# Binary Tree

Binary Tree is one of the common interview questions, maybe the easiest. To say it is easy because the way of coding in Binary Tree search is in some standard pattern.

First, we need to be familiar with binary tree traverse. In the text book, it is described as pre-order, inorder and postorder traversal, such tree traversal can normally resolve 80% of the problems. The most common pattern you will use is the post-order traversal, which is visit all the tree the the parent node itself. When we traverse back from the children nodes we should ask ourselves, **what kind of information we want to return from the children**, such information will determine the process logic on the parent node.

## Tree Traversal

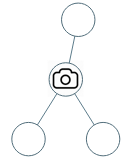
### 968. Binary Tree Cameras

Given a binary tree, we install cameras on the nodes of the tree.

Each camera at a node can monitor **its parent, itself, and its immediate children**.

Calculate the minimum number of cameras needed to monitor all nodes of the tree.

**Example 1:**

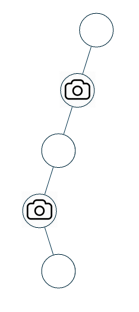


**Input:** [0,0,null,0,0]

**Output:** 1

**Explanation:** One camera is enough to monitor all nodes if placed as shown.

**Example 2:**



**Input:** [0,0,null,0,null,0,null,null,0]

**Output:** 2

**Explanation:** At least two cameras are needed to monitor all nodes of the tree. The above image shows one of the valid configurations of camera placement.

**Note:**

1. The number of nodes in the given tree will be in the range [1, 1000].
2. **Every** node has value 0.

**Analysis:**

1. **First the camera status is only determined in a path from root to current node, or say in reverse, the children to the current node. The camera status does not care in other branches in the tree. This is to say we can use traversal, maybe the postorder to resolve the problem.**
2. **Let’s look at each node and its children nodes. We cah assign 3 state to each node based where the camera is placed.**
   1. **There are no cameras on either left child or right child node, but there are cameras in direct children for both. This means the children are covered, and we do not need to put camera on the current node. We will give the current node the status 0. This status means the current node is not covered by camera.**
   2. **If any left child or right child is not covered by camera, we should anyway place a camera on the current node. Otherwise we will miss a node. We give this status as 2, means there is a camera on this node.**
   3. **In the other case, it means both left children and right children are covered by camera and there is at least one camera in either left child or right child. We can consider the current node is already covered in camera. We assign the value of 1 to the current node.**
3. **Based on the analysis 2, we can have the logic to calculate the value for a node.**
   1. **Give null node (empty children, not existing) a value of 1. Assume it is covered.**
   2. **If both children having the value of 1, we assign 0 to the node.**
   3. **If any child node having the value of 0, we assign 2 to the node and place a camera and we count each camera.**
   4. **Otherwise we assign 1 to the node and assume it is covered.**

/// <summary>

/// Leet code #968. Binary Tree Cameras

/// </summary>

int LeetCode::minCameraCover(TreeNode\* root, int &result)

{

// the child is null or covered

if (root == nullptr) return 1;

int left = minCameraCover(root->left, result);

int right = minCameraCover(root->right, result);

// if left and right are all covered, but no camera, return 0

// means myself not covered.

if (left == 1 && right == 1) return 0;

// either left or right not covered, we need a camera here

if (left == 0 || right == 0)

{

result++;

return 2;

}

else // either left or right contains a camera

{

return 1;

}

}

/// <summary>

/// Leet code #968. Binary Tree Cameras

///

/// Given a binary tree, we install cameras on the nodes of the tree.

///

/// Each camera at a node can monitor its parent, itself, and its immediate

/// children.

///

/// Calculate the minimum number of cameras needed to monitor all nodes of

/// the tree.

///

/// Example 1:

/// Input: [0,0,null,0,0]

/// Output: 1

/// Explanation: One camera is enough to monitor all nodes if placed as shown.

///

/// Example 2:

/// Input: [0,0,null,0,null,0,null,null,0]

/// Output: 2

/// Explanation: At least two cameras are needed to monitor all nodes of the

/// tree. The above image shows one of the valid configurations of camera

/// placement.

