;

int right\_depth = 0;

if (!isBalanced(root->left, left\_depth) ||

!isBalanced(root->right, right\_depth))

{

depth = max(left\_depth, right\_depth) + 1;

return false;

}

else

{

depth = max(left\_depth, right\_depth) + 1;

if (abs(left\_depth - right\_depth) <= 1)

{

return true;

}

else

{

return false;

}

}

}

}

/// <summary>

/// Leet Code 110. Balanced Binary Tree

///

/// Easy

///

/// Given a binary tree, determine if it is height-balanced.

///

/// For this problem, a height-balanced binary tree is defined as:

/// a binary tree in which the left and right subtrees of every node

/// differ in height by no more than 1.

///

/// Example 1:

/// Input: root = [3,9,20,null,null,15,7]

/// Output: true

///

/// Example 2:

/// Input: root = [1,2,2,3,3,null,null,4,4]

/// Output: false

///

/// Example 3:

/// Input: root = []

/// Output: true

///

/// Constraints:

/// 1. The number of nodes in the tree is in the range [0, 5000].

/// -10^4 <= Node.val <= 10^4

/// </summary>

bool LeetCodeTree::isBalanced(TreeNode\* root)

{

int depth;

return isBalanced(root, depth);

}

## 107. Binary Tree Level Order Traversal II

Medium

Given the root of a binary tree, return *the bottom-up level order traversal of its nodes' values*. (i.e., from left to right, level by level from leaf to root).

**Example 1:**

A picture containing text, clock, clipart

Description automatically generated

**Input:** root = [3,9,20,null,null,15,7]

**Output:** [[15,7],[9,20],[3]]

**Example 2:**

**Input:** root = [1]

**Output:** [[1]]

**Example 3:**

**Input:** root = []

**Output:** []

**Constraints:**

* The number of nodes in the tree is in the range [0, 2000].
* -1000 <= Node.val <= 1000

### Analysis:

This is a level order traversal, we iterate all nodes in levels and store in a array of array, and then reverse it.

/// <summary>

/// Leet Code 107. Binary Tree Level Order Traversal II

///

/// Medium

///

/// Given the root of a binary tree, return the bottom-up level order

/// traversal of its nodes' values. (i.e., from left to right, level

/// by level from leaf to root).

///

/// Example 1:

/// Input: root = [3,9,20,null,null,15,7]

/// Output: [[15,7],[9,20],[3]]

///

/// Example 2:

/// Input: root = [1]

/// Output: [[1]]

///

/// Example 3:

/// Input: root = []

/// Output: []

///

/// Constraints:

/// 1. The number of nodes in the tree is in the range [0, 2000].

/// 2. -1000 <= Node.val <= 1000

/// </summary>

vector<vector<int>> LeetCodeTree::levelOrderBottom(TreeNode\* root)

{

vector<int> level;

stack<vector<int>> tree\_stack;

vector<vector<int>> result;

queue<TreeNode \*> node\_queue;

if (root != nullptr)

{

node\_queue.push(root);

}

while (!node\_queue.empty())

{

size\_t size = node\_queue.size();

level.clear();

for (size\_t i = 0; i < size; i++)

{

TreeNode \* node = node\_queue.front();

node\_queue.pop();

level.push\_back(node->val);

if (node->left != nullptr)

{

node\_queue.push(node->left);

}

if (node->right != nullptr)

{

node\_queue.push(node->right);

}

}

tree\_stack.push(level);

}

while (!tree\_stack.empty())

{

result.push\_back(tree\_stack.top());

tree\_stack.pop();

}

return result;

}

## 236. Lowest Common Ancestor of a Binary Tree

Medium

Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.

According to the [definition of LCA on Wikipedia](https://en.wikipedia.org/wiki/Lowest_common_ancestor): “The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow **a node to be a descendant of itself**).”

**Example 1:**

Shape, circle

Description automatically generated

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

**Output:** 3

**Explanation:** The LCA of nodes 5 and 1 is 3.

**Example 2:**

Shape, circle

Description automatically generated

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

**Output:** 5

**Explanation:** The LCA of nodes 5 and 4 is 5, since a node can be a descendant of itself according to the LCA definition.

**Example 3:**

**Input:** root = [1,2], p = 1, q = 2

**Output:** 1

**Constraints:**

* The number of nodes in the tree is in the range [2, 105].
* -109 <= Node.val <= 109
* All Node.val are **unique**.
* p != q
* p and q will exist in the tree.

### Analysis:

On every node traverse we track if we found any of the target nodes, for the first time we see both of the target nodes, we return current node as the answer and terminate the search.

/// <summary>

/// Leet code #236. Lowest Common Ancestor of a Binary Tree

/// </summary>

int LeetCodeTree::lowestCommonAncestor(TreeNode\* root, TreeNode\* p, TreeNode\* q,

TreeNode\* &result)

{

int count = 0;

if (root == nullptr) return 0;

if (root == p || root == q) count = 1;

count += lowestCommonAncestor(root->left, p, q, result);

if (count < 2)

{

count += lowestCommonAncestor(root->right, p, q, result);

}

// if first time we see 2 node found, this is the answer.

if (count == 2 && result == nullptr) result = root;

return count;

}

/// <summary>

/// Leet Code 236. Lowest Common Ancestor of a Binary Tree

///

/// Medium

///

/// Given a binary tree, find the lowest common ancestor (LCA) of two

/// given nodes in the tree.

