# Binary Tree Traverse

The binary tree problem is quite standard regarding the solution. The most important thing is that you only need to know how to traverse the tree. Such traversal can be implemented by a recursive function. Please Keep in mind when you traverse the tree, pass down and bring back from the subtree.

There are some special problems in this category, for example how to find the next node in a binary tree, how to deal with a Binary Sort team and convert a N way tree to binary tree.

## 145. Binary Tree Postorder Traversal

Hard

Given a binary tree, return the *postorder* traversal of its nodes' values.

**Example:**

**Input:** [1,null,2,3]

1

\

2

/

3

**Output:** [3,2,1]

**Follow up:** Recursive solution is trivial, could you do it iteratively?

### Analysis:

The post order traversal itself is not hard at all if you can use recursive function. The trick is how to do it in a non-recursive way. You can still use stack form this purpose. You need to push the node into stack twice, first iteration is to traverse subtree, the second iteration is to visit itself. Remember you need to traverse right subtree before the left one due to LIFO for stack.

/// <summary>

/// Leet code #145. Binary Tree Postorder Traversal

///

/// Given a binary tree, return the postorder traversal of its nodes' values.

/// For example:

/// Given binary tree {1,#,2,3},

/// 1

/// \

/// 2

/// /

/// 3

/// return [3,2,1].

/// Note: Recursive solution is trivial, could you do it iteratively?

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

{

std::vector<int> result;

std::stack<pair<int, TreeNode \*>> node\_stack;

node\_stack.push(make\_pair(0, root));

// When we start the stack is empty

while (!node\_stack.empty())

{

pair<int, TreeNode\*> p = node\_stack.top();

node\_stack.pop();

if (p.second == nullptr) continue;

if (p.first == 0)

{

node\_stack.push(make\_pair(1, p.second));

node\_stack.push(make\_pair(0, p.second->right));

node\_stack.push(make\_pair(0, p.second->left));

}

else

{

result.push\_back(p.second->val);

}

}

return result;

}

## 105. Construct Binary Tree from Preorder and Inorder Traversal

Medium

Given preorder and inorder traversal of a tree, construct the binary tree.

**Note:**  
You may assume that duplicates do not exist in the tree.

For example, given

preorder = [3,9,20,15,7]

inorder = [9,3,15,20,7]

Return the following binary tree:

3

/ \

9 20

/ \

15 7

### Analysis:

The root is always the first node in pre-order sequence, and if we can find the node in in-order sequence, then the left part of the in-order in sequence is the left subtree and right part of the in-order sequence is the right subtree. You solve the problem recursively.

/// <summary>

/// Leet code #105. Build binary tree from preoder and inorder travesal

/// </summary>

TreeNode\* LeetCodeTree::buildTreeFromPreorderandInorder(

vector<int>& preorder, vector<int>& inorder,

size\_t start\_preorder, size\_t start\_inorder,

size\_t length)

{

if (length == 0)

{

return nullptr;

}

TreeNode \*root;

root = new TreeNode(preorder[start\_preorder]);

size\_t index;

for (index = 0; index < length; index++)

{

if (inorder[start\_inorder + index] == preorder[start\_preorder])

{

break;

}

}

TreeNode \*left = buildTreeFromPreorderandInorder(preorder, inorder, start\_preorder + 1,

start\_inorder, index);

TreeNode \*right = buildTreeFromPreorderandInorder(preorder, inorder,

start\_preorder + 1 + index, start\_inorder + index + 1, length - index - 1);

root->left = left;

root->right = right;

return root;

}

/// <summary>

/// Leet code #105. Construct Binary Tree from Preorder and Inorder Traversal

///

/// Medium

///

/// Given preorder and inorder traversal of a tree, construct the binary tree.

///

/// Note:

/// You may assume that duplicates do not exist in the tree.

