Binary Tree Summary

Summary Table

Topic	Approach	Type	Code Available
TreeNode Struct	Manual	C++	
Traversals	Recursive & Iterative	C++	
Max Height	Recursion	C++	
Max Path Sum	Recursion + DP	C++	
Compare Trees	DFS	C++	
Vertical Traversal	BFS + Map	C++	
Left View	BFS with Level Info	C++	
Right View	BFS with Level Info	C++	
Top View	BFS + HD Map	C++	
Bottom View	BFS + HD Map	C++	
Zigzag Traversal	BFS + Direction Flag	C++/Java	
Boundary Traversal	DFS + Level Order	C++	

TreeNode Structure (C++)

```
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode() : val(0), left(nullptr), right(nullptr) {}
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left)
};
```

Tree Traversals

Inorder Traversal (C++)

```
void inorder(TreeNode* root, vector<int>& result) {
    if (root == nullptr) return;
    inorder(root->left, result);
    result.push_back(root->val);
    inorder(root->right, result);
}
```

Preorder Traversal (C++)

```
void preorder(TreeNode* root, vector<int>& result) {
    if (root == nullptr) return;
    result.push_back(root->val);
    preorder(root->left, result);
    preorder(root->right, result);
}
```

Postorder Traversal (C++)

```
void postorder(TreeNode* root, vector<int>& result) {
    if (root == nullptr) return;
    postorder(root->left, result);
    postorder(root->right, result);
    result.push_back(root->val);
}
```

Level Order Traversal (Java)

```
public List<List<Integer>> levelOrder(TreeNode root) {
    Queue<TreeNode> q = new LinkedList<>();
    List<List<Integer>> result = new ArrayList<>();
    if (root == null) return result;

    q.offer(root);
```

```
    while (!q.isEmpty()) {
        int levelSize = q.size();
        List<Integer> currentLevel = new ArrayList<>();
        for (int i = 0; i < levelSize; i++) {
            TreeNode curr = q.poll();
            currentLevel.add(curr.val);
            if (curr.left != null) q.offer(curr.left);
            if (curr.right != null) q.offer(curr.right);
        }
        result.add(currentLevel);
    }
    return result;
}
```

Maximum Height of Binary Tree (C++)

```
int maxDepth(TreeNode* root) {
    if (root == nullptr) return 0;
    return 1 + max(maxDepth(root->left), maxDepth(root->right));
}
```

Maximum Path Sum (C++)

```
int find(TreeNode* node, int& maxi) {
    if (!node) return 0;
    int left = max(0, find(node->left, maxi));
    int right = max(0, find(node->right, maxi));
    maxi = max(maxi, node->val + left + right);
    return node->val + max(left, right);
}

int maxPathSum(TreeNode* root) {
    int result = INT_MIN;
    find(root, result);
    return result;
}
```

Compare Trees (C++)

```
bool isSameTree(TreeNode* p, TreeNode* q) {
    if (!p && !q) return true;
    if (!p || !q || p->val != q->val) return false;
    return isSameTree(p->left, q->left) && isSameTree(p->right, q->right)
}
```

Vertical Order Traversal (C++)

```
vector<vector<int>> verticalOrder(TreeNode* root) {
    map<int, vector<int>> m;
    queue<pair<TreeNode*, int>> q;
    if (root) q.push({root, 0});

    while (!q.empty()) {
        auto [node, hd] = q.front();
        q.pop();
        m[hd].push_back(node->val);
        if (node->left) q.push({node->left, hd - 1});
        if (node->right) q.push({node->right, hd + 1});
    }

    vector<vector<int>> res;
    for (auto& [_, v] : m) res.push_back(v);
    return res;
}
```

Views of Binary Tree

Left View (C++)

```
vector<int> leftView(TreeNode* root) {
    vector<int> result;
    if (!root) return result;
    queue<TreeNode*> q;
    q.push(root);
```

```
    while (!q.empty()) {
        int levelSize = q.size();
        for (int i = 0; i < levelSize; i++) {
            TreeNode* node = q.front(); q.pop();
            if (i == 0) result.push_back(node->val);
            if (node->left) q.push(node->left);
            if (node->right) q.push(node->right);
        }
    }
    return result;
}
```