/// \_\_\_\_\_\_\_3\_\_\_\_\_\_

/// / \

/// \_\_\_5\_\_ \_\_\_1\_\_

/// / \ / \

/// 6 2 0 8

/// / \

/// 7 4

/// According to the definition of LCA on Wikipedia: “The lowest common

/// ancestor is defined between two nodes p and q as the lowest node

/// in T that has both p and q as descendants (where we allow a node

/// to be a descendant of itself).”

///

/// Example 1:

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

/// Output: 3

/// Explanation: The LCA of nodes 5 and 1 is 3.

///

/// Example 2:

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

/// Output: 5

/// Explanation: The LCA of nodes 5 and 4 is 5, since a node can be

/// a descendant of itself according to the LCA definition.

///

/// Example 3:

/// Input: root = [1,2], p = 1, q = 2

/// Output: 1

///

/// Constraints:

/// 1. The number of nodes in the tree is in the range [2, 10^5].

/// 2. -10^9 <= Node.val <= 10^9

/// 3. All Node.val are unique.

/// 4. p != q

/// 5. p and q will exist in the tree.

/// </summary>

TreeNode\* LeetCodeTree::lowestCommonAncestor(TreeNode\* root, TreeNode\* p, TreeNode\* q)

{

TreeNode \* result = nullptr;

int count = lowestCommonAncestor(root, p, q, result);

return result;

}

## 235. Lowest Common Ancestor of a Binary Search Tree

Easy

Given a binary search tree (BST), find the lowest common ancestor (LCA) of two given nodes in the BST.

According to the [definition of LCA on Wikipedia](https://en.wikipedia.org/wiki/Lowest_common_ancestor): “The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow **a node to be a descendant of itself**).”

**Example 1:**

Shape, circle

Description automatically generated

**Input:** root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8

**Output:** 6

**Explanation:** The LCA of nodes 2 and 8 is 6.

**Example 2:**

Shape, circle

Description automatically generated

**Input:** root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 4

**Output:** 2

**Explanation:** The LCA of nodes 2 and 4 is 2, since a node can be a descendant of itself according to the LCA definition.

**Example 3:**

**Input:** root = [2,1], p = 2, q = 1

**Output:** 2

**Constraints:**

* The number of nodes in the tree is in the range [2, 105].
* -109 <= Node.val <= 109
* All Node.val are **unique**.
* p != q
* p and q will exist in the BST.

### Analysis:

The tree is binary search tree, if we find any node with value greater than the target, then we know we can skip the right subtree, if we find any node with value less than the target, then we know we can skip the left subtree. Also if we find any node matches one of the p or q, we know it is the answer.

/// <summary>

/// Leet code #235. Lowest Common Ancestor of a Binary Search Tree

/// Given a binary search tree (BST), find the lowest common ancestor (LCA) of two given nodes in the BST.

/// According to the definition of LCA on Wikipedia: "The lowest common ancestor is defined between two nodes

/// v and w as the lowest node in T that has both v and w as descendants (where we allow a node to be a descendant of itself)."

/// \_\_\_\_\_\_\_6\_\_\_\_\_\_

/// / \

/// \_\_\_2\_\_ \_\_\_8\_\_

/// / \ / \

/// 0 4 7 9

/// / \

/// 3 5

/// For example, the lowest common ancestor (LCA) of nodes 2 and 8 is 6. Another example is LCA of nodes 2 and 4 is 2,

/// since a node can be a descendant of itself according to the LCA definition.

/// </summary>

TreeNode\* LeetCodeTree::lowestCommonAncestorBST(TreeNode\* root, TreeNode\* p, TreeNode\* q)

{

TreeNode\* node = root;

while (true)

{

if ((node == nullptr) || (p == nullptr) || (q == nullptr))

{

return nullptr;

}

if ((node->val > p->val) && (node->val > q->val))

{

node = node->left;

}

else if ((node->val < p->val) && (node->val < q->val))

{

node = node->right;

}

else

{

return node;

}

}

}

## 112. Path Sum

Easy

Given the root of a binary tree and an integer targetSum, return true if the tree has a **root-to-leaf** path such that adding up all the values along the path equals targetSum.

A **leaf** is a node with no children.

**Example 1:**

A picture containing text, watch

Description automatically generated

**Input:** root = [5,4,8,11,null,13,4,7,2,null,null,null,1], targetSum = 22

**Output:** true

**Explanation:** The root-to-leaf path with the target sum is shown.

**Example 2:**

A picture containing text, clipart

Description automatically generated

**Input:** root = [1,2,3], targetSum = 5

**Output:** false

**Explanation:** There two root-to-leaf paths in the tree:

(1 --> 2): The sum is 3.

(1 --> 3): The sum is 4.

There is no root-to-leaf path with sum = 5.

**Example 3:**

**Input:** root = [], targetSum = 0

**Output:** false

**Explanation:** Since the tree is empty, there are no root-to-leaf paths.

**Constraints:**

* The number of nodes in the tree is in the range [0, 5000].
* -1000 <= Node.val <= 1000
* -1000 <= targetSum <= 1000

### Analysis:

We need a sum from root to leaf, so first we need to check what is leaf (no children), then we need all the sum from parents, so a preorder traversal is preferred.

/// <summary>

/// Leet code #112. Path Sum

/// Given a binary tree and a sum, determine if the tree has a root-to-leaf path

/// such that adding up all the values along the path equals the given sum.

/// For example:

/// Given the below binary tree and sum = 22,

/// 5

/// / \

/// 4 8

/// / / \

/// 11 13 4

/// / \ \

/// 7 2 1

/// return true, as there exist a root-to-leaf path 5->4->11->2 which sum is 22.