///

/// For example, given

///

/// preorder = [3,9,20,15,7]

/// inorder = [9,3,15,20,7]

/// Return the following binary tree:

///

/// 3

/// / \

/// 9 20

/// / \

/// 15 7

/// </summary>

TreeNode\* LeetCodeTree::buildTreeFromPreorderandInorder(vector<int>& preorder, vector<int>& inorder)

{

if ((preorder.size() == 0) || (inorder.size() == 0))

{

return nullptr;

}

TreeNode \*root =

buildTreeFromPreorderandInorder(preorder, inorder, 0, 0, preorder.size());

return root;

}

## 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.

**Follow up:**

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

**Example 1:**



**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.

**Constraints:**

* The number of nodes in the given tree is less than 6000.
* -100 <= node.val <= 100

### Analysis:

To track the next right in current level, you need to track the next right in the parent level, and the children node from the next parent.

/// <summary>

/// Append the tree linked node to end

/// </summary>

void LeetCode::connectRight(TreeLinkNode \*&head, TreeLinkNode \*&ptr, TreeLinkNode \* node)

{

if (node == nullptr) return;

node->next = nullptr;

if (head == nullptr)

{

head = node;

}

else

{

ptr->next = node;

}

ptr = node;

}

/// <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 LeetCode::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;

}

}

## 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:**

****

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.

/// <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;

}

};

## 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

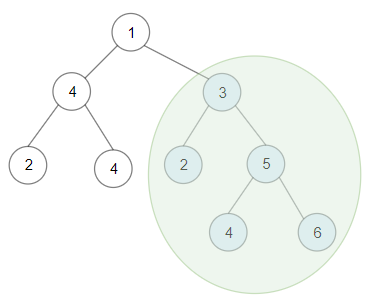
Hard

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:**

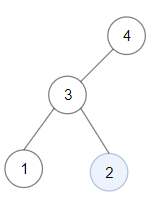


**Input:** root = [1,4,3,2,4,2,5,null,null,null,null,null,null,4,6]

**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:**

* 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;

}

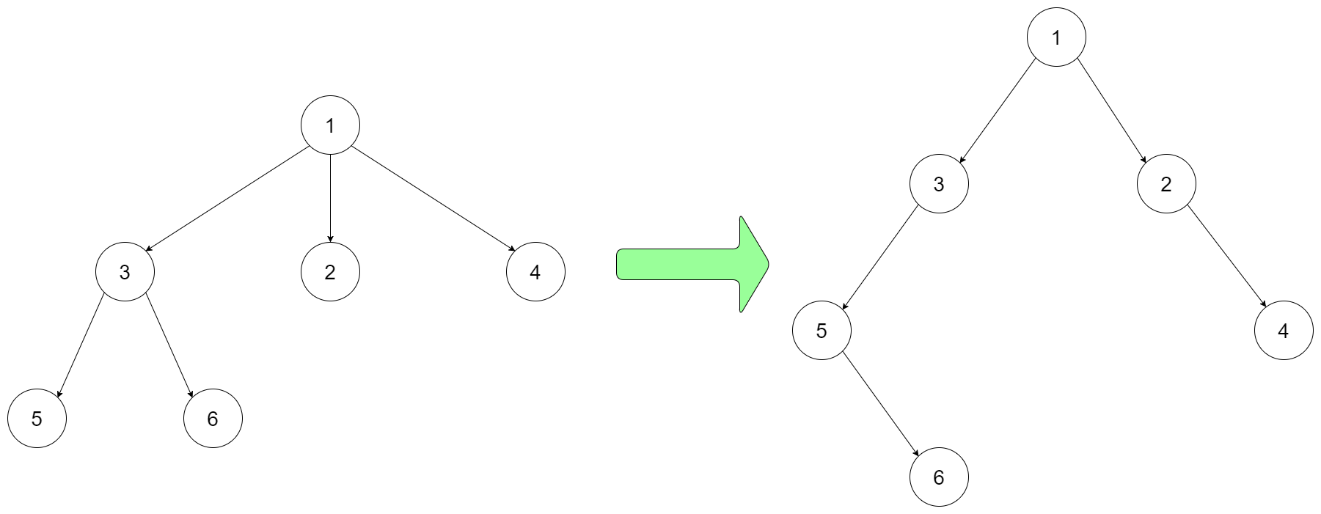
## 431. Encode N-ary Tree to Binary Tree

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:



**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;

}

};