///

/// Note:

///

/// 1. The number of nodes in the given tree will be in the range [1, 1000].

/// 2. Every node has value 0.

/// </summary>

int LeetCode::minCameraCover(TreeNode\* root)

{

int result = 0;

if (minCameraCover(root, result) == 0)

{

result++;

}

return result;

}

### 236. Lowest Common Ancestor of a Binary Tree

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**).”

Given the following binary tree:  root = [3,5,1,6,2,0,8,null,null,7,4]



**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.

**Note:**

* All of the nodes' values will be unique.
* p and q are different and both values will exist in the binary tree.

/// <summary>

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

/// </summary>

int LeetCode::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

/// 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: “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).”

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

/// / \

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

/// / \ / \

/// 6 2 0 8

/// / \

/// 7 4

/// For example, the lowest common ancestor (LCA) of nodes 5 and 1 is 3.

/// Another example is LCA of nodes 5 and 4 is 5,

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

/// </summary>

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

{

TreeNode \* result = nullptr;

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

return result;

}

### 124. Binary Tree Maximum Path Sum

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

/// <summary>

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

/// </summary>

void LeetCode::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 LeetCode::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;

}

## Level Traversal and Tree Serialization

### 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.

**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.

Please notice that when you encounter this problem in a real interview, you do not need to follow the exact same format as LeetCode. you can have tailing null in the end.

The serialization process is a simple level traversal. The deserialization process is a little bit tricky. You need to split the string and process them one by one in the queue, and put the node in the output queue, for every node coming from output queue, you need to construct two children for node, and these children are from the input queue.

/// <summary>

/// LeetCode 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 LeetCode::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\* LeetCode::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();

input\_queue.pop();

}

if (!input\_queue.empty())

{

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

input\_queue.pop();

}

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

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

}

}

}

return root;

}

**Please notice that the tree serialization can be used many scenarios, you do not need to make them as a string, maybe just an array. For example, it can be used to compare one tree is a subtree of another and to determine if a tree is a complete tree or not. When doing so you may need to output all the empty leaves.**

## Predecessor and Successor

To find the predecessor, the common steps are below.

1. Search from top to bottom, push the path to the stack,
2. If the target node is found, pop it out of stack
3. check if the target has left child, if the answer is yes, **greedily** look for its (the left child) right child and push the path into stack, until there is no right child, the stack top is the predecessor.
4. If the target does not contain a left child, pop up the stack to search the ancestor until we find a one the target node is in its right branch.
5. If we want to find the next predecessor, we simply pop up a node from stack, repeat step #3.

The similar way can be used to find successor, just switch the left child and right child in the steps above.

### 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.

**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.

/// <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;

}

};

**The following scenario is more complicated.**

### 272. Closest Binary Search Tree Value II

Given a non-empty binary search tree and a target value, find *k* values in the BST that are closest to the target.

**Note:**

* Given target value is a floating point.
* You may assume *k* is always valid, that is: *k* ≤ total nodes.
* You are guaranteed to have only one unique set of *k* values in the BST that are closest to the target.

**Example:**

**Input:** root = [4,2,5,1,3], target = 3.714286, and *k* = 2

4

/ \

2 5

/ \

1 3

**Output:** [4,3]

**Follow up:**  
Assume that the BST is balanced, could you solve it in less than *O*(*n*) runtime (where *n* = total nodes)?

/// <summary>

/// Leet code #272. Closest Binary Search Tree Value II

/// </summary>

vector<int> LeetCode::getPredecessor(stack<TreeNode\*> left\_stack, int k)

{

vector<int> result;

while (!left\_stack.empty())

{

TreeNode \* node = left\_stack.top();

left\_stack.pop();

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

if (result.size() == k) break;

node = node->left;

while (node != nullptr)

{

left\_stack.push(node);

node = node->right;

}

}

return result;

}

/// <summary>

/// Leet code #272. Closest Binary Search Tree Value II

/// </summary>

vector<int> LeetCode::getSuccessor(stack<TreeNode\*> right\_stack, int k)

{

vector<int> result;

while (!right\_stack.empty())

{

TreeNode \* node = right\_stack.top();

right\_stack.pop();

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

if (result.size() == k) break;

node = node->right;

while (node != nullptr)

{

right\_stack.push(node);

node = node->left;

}

}

return result;

}

/// <summary>

/// Leet code #272. Closest Binary Search Tree Value II

///

/// Given a non-empty binary search tree and a target value, find k values in the BST that are closest to the target.