/// </summary>

bool LeetCodeTree::hasPathSum(TreeNode\* root, int sum)

{

if (root == nullptr) return false;

if (root != nullptr)

{

if ((root->left == nullptr) && (root->right == nullptr) && (root->val == sum))

{

return true;

}

if ((root->left != nullptr) && hasPathSum(root->left, sum - root->val))

{

return true;

}

if ((root->right != nullptr) && hasPathSum(root->right, sum - root->val))

{

return true;

}

}

return false;

}

## 101. Symmetric Tree

Easy

Given the root of a binary tree, *check whether it is a mirror of itself* (i.e., symmetric around its center).

**Example 1:**

Shape, arrow, circle

Description automatically generated

**Input:** root = [1,2,2,3,4,4,3]

**Output:** true

**Example 2:**

Shape

Description automatically generated

**Input:** root = [1,2,2,null,3,null,3]

**Output:** false

**Constraints:**

* The number of nodes in the tree is in the range [1, 1000].
* -100 <= Node.val <= 100

**Follow up:** Could you solve it both recursively and iteratively?

### Analysis:

This is a level order traversal, we use BFS to push all the nodes in an array, and use two pointers from both end to check if it mirrors.

/// <summary>

/// Leet Code 101. Symmetric Tree

///

/// Easy

///

/// Given the root of a binary tree, check whether it is a mirror of

/// itself (i.e., symmetric around its center).

///

/// Example 1:

/// Input: root = [1,2,2,3,4,4,3]

/// Output: true

///

/// Example 2:

/// Input: root = [1,2,2,null,3,null,3]

/// Output: false

///

/// Constraints:

/// 1. The number of nodes in the tree is in the range [1, 1000].

/// 2. -100 <= Node.val <= 100

///

/// Follow up: Could you solve it both recursively and iteratively?

/// </summary>

bool LeetCodeTree::isSymmetric(TreeNode\* root)

{

vector<TreeNode\*> tree\_list;

queue<TreeNode\*> tree\_queue;

TreeNode \* node = root;

tree\_queue.push(root);

while (!tree\_queue.empty())

{

size\_t size = tree\_queue.size();

tree\_list.clear();

for (size\_t i = 0; i < size; i++)

{

TreeNode \* node = tree\_queue.front();

tree\_list.push\_back(node);

tree\_queue.pop();

if (node != nullptr)

{

tree\_queue.push(node->left);

tree\_queue.push(node->right);

}

}

int first = 0, last = tree\_list.size() - 1;

while (first < last)

{

if (((tree\_list[first] == nullptr) && (tree\_list[last] != nullptr)) ||

((tree\_list[first] != nullptr) && (tree\_list[last] == nullptr)))

{

return false;

}

if ((tree\_list[first] != nullptr) && (tree\_list[last] != nullptr) &&

(tree\_list[first]->val != tree\_list[last]->val))

{

return false;

}

first++; last--;

}

}

return true;

}

## 230. Kth Smallest Element in a BST

Medium

Given the root of a binary search tree, and an integer k, return *the* kth *smallest value (****1-indexed****) of all the values of the nodes in the tree*.

**Example 1:**

Diagram

Description automatically generated

**Input:** root = [3,1,4,null,2], k = 1

**Output:** 1

**Example 2:**

Diagram

Description automatically generated

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

**Output:** 3

**Constraints:**

* The number of nodes in the tree is n.
* 1 <= k <= n <= 104
* 0 <= Node.val <= 104

**Follow up:** If the BST is modified often (i.e., we can do insert and delete operations) and you need to find the kth smallest frequently, how would you optimize?

### Analysis:

This is an inorder traverse, we can keep on deduct k until 0 when we visited the node, assume we visited X nodes in left tree, in which all the values are less than the current node, then we have already deducted X from K. When K become 0 we no longer update the answer.

For the follow up, we just need to record the total nodes in the sub tree in a parent.

/// <summary>

/// Return kth smallest node in BST

/// </summary>

int LeetCodeTree::findKthSmallest(TreeNode\* root, int &k)

{

if (root == nullptr) return 0;

int value = findKthSmallest(root->left, k);

if (k == 0) return value;

k--;

if (k == 0) return root->val;

value = findKthSmallest(root->right, k);

return value;

}

/// <summary>

/// Leet Code 230. Kth Smallest Element in a BST

///

/// Medium

///

/// Given the root of a binary search tree, and an integer k, return the

/// kth smallest value (1-indexed) of all the values of the nodes in the

/// tree.

///

/// Example 1:

/// Input: root = [3,1,4,null,2], k = 1

/// Output: 1

///

/// Example 2:

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

/// Output: 3

///

/// Constraints:

/// 1. The number of nodes in the tree is n.

/// 2. 1 <= k <= n <= 10^4

/// 3. 0 <= Node.val <= 10^4

///

/// Follow up: If the BST is modified often (i.e., we can do insert and

/// delete operations) and you need to find the kth smallest frequently,

/// how would you optimize?

/// </summary>

int LeetCodeTree::kthSmallest(TreeNode\* root, int k)

{

return findKthSmallest(root, k);

}

## 333. Largest BST Subtree

Medium

Given the root of a binary tree, find the largest subtree, which is also a Binary Search Tree (BST), where the largest means subtree has the largest number of nodes.

A **Binary Search Tree (BST)** is a tree in which all the nodes follow the below-mentioned properties:

* The left subtree values are less than the value of their parent (root) node's value.
* The right subtree values are greater than the value of their parent (root) node's value.

**Note:** A subtree must include all of its descendants.

**Example 1:**

**A picture containing clipart

Description automatically generated**

**Input:** root = [10,5,15,1,8,null,7]

**Output:** 3

**Explanation:** The Largest BST Subtree in this case is the highlighted one. The return value is the subtree's size, which is 3.

