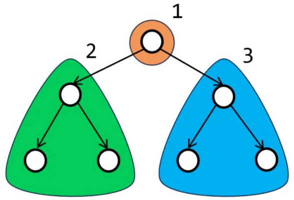
TREE

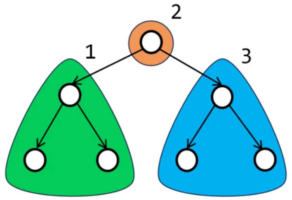
Tree Traversals

Depth-First Traversals

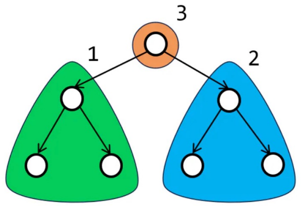
- **Pre-order**: Root – Left – Right



- **In-order**: Left – Root – Right



- **Post-order**: Left – Right – Root



Breadth-First Traversal (Level Order Traversal)

Visit every node on a level before moving to a lower level.

Image credits:
https://deriveit.org/coding/tree-dfs-113

Tree Traversals

Depth-First Traversals

Use a recursive algorithm to traverse according to the order

- **Pre-order**: Root – Left – Right

```
if (!root) return;
doSomething();
visit(node->left);
visit(node->right);
```

- **In-order**: Left – Root – Right

```
if (!root) return;
visit(node->left);
doSomething();
visit(node->right);
```

- **Post-order**: Left – Right – Root

```
if (!root) return;
visit(node->left);
visit(node->right);
doSomething();
```

Tree Traversals

Example of pre-order and in-order

```
struct TreeNode {
    int val;
    TreeNode *left, *right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

// Pre-order traversal
void preorderTraversal(TreeNode* root) {
    if (root == nullptr) return;
    cout << root->val << " ";
    preorderTraversal(root->left);
    preorderTraversal(root->right);
}

// In-order traversal
void inorderTraversal(TreeNode* root) {
    if (root == nullptr) return;
    inorderTraversal(root->left);
    cout << root->val << " ";
    inorderTraversal(root->right);
}
```

Tree Traversals

Example of post-order and level-order

```cpp
struct TreeNode {
    int val;
    TreeNode *left, *right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

// Post-order traversal
void postorderTraversal(TreeNode* root) {
    if (root == nullptr) return;
    postorderTraversal(root->left);
    postorderTraversal(root->right);
    cout << root->val << " ";
}

// Level-order traversal using a queue
void levelOrderTraversal(TreeNode* root) {
    if (root == nullptr) return;
    queue<TreeNode*> q;
    q.push(root);
    while (!q.empty()) {
        TreeNode* current = q.front();
        q.pop();
        cout << current->val << " ";
        if (current->left != nullptr) q.push(current->left);
        if (current->right != nullptr) q.push(current->right);
    }
}
```

Problem – Maximum Depth of Binary Tree

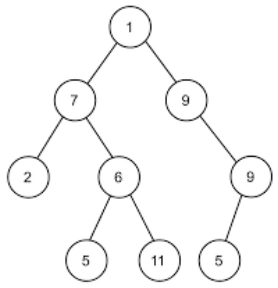
Easy

LeetCode https://leetcode.com/problems/maximum-depth-of-binary-tree

Problem Statement

- Given the root of a binary tree, find the *maximum depth*
- **Example:**

  root = [1,7,9,2,6,null,9,null,null,5,11,5,null]
- **Output**: 4



Solution – Maximum Depth of Binary Tree

Easy

LeetCode https://leetcode.com/problems/maximum-depth-of-binary-tree

Solution

- Perform **post-order** traversal: left - right - root
- Recursively go left and right to find each value
- Return the max of each one

Code – Maximum Depth of Binary Tree

Easy

LeetCode https://leetcode.com/problems/maximum-depth-of-binary-tree

```
int maxDepth(TreeNode* root) {
    if (!root) return 0;
    // find max left
    int maxLeft = maxDepth(root->left);
    // find max right
    int maxRight = maxDepth(root->right);
    // return max +1 (account for root)
    return std::max(maxLeft, maxRight) + 1;
}
```