/// Note:

/// Given target value is a floating point.

/// You may assume k is always valid, that is: k ≤ total nodes.

/// You are guaranteed to have only one unique set of k values in the BST that are closest to the target.

/// Follow up:

/// Assume that the BST is balanced, could you solve it in less than O(n) runtime (where n = total nodes)?

/// Hint:

/// 1.Consider implement these two helper functions:

/// i.getPredecessor(N), which returns the next smaller node to N.

/// ii.getSuccessor(N), which returns the next larger node to N.

/// 2.Try to assume that each node has a parent pointer, it makes the problem much easier.

/// 3.Without parent pointer we just need to keep track of the path from the root to the current node using a stack.

/// 4.You would need two stacks to track the path in finding predecessor and successor node separately.

/// </summary>

vector<int> LeetCode::closestKValues(TreeNode\* root, double target, int k)

{

vector<int> result;

stack<TreeNode\*> left\_stack;

stack<TreeNode\*> right\_stack;

TreeNode \*node = root;

while (node != nullptr)

{

if (target <= node->val)

{

right\_stack.push(node);

node = node->left;

}

else if (target > node->val)

{

left\_stack.push(node);

node = node->right;

}

}

vector<int> left\_values = getPredecessor(left\_stack, k);

vector<int> right\_values = getSuccessor(right\_stack, k);

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

while (left\_index < (int)left\_values.size() || right\_index < (int)right\_values.size())

{

if (result.size() == k) break;

if (left\_index == left\_values.size())

{

result.push\_back(right\_values[right\_index]);

right\_index++;

}

else if (right\_index == right\_values.size())

{

result.push\_back(left\_values[left\_index]);

left\_index++;

}

else if (abs(left\_values[left\_index] - target) < abs(right\_values[right\_index] - target))

{

result.push\_back(left\_values[left\_index]);

left\_index++;

}

else

{

result.push\_back(right\_values[right\_index]);

right\_index++;

}

}

return result;

}

## Complex Example

### 99. Recover Binary Search Tree

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?

**#99 Recover binary tree**

<https://leetcode.com/problems/recover-binary-search-tree>

/// <summary>

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

/// </summary>

void LeetCode::recoverTree(TreeNode\* root, TreeNode\* &min\_node, TreeNode\* &max\_node, TreeNode\* &first, TreeNode\* &second)

{

if (root == nullptr) return;

TreeNode\* left\_min = root;

TreeNode\* left\_max = root;

TreeNode\* right\_min = root;

TreeNode\* right\_max = root;

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

{

min\_node = root;

max\_node = root;

return;

}

if (root->left != nullptr)

{

recoverTree(root->left, left\_min, left\_max, first, second);

}

if (root->right != nullptr)

{

recoverTree(root->right, right\_min, right\_max, first, second);

}

if (left\_max->val > root->val)

{

first = left\_max;

second = root;

}

if (root->val > right\_min->val)

{

if (left\_max->val <= root->val) first = root;

second = right\_min;

}

min\_node = left\_min;

if (root->val < min\_node->val) min\_node = root;

if (right\_min->val < min\_node->val) min\_node = right\_min;

max\_node = left\_max;

if (root->val > max\_node->val) max\_node = root;

if (right\_max->val > max\_node->val) max\_node = right\_max;

}

/// <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 LeetCode::recoverTree(TreeNode\* root)

{

TreeNode \*min\_node = nullptr, \*max\_node = nullptr;

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

recoverTree(root, min\_node, max\_node, first, second);

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

{

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

}

}

/// <summary>

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

/// </summary>

void LeetCode::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 LeetCode::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);

}

}