**Example 2:**

**Input:** root = [4,2,7,2,3,5,null,2,null,null,null,null,null,1]

**Output:** 2

**Constraints:**

* The number of nodes in the tree is in the range [0, 104].
* -104 <= Node.val <= 104

### Analysis:

This is a very classic complex problem in tree traversal. First we need to determine the current subtree is a Binary Search Tree. This is done after we visited all the children, so it is post-order. To determine it is a BST, we need to know both of left subtree and right subtree is BST, so we need a boolean to be returned in each subtree to indicate if this is a BST. Then to check itself is BST or not, we need to have the max\_val from left tree and min\_val from right tree, this means each sub tree should return its min\_val and max\_val. Finally if it is a BST we need to return the size of all the subtree and track the maximum size so far in the result.

/// <summary>

/// Leet code #333. Largest BST Subtree

/// </summary>

bool LeetCodeTree::checkBSTSubtree(TreeNode\* root, int& min\_val, int& max\_val, int& size)

{

if (root == nullptr)

{

size = 0;

return true;

}

bool isBST = true;

// we need to initial new variable to avoid poplue carried in variable, if you use tuple

// or structure to return values, no such issue

int left\_min = root->val, left\_max = root->val, left\_size = 0;

if (root->left != nullptr)

{

// check should go first to make sure subtree is searched

bool ret = checkBSTSubtree(root->left, left\_min, left\_max, left\_size);

isBST = ret && isBST;

isBST = isBST && (left\_max < root->val);

}

int right\_min = root->val, right\_max = root->val, right\_size = 0;

if (root->right != nullptr)

{

// check should go first to make sure subtree is searched

bool ret = checkBSTSubtree(root->right, right\_min, right\_max, right\_size);

isBST = ret && isBST;

isBST = isBST && (right\_min > root->val);

}

min\_val = min(left\_min, right\_min); max\_val = max(right\_min, right\_max);

if (isBST)

{

size = 1 + left\_size + right\_size;

return true;

}

else

{

size = max(left\_size, right\_size);

return false;

}

}

/// <summary>

/// Leet code #333. Largest BST Subtree

///

/// Given a binary tree, find the largest subtree which is a Binary Search Tree

/// (BST), where largest means subtree with largest number of nodes in it.

/// Note:

/// A subtree must include all of its descendants.

/// Here's an example:

///

/// 10

/// / \

/// 5 15

/// / \ \

/// 1 8 7

/// The Largest BST Subtree in this case is the highlighted one.

/// The return value is the subtree's size, which is 3.

/// Hint:

/// 1. You can recursively use algorithm similar to 98. Validate Binary

/// Search Tree at each node of the tree,

/// which will result in O(nlogn) time complexity.

/// Follow up:

/// Can you figure out ways to solve it with O(n) time complexity?

/// </summary>

int LeetCodeTree::largestBSTSubtree(TreeNode\* root)

{

int min\_val, max\_val, size;

checkBSTSubtree(root, min\_val, max\_val, size);

return size;

}

## 117. Populating Next Right Pointers in Each Node II

Medium

Given a binary tree

struct Node {

int val;

Node \*left;

Node \*right;

Node \*next;

}

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to NULL.

Initially, all next pointers are set to NULL.

**Example 1:**

Diagram

Description automatically generated

**Input:** root = [1,2,3,4,5,null,7]

**Output:** [1,#,2,3,#,4,5,7,#]

**Explanation:** Given the above binary tree (Figure A), your function should populate each next pointer to point to its next right node, just like in Figure B. The serialized output is in level order as connected by the next pointers, with '#' signifying the end of each level.

**Example 2:**

**Input:** root = []

**Output:** []

**Constraints:**

* The number of nodes in the tree is in the range [0, 6000].
* -100 <= Node.val <= 100

**Follow-up:**

* You may only use constant extra space.
* The recursive approach is fine. You may assume implicit stack space does not count as extra space for this problem.

### Analysis:

Normally we do by traversing the tree in level order, and store the nodes in same leve in a queue, but due to constraint we need to use constant space, we can not use queue. However, the tree node itself, give us a next pointer, so we just need to track the header in the previous line and the header in the current line.

/// <summary>

/// Leet code #117. Populating Next Right Pointers in Each Node II

///

/// Given a binary tree

/// struct TreeLinkNode {

/// int val;

/// TreeLinkNode \*left, \*right, \*next;

/// TreeLinkNode(int x) : val(x), left(NULL), right(NULL), next(NULL) {}

/// };

///

/// Follow up for problem "Populating Next Right Pointers in Each Node".

///

/// What if the given tree could be any binary tree? Would your previous

/// solution still work?

///

/// Note:

/// You may only use constant extra space.

///

/// For example,

/// Given the following binary tree,

///

/// 1

/// / \

/// 2 3

/// / \ \

/// 4 5 7

/// After calling your function, the tree should look like:

///

/// 1 -> NULL

/// / \

/// 2 -> 3 -> NULL

/// / \ \

/// 4-> 5 -> 7 -> NULL

/// </summary>

void LeetCodeTree::connectRightII(TreeLinkNode \*root)

{

TreeLinkNode\* prev\_head = nullptr, \*prev\_ptr = nullptr;

TreeLinkNode\* curr\_head, \*curr\_ptr;

prev\_head = root;

while (prev\_head)

{

prev\_ptr = prev\_head;

curr\_head = nullptr;

curr\_ptr = nullptr;

while (prev\_ptr != nullptr)

{

connectRight(curr\_head, curr\_ptr, prev\_ptr->left);

connectRight(curr\_head, curr\_ptr, prev\_ptr->right);

prev\_ptr = prev\_ptr->next;

}

prev\_head = curr\_head;

}

return;

}

## 173. Binary Search Tree Iterator

Medium

Implement an iterator over a binary search tree (BST). Your iterator will be initialized with the root node of a BST.