Problem – Path Sum

Easy

LeetCode https://leetcode.com/problems/path-sum

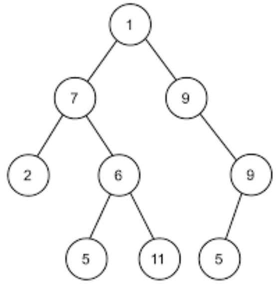
Problem Statement

- It is given the **root** of a binary tree and an integer **target sum**
- **Example:**

  root = [1,7,9,2,6,null,9,null,null,5,11,5,null]

  target sum = 10
- Return true if there is a path from root to leaf that adds up to 10
- **Output**: true

  Node 1 + Node 7 + Node 2 = 10



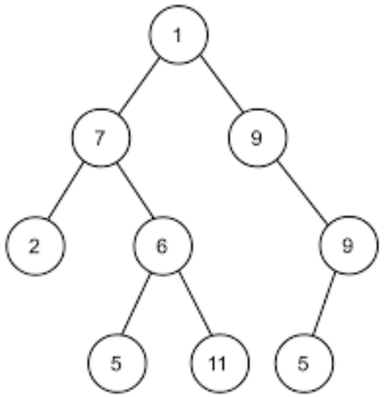
Solution – Path Sum

Easy

LeetCode https://leetcode.com/problems/path-sum

Solution

- Start from root node (1)
- Subtract from target number (example 10 – 1 = 9)
- Continue going down the tree, until the target is 0, return true
- After visiting all nodes, if the target is not zero, return false



Code – Path Sum

Easy

LeetCode https://leetcode.com/problems/path-sum

```
bool hasPathSum(TreeNode* root, int targetSum) {
    if (!root) {
        return false;
    }
    // we want targetSum to be zero
    targetSum -= root->val;
    // if there is no left, no right, we've reached the end of the path
    // so if the targetSum is zero, then the nodes summed up to the targetSum
    if (!root->left && !root->right && targetSum == 0) {
        return true;
    }
    // propagate to left and right
    return hasPathSum(root->left, targetSum) || hasPathSum(root->right, targetSum);
}
```

Also, a small performance tweak can be made by avoiding writing *targetSum:* `targetSum -= root->val`
This will avoid a memory write access, making the calculation directly in the CPU, but also at a cost of readability

```
    if (!root->left && !root->right && targetSum – root->val == 0) {
     …
    return hasPathSum(root->left, targetSum – root->val) || hasPathSum(root->right, targetSum – root->val);
```

Problem – Kth Smallest Element in a BST

Medium

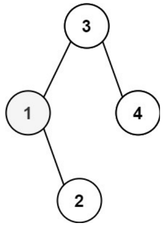
https://leetcode.com/problems/kth-smallest-element-in-a-bst

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

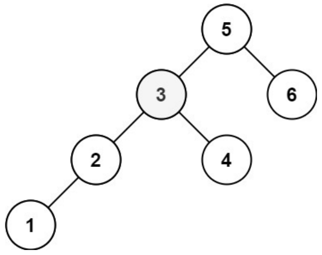
Output: 1



Example 2

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

Output: 3



Solution – Kth Smallest Element in a BST

Medium

https://leetcode.com/problems/maximum-depth-of-binary-tree

```
int kthSmallest(TreeNode* root, int k) {
    int count = 0;
    int output;
    traverse(root, count, output, k);
    return output;
}

// perform in-order traversal:  left, node, right
void traverse(TreeNode* node, int& count, int &output, int k) {
    if (!node) return;
    traverse(node->left, count, output, k);
    count++;
    if (count == k) {
        output = node->val;
        return;
    }
    traverse(node->right, count, output, k);
}
```