Right View (C++)

```
vector<int> rightView(TreeNode* root) {
    vector<int> result;
    if (!root) return result;
    queue<TreeNode*> q;
    q.push(root);
    while (!q.empty()) {
        int levelSize = q.size();
        for (int i = 0; i < levelSize; i++) {
            TreeNode* node = q.front(); q.pop();
            if (i == levelSize - 1) result.push_back(node->val);
            if (node->left) q.push(node->left);
            if (node->right) q.push(node->right);
        }
    }
    return result;
}
```

Top View (C++)

```
vector<int> topView(TreeNode* root) {
    map<int, int> m;
    queue<pair<TreeNode*, int>> q;
    if (root) q.push({root, 0});
    while (!q.empty()) {
        auto [node, hd] = q.front(); q.pop();
        if (m.find(hd) == m.end()) m[hd] = node->val;
        if (node->left) q.push({node->left, hd - 1});
```

```
        if (node->right) q.push({node->right, hd + 1});
    }
    vector<int> result;
    for (auto& [_, val] : m) result.push_back(val);
    return result;
}
```

Bottom View (C++)

```
vector<int> bottomView(TreeNode* root) {
    map<int, int> m;
    queue<pair<TreeNode*, int>> q;
    if (root) q.push({root, 0});
    while (!q.empty()) {
        auto [node, hd] = q.front(); q.pop();
        m[hd] = node->val;
        if (node->left) q.push({node->left, hd - 1});
        if (node->right) q.push({node->right, hd + 1});
    }
    vector<int> result;
    for (auto& [_, val] : m) result.push_back(val);
    return result;
}
```

Zigzag Level Order Traversal

C++

```
vector<vector<int>> zigzagLevelOrder(TreeNode* root) {
    vector<vector<int>> result;
    if (!root) return result;
    queue<TreeNode*> q;
    q.push(root);
    bool leftToRight = true;
    while (!q.empty()) {
        int size = q.size();
        vector<int> level(size);
        for (int i = 0; i < size; i++) {
            TreeNode* node = q.front(); q.pop();
```

```
            int index = leftToRight ? i : size - i - 1;
            level[index] = node->val;
            if (node->left) q.push(node->left);
            if (node->right) q.push(node->right);
        }
        result.push_back(level);
        leftToRight = !leftToRight;
    }
    return result;
}
```

Java

```
public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
    List<List<Integer>> result = new ArrayList<>();
    if (root == null) return result;

    Queue<TreeNode> q = new LinkedList<>();
    q.offer(root);
    boolean leftToRight = true;

    while (!q.isEmpty()) {
        int size = q.size();
        Integer[] level = new Integer[size];

        for (int i = 0; i < size; i++) {
            TreeNode node = q.poll();
            int index = leftToRight ? i : size - 1 - i;
            level[index] = node.val;

            if (node.left != null) q.offer(node.left);
            if (node.right != null) q.offer(node.right);
        }

        result.add(Arrays.asList(level));
        leftToRight = !leftToRight;
    }

    return result;
}
```

Boundary Traversal (C++)

```
void addLeftBoundary(TreeNode* root, vector<int>& res) {
    TreeNode* cur = root->left;
    while (cur) {
        if (cur->left || cur->right) res.push_back(cur->val);
        if (cur->left) cur = cur->left;
        else cur = cur->right;
    }
}

void addRightBoundary(TreeNode* root, vector<int>& res) {
    TreeNode* cur = root->right;
    stack<int> tmp;
    while (cur) {
        if (cur->left || cur->right) tmp.push(cur->val);
        if (cur->right) cur = cur->right;
        else cur = cur->left;
    }
    while (!tmp.empty()) {
        res.push_back(tmp.top()); tmp.pop();
    }
}

void addLeaves(TreeNode* root, vector<int>& res) {
    if (!root) return;
    if (!root->left && !root->right) {
        res.push_back(root->val);
        return;
    }
    addLeaves(root->left, res);
    addLeaves(root->right, res);
}

vector<int> boundaryTraversal(TreeNode* root) {
    vector<int> res;
    if (!root) return res;
    if (root->left || root->right) res.push_back(root->val);
    addLeftBoundary(root, res);
    addLeaves(root, res);
    addRightBoundary(root, res);
    return res;
}
```