Calling next() will return the next smallest number in the BST.

**Example:**

**Shape

Description automatically generated**

BSTIterator iterator = new BSTIterator(root);

iterator.next(); // return 3

iterator.next(); // return 7

iterator.hasNext(); // return true

iterator.next(); // return 9

iterator.hasNext(); // return true

iterator.next(); // return 15

iterator.hasNext(); // return true

iterator.next(); // return 20

iterator.hasNext(); // return false

**Note:**

* next() and hasNext() should run in average O(1) time and uses O(*h*) memory, where *h* is the height of the tree.
* You may assume that next() call will always be valid, that is, there will be at least a next smallest number in the BST when next() is called.

### Analysis:

This is another typical scenario in binary tree problem, we are looking for next node in the binary tree. The next node is either the left most node in the right subtree, or if there is no right subtree then it is the parent node with current node as left children.

During the iteration, we push all the parent if we traverse left side on the path.

/// <summary>

/// LeetCode #173. Binary Search Tree Iterator

/// Implement an iterator over a binary search tree (BST). Your iterator

/// will be initialized with the root node of a BST.

/// Calling next() will return the next smallest number in the BST.

/// Note: next() and hasNext() should run in average O(1) time and uses

/// O(h) memory, where h is the height of the tree.

/// </summary>

/\*\*

\* Your BSTIterator will be called like this:

\* BSTIterator i = BSTIterator(root);

\* while (i.hasNext()) cout << i.next();

\*/

class BSTIterator

{

private:

stack<TreeNode\*> m\_TreeStack;

public:

/// <summary>

/// Constructor, which will lead to the smallest child.

/// </summary>

BSTIterator(TreeNode\* root)

{

TreeNode\* node = root;

while (node != nullptr)

{

m\_TreeStack.push(node);

node = node->left;

}

}

/// <summary>

/// return whether we have a next smallest number

/// </summary>

bool hasNext()

{

return (!m\_TreeStack.empty());

}

/// <summary>

/// return the next smallest number

/// </summary>

int next()

{

TreeNode\* node = m\_TreeStack.top();

int value = node->val;

m\_TreeStack.pop();

if (node->right != nullptr)

{

node = node->right;

while (node != nullptr)

{

m\_TreeStack.push(node);

node = node->left;

}

}

return value;

}

};

# Advanced Problem

## 222. Count Complete Tree Nodes

Medium

Given a **complete** binary tree, count the number of nodes.

**Note:**

**Definition of a complete binary tree from**[**Wikipedia**](http://en.wikipedia.org/wiki/Binary_tree#Types_of_binary_trees)**:**  
In a complete binary tree every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible. It can have between 1 and 2h nodes inclusive at the last level h.

**Example:**

**Input:**

1

/ \

2 3

/ \ /

4 5 6

**Output:** 6

### Analysis:

If we check the depth for the left most path and right most path and they are equal then it is a full tree. If not we check the left subtree and the right sub tree to see if they are complete. The complexity is O(Log(N) \*depth).

/// <summary>

/// Leet code #222. Count Complete Tree Nodes

/// Given a complete binary tree, count the number of nodes.

/// Definition of a complete binary tree from Wikipedia:

/// In a complete binary tree every level, except possibly the last, is completely filled,

/// and all nodes in the last level are as far left as possible. It can have between 1 and 2h nodes

/// inclusive at the last level h.

/// </summary>

int LeetCodeTree::countCompleteTreeNodes(TreeNode\* root)

{

int left\_height = 0, right\_height = 0;

TreeNode \* node = root;

while (node != nullptr)

{

left\_height += 1;

node = node->left;

}

node = root;

while (node != nullptr)

{

right\_height += 1;

node = node->right;

}

int count = 0;

if (left\_height == right\_height)

{

count = (1 << left\_height) - 1;

}

else

{

count = 1;

count += countCompleteTreeNodes(root->left);

count += countCompleteTreeNodes(root->right);

}

return count;

}

## 297. Serialize and Deserialize Binary Tree

Hard

Serialization is the process of converting a data structure or object into a sequence of bits so that it can be stored in a file or memory buffer, or transmitted across a network connection link to be reconstructed later in the same or another computer environment.

Design an algorithm to serialize and deserialize a binary tree. There is no restriction on how your serialization/deserialization algorithm should work. You just need to ensure that a binary tree can be serialized to a string and this string can be deserialized to the original tree structure.

**Example:**

You may serialize the following tree:

1

/ \

2 3

/ \

4 5

as "[1,2,3,null,null,4,5]"

**Clarification:** The above format is the same as [how LeetCode serializes a binary tree](https://leetcode.com/faq/#binary-tree). You do not necessarily need to follow this format, so please be creative and come up with different approaches yourself.

**Note:**Do not use class member/global/static variables to store states. Your serialize and deserialize algorithms should be stateless.

### Analysis:

We can serialize the tree in level order sequence, make sure empty node should have a special tag.

/// <summary>

/// Leet code 297. Serialize and Deserialize Binary Tree

/// Serialization is the process of converting a data structure or object

/// into a sequence of bits so that it can be stored in a file or memory

/// buffer, or transmitted across a network connection link to be

/// reconstructed later in the same or another computer environment.

///

/// Design an algorithm to serialize and deserialize a binary tree. There

/// is no restriction on how your serialization/deserialization algorithm

/// should work. You just need to ensure that a binary tree can be

/// serialized to a string and this string can be deserialized to the

/// original tree structure.

///

/// For example, you may serialize the following tree

/// 1

/// / \

/// 2 3

/// / \

/// 4 5

///

/// as "[1,2,3,null,null,4,5]", just the same as how LeetCode OJ serializes

/// a binary tree.

/// You do not necessarily need to follow this format, so please be

/// creative and come up with different approaches yourself.

/// Note: Do not use class member/global/static variables to store states.

/// Your serialize and deserialize algorithms should be stateless.

/// </summary>

/// <summary>

/// Encodes a tree to a single string.

/// </summary>

/// <param name="root">the root</param>

/// <returns>The string</returns>

string LeetCodeTree::serialize(TreeNode\* root)

{

string result = "";

queue<TreeNode \*> queue;

if (root != nullptr) queue.push(root);

while (!queue.empty())

{

TreeNode \* node = queue.front();

queue.pop();

if (!result.empty()) { result.push\_back(','); }

if (node == nullptr)

{

result.append("null");

}

else

{

result.append(std::to\_string(node->val));

queue.push(node->left);

queue.push(node->right);

}

}

while (true)

{

if ((result.size() > 4) && (result.substr(result.size() - 4) == "null"))

{

result.erase(result.size() - 4);

}

else if ((result.size() > 1) && (result.substr(result.size() - 1) == ","))

{

result.erase(result.size() - 1);

}

else

{

break;

}

}

return "[" + result + "]";

}

/// <summary>

/// Decodes your encoded data to tree.

/// </summary>

/// <param name="data">the string data</param>

/// <returns>The root</returns>

TreeNode\* LeetCodeTree::deserialize(string data)

{

queue<TreeNode\*> input\_queue;

queue<TreeNode\*> output\_queue;

string number;

for (size\_t i = 0; i < data.size(); i++)

{

if ((data[i] == '[') || (data[i] == ',') || data[i] == ']' ||

isspace(data[i]))

{

if (number.size() != 0)

{

if (number == "null")

{

input\_queue.push(nullptr);

}

else

{

input\_queue.push(new TreeNode(std::stoi(number)));

}

number.clear();

}

}

else

{

number.push\_back(data[i]);

}

}

TreeNode\* root = nullptr;

TreeNode\* node = nullptr;

while (!input\_queue.empty())

{

if (output\_queue.empty())

{

root = input\_queue.front();

input\_queue.pop();

node = root;

output\_queue.push(node);

}

else

{

node = output\_queue.front();

output\_queue.pop();

if (node != nullptr)

{

if (!input\_queue.empty())

{

node->left = input\_queue.front();

if (node->left != nullptr) node->left->parent = node;

input\_queue.pop();

}

if (!input\_queue.empty())

{

node->right = input\_queue.front();

if (node->right != nullptr) node->right->parent = node;

input\_queue.pop();

}

output\_queue.push(node->left);

output\_queue.push(node->right);

}

}

}

return root;

}

## 124. Binary Tree Maximum Path Sum

Hard

Given a **non-empty** binary tree, find the maximum path sum.

For this problem, a path is defined as any sequence of nodes from some starting node to any node in the tree along the parent-child connections. The path must contain **at least one node** and does not need to go through the root.

**Example 1:**

**Input:** [1,2,3]

**1**

**/ \**

**2** **3**

**Output:** 6

**Example 2:**

**Input:** [-10,9,20,null,null,15,7]

  -10

   / \

  9  **20**

**/  \**

**15   7**

**Output:** 42

### Analysis:

The maximum path can exist under any node, the path can be either left leg, right leg or with two legs including itself. You just need to bring up the maximum sum from a two-leg path, and a one leg path.

/// <summary>

/// Leet code #124. Binary Tree Maximum Path Sum

/// </summary>

void LeetCodeTree::maxPathSum(TreeNode\* root, int &max\_path\_sum, int&max\_path\_loop)

{

if (root == nullptr)

{

max\_path\_sum = 0;

max\_path\_loop = INT\_MIN;

}

else

{

int max\_path\_sum\_left, max\_path\_loop\_left;

maxPathSum(root->left, max\_path\_sum\_left, max\_path\_loop\_left);

int max\_path\_sum\_right, max\_path\_loop\_right;

maxPathSum(root->right, max\_path\_sum\_right, max\_path\_loop\_right);

max\_path\_sum = max(max\_path\_sum\_left + root->val, max\_path\_sum\_right + root->val);

max\_path\_sum = max(max\_path\_sum, root->val);

max\_path\_loop = max(max\_path\_loop\_left, max\_path\_loop\_right);

max\_path\_loop = max(max\_path\_loop, root->val + max\_path\_sum\_left + max\_path\_sum\_right);

max\_path\_loop = max(max\_path\_loop, max\_path\_sum);

}

}

/// <summary>

/// Leet code #124. Binary Tree Maximum Path Sum

///

/// Given a binary tree, find the maximum path sum.

/// For this problem, a path is defined as any sequence of nodes from some

/// starting node to any node in the tree along the parent-child connections.

/// The path must contain at least

/// one node and does not need to go through the root.

/// For example:

/// Given the below binary tree,

/// 1

/// / \

/// 2 3

/// Return 6.

/// Explanation:

/// The max\_path must come from the left direct path + self, the right direct

/// path + self and the maximum left loop path and maximum right loop path.

/// </summary>

int LeetCodeTree::maxPathSum(TreeNode\* root)

{

int max\_path\_loop = 0;

int max\_path\_sum = 0;

maxPathSum(root, max\_path\_sum, max\_path\_loop);

return max\_path\_loop;

}

## 99. Recover Binary Search Tree

Hard

Two elements of a binary search tree (BST) are swapped by mistake.

Recover the tree without changing its structure.

**Example 1:**

**Input:** [1,3,null,null,2]

  1

  /

 3

  \

  2

**Output:** [3,1,null,null,2]

  3

  /

 1

  \

  2

**Example 2:**

**Input:** [3,1,4,null,null,2]

3

/ \

1 4

  /

  2

**Output:** [2,1,4,null,null,3]

2

/ \

1 4

  /

 3

**Follow up:**

* A solution using O(*n*) space is pretty straight forward.
* Could you devise a constant space solution?

### Analysis:

We can traverse the binary sort tree, and compare the previous node and current node, if only one inverse relationship found say A > B, the we must swap (A,B), but if two inverse relationship found, say, A > B and C > D, we must swap (A, D), this is by condition only two nodes needs to be swapped.

/// <summary>

/// Find the two disordered nodes in the binary search tree

/// </summary>

void LeetCodeTree::recoverTreeII(TreeNode\* root, TreeNode\* &prev,

TreeNode\* &first, TreeNode\* &second)

{

if (root == nullptr) return;

if (root->left != nullptr)

{

recoverTreeII(root->left, prev, first, second);

}

if ((prev != nullptr) && (prev->val > root->val))

{

if (first == nullptr)

{

first = prev;

}

second = root;

}

prev = root;

if (root->right != nullptr)

{

recoverTreeII(root->right, prev, first, second);

}

}

/// <summary>

/// Leet code #99. Recover Binary Search Tree

/// Two elements of a binary search tree (BST) are swapped by mistake.

/// Recover the tree without changing its structure.

/// Note:

/// A solution using O(n) space is pretty straight forward.

/// Could you devise a constant space solution?

/// </summary>

void LeetCodeTree::recoverTreeII(TreeNode\* root)

{

TreeNode \*prev = nullptr, \*first = nullptr, \*second = nullptr;

recoverTreeII(root, prev, first, second);

if ((first != nullptr) && (second != nullptr))

{

swap(first->val, second->val);

}

}

## 1373. Maximum Sum BST in Binary Tree

Hard

Given a **binary tree** root, the task is to return the maximum sum of all keys of **any** sub-tree which is also a Binary Search Tree (BST).

Assume a BST is defined as follows:

* The left subtree of a node contains only nodes with keys **less than** the node's key.
* The right subtree of a node contains only nodes with keys **greater than** the node's key.
* Both the left and right subtrees must also be binary search trees.

**Example 1:**

A picture containing electronics

Description automatically generated

**Input:** root = [1,4,3,2,4,2,5,null,null,null,null,null,null,4,6]

**Output:** 20

**Explanation:** Maximum sum in a valid Binary search tree is obtained in root node with key equal to 3.

**Example 2:**

Diagram

Description automatically generated

**Input:** root = [4,3,null,1,2]

**Output:** 2

**Explanation:** Maximum sum in a valid Binary search tree is obtained in a single root node with key equal to 2.

**Example 3:**

**Input:** root = [-4,-2,-5]

**Output:** 0

**Explanation:** All values are negatives. Return an empty BST.

**Example 4:**

**Input:** root = [2,1,3]

**Output:** 6

**Example 5:**

**Input:** root = [5,4,8,3,null,6,3]

**Output:** 7

**Constraints:**

* Each tree has at most 40000 nodes..
* Each node's value is between [-4 \* 10^4 , 4 \* 10^4].

### Analysis:

This is a typical problem that you need to bring back enough information from subtree traverse. When traverse subtree, you need to bring back the minimum value, maximum value, which is to determine the high level tree is a BST, whether the subtree is a BST or not, and the max\_sum from any subtree which is a BST.

/// <summary>

/// Leet code #1373. Maximum Sum BST in Binary Tree

/// </summary>

int LeetCodeTree::maxSumBST(TreeNode\* root, int &min\_val, int&max\_val, bool &is\_bst, int &max\_sum)

{

int sum = 0;

if (root == nullptr)

{

max\_sum = 0;

return sum;

}

if (root->left != nullptr)

{

int left\_min = INT\_MAX;

int left\_max = INT\_MIN;

bool left\_is\_bst = true;

int left\_max\_sum = 0;

int left\_sum = maxSumBST(root->left, left\_min, left\_max, left\_is\_bst, left\_max\_sum);

is\_bst = is\_bst && left\_is\_bst && (left\_max < root->val);

max\_sum = max(max\_sum, left\_max\_sum);

min\_val = min(min\_val, left\_min);

max\_val = max(max\_val, left\_max);

sum += left\_sum;

}

if (root->right != nullptr)

{

int right\_min = INT\_MAX;

int right\_max = INT\_MIN;

bool right\_is\_bst = true;

int right\_max\_sum = 0;

int right\_sum = maxSumBST(root->right, right\_min, right\_max, right\_is\_bst, right\_max\_sum);

is\_bst = is\_bst && right\_is\_bst && (right\_min > root->val);

max\_sum = max(max\_sum, right\_max\_sum);

min\_val = min(min\_val, right\_min);

max\_val = max(max\_val, right\_max);

sum += right\_sum;

}

sum += root->val;

min\_val = min(min\_val, root->val);

max\_val = max(max\_val, root->val);

if (is\_bst)

{

max\_sum = max(max\_sum, sum);

}

return sum;

}

/// <summary>

/// Leet code #1373. Maximum Sum BST in Binary Tree

///

/// Hard

///

/// Given a binary tree root, the task is to return the maximum sum

/// of all keys of any sub-tree which is also a Binary Search Tree (BST).

///

/// Assume a BST is defined as follows:

///

/// The left subtree of a node contains only nodes with keys less than

/// the node's key.

/// The right subtree of a node contains only nodes with keys greater

/// than the node's key.

/// Both the left and right subtrees must also be binary search trees.

///

/// Example 1:

/// Input: root = [1,4,3,2,4,2,5,null,null,null,null,null,null,4,6]

/// Output: 20

/// Explanation: Maximum sum in a valid Binary search tree is obtained

/// in root node with key equal to 3.

///

/// Example 2:

/// Input: root = [4,3,null,1,2]

/// Output: 2

/// Explanation: Maximum sum in a valid Binary search tree is obtained

/// in a single root node with key equal to 2.

///

/// Example 3:

/// Input: root = [-4,-2,-5]

/// Output: 0

/// Explanation: All values are negatives. Return an empty BST.

///

/// Example 4:

/// Input: root = [2,1,3]

/// Output: 6

///

/// Example 5:

///

/// Input: root = [5,4,8,3,null,6,3]

/// Output: 7

/// Constraints:

/// 1. Each tree has at most 40000 nodes..

/// 2. Each node's value is between [-4 \* 10^4 , 4 \* 10^4].

/// </summary>

int LeetCodeTree::maxSumBST(TreeNode\* root)

{

int min\_val = INT\_MAX;

int max\_val = INT\_MIN;

bool is\_bst = true;

int max\_sum = 0;

maxSumBST(root, min\_val, max\_val, is\_bst, max\_sum);

return max\_sum;

}

## 431. Encode N-ary Tree to Binary Tree

Hard

Design an algorithm to encode an N-ary tree into a binary tree and decode the binary tree to get the original N-ary tree. An N-ary tree is a rooted tree in which each node has no more than N children. Similarly, a binary tree is a rooted tree in which each node has no more than 2 children. There is no restriction on how your encode/decode algorithm should work. You just need to ensure that an N-ary tree can be encoded to a binary tree and this binary tree can be decoded to the original N-nary tree structure.

*Nary-Tree input serialization is represented in their level order traversal, each group of children is separated by the null value (See following example).*

For example, you may encode the following 3-ary tree to a binary tree in this way:

Diagram

Description automatically generated

**Input:** root = [1,null,3,2,4,null,5,6]

Note that the above is just an example which *might or might not* work. You do not necessarily need to follow this format, so please be creative and come up with different approaches yourself.

**Constraints:**

* The height of the n-ary tree is less than or equal to 1000
* The total number of nodes is between [0, 10^4]
* Do not use class member/global/static variables to store states. Your encode and decode algorithms should be stateless.

### Analysis:

The idea is to keep only the first child as left child and keep the brother as right child.

/// <summary>

/// Leet code #431. Encode N-ary Tree to Binary Tree

///

/// Design an algorithm to encode an N-ary tree into a binary tree and decode

/// the binary tree to get the original N-ary tree. An N-ary tree is a rooted

/// tree in which each node has no more than N children. Similarly, a binary

/// tree is a rooted tree in which each node has no more than 2 children. There

/// is no restriction on how your encode/decode algorithm should work. You just

/// need to ensure that an N-ary tree can be encoded to a binary tree and this

/// binary tree can be decoded to the original N-nary tree structure.

///

/// For example, you may encode the following 3-ary tree to a binary tree in

/// this way:

///

/// Note that the above is just an example which might or might not work. You

/// do not necessarily need to follow this format, so please be creative and

/// come up with different approaches yourself.

///

/// Note:

///

/// N is in the range of [1, 1000]

/// Do not use class member/global/static variables to store states. Your

/// encode and decode algorithms should be stateless.

/// or if B is true, or if both A and B are true.

/// </summary>

class NaryTreeBinaryCodec

{

private:

// Encodes an n-ary tree to a binary tree.

TreeNode\* encode(queue<Node\*> sibling\_queue)

{

TreeNode \* result = nullptr;

if (sibling\_queue.empty()) return result;

Node \* node = sibling\_queue.front();

sibling\_queue.pop();

result = new TreeNode(node->val);

queue<Node \*> children\_queue;

for (size\_t i = 0; i < node->children.size(); i++)

{

children\_queue.push(node->children[i]);

}

result->left = encode(children\_queue);

result->right = encode(sibling\_queue);

return result;

}

// Decodes your binary tree to an n-ary tree.

void decode(TreeNode \* tree\_node, vector<Node \*>& children\_queue)

{

if (tree\_node == nullptr) return;

Node \* node = new Node();

node->val = tree\_node->val;

decode(tree\_node->left, node->children);

children\_queue.push\_back(node);

decode(tree\_node->right, children\_queue);

}

public:

// Encodes an n-ary tree to a binary tree.

TreeNode\* encode(Node\* root)

{

TreeNode \* result = nullptr;

if (root == nullptr) return result;

queue<Node\*> sibling\_queue;

sibling\_queue.push(root);

result = encode(sibling\_queue);

return result;

}

// Decodes your binary tree to an n-ary tree.

Node\* decode(TreeNode\* root)

{

Node \* result = nullptr;

if (root == nullptr) return result;

vector<Node \*>children\_queue;

decode(root, children\_queue);

result = children\_queue[0];

return result;

}

};