Binary Tree Learning 2

This document summarizes all the binary tree topics learned today with structured explanations, step-wise approaches, and C++ code samples.

---

1. Check Symmetric Tree (Mirror Tree)

Problem:

Check whether a binary tree is symmetric (mirror image about the center).

Approach:

- Use recursion to check if left and right subtrees are mirrors of each other.

Code:

```
bool isMirror(TreeNode* t1, TreeNode* t2) {
    if (!t1 && !t2) return true;
    if (!t1 || !t2 || t1->val != t2->val) return false;
    return isMirror(t1->left, t2->right) && isMirror(t1->right, t2->left)
}

bool isSymmetric(TreeNode* root) {
    if (!root) return true;
    return isMirror(root->left, root->right);
}
```

Time Complexity: O(N)
Space Complexity: O(H) — recursion stack

---

2. Print Root-to-Node Path

Problem:

Given a binary tree and a target node, print the path from root to that node.

Approach:

- Use backtracking DFS.
- Push node value to path, recurse, and pop if not found.

Code:

```
bool getPath(TreeNode* root, int target, vector<int>& path) {
    if (!root) return false;
    path.push_back(root->val);
    if (root->val == target) return true;

    if (getPath(root->left, target, path) || getPath(root->right, target,

    path.pop_back();
    return false;
}

void printPath(TreeNode* root, int target) {
    vector<int> path;
    if (getPath(root, target, path)) {
        for (int val : path) cout << val << " ";
    } else {
        cout << "Target not found\n";
    }
}
```

Time Complexity: O(N)

Space Complexity: O(H)

---

3. Binary Tree Paths (Root to Leaf)

Problem:

Print all paths from root to all leaf nodes as strings.

Approach:

- Use recursive DFS.
- Store current path in a vector, push when leaf is reached.

Code:

```
void dfs(TreeNode* node, vector<int> path, vector<string>& res) {
    if (!node) return;
    path.push_back(node->val);
    if (!node->left && !node->right) {
        string s = to_string(path[0]);
        for (int i = 1; i < path.size(); i++) s += "->" + to_string(path[
        res.push_back(s);
        return;
    }
    dfs(node->left, path, res);
    dfs(node->right, path, res);
}

vector<string> binaryTreePaths(TreeNode* root) {
    vector<string> result;
    dfs(root, {}, result);
    return result;
}
```

Time Complexity: O(N)
Space Complexity: O(H)

---

4. Lowest Common Ancestor (LCA)

Problem:

Find the lowest common ancestor of two nodes in a binary tree.

Approach:

- Use post-order recursion.
- Return the node if found either p or q, else combine results.

Code:

```
TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q)
    if (!root || root == p || root == q) return root;

    TreeNode* left = lowestCommonAncestor(root->left, p, q);
    TreeNode* right = lowestCommonAncestor(root->right, p, q);

    if (left && right) return root;
    return left ? left : right;
}
```

Time Complexity: O(N)
Space Complexity: O(H)

---

5. Maximum Width of Binary Tree

Problem:

Find the maximum width of the binary tree — the width between leftmost and rightmost non-null nodes at any level.

Approach:

- Use level-order traversal with indices like a complete binary tree.
- Normalize indices per level to avoid overflow.

Code:

```
int widthOfBinaryTree(TreeNode* root) {
    if (!root) return 0;
    queue<pair<TreeNode*, unsigned long long>> q;
    q.push({root, 0});
    int maxWidth = 0;

    while (!q.empty()) {
        int size = q.size();
        unsigned long long minIndex = q.front().second;
        unsigned long long first = 0, last = 0;

        for (int i = 0; i < size; i++) {
```

```
            auto [node, idx] = q.front(); q.pop();
            idx -= minIndex; // normalize
            if (i == 0) first = idx;
            if (i == size - 1) last = idx;
            if (node->left) q.push({node->left, 2 * idx + 1});
            if (node->right) q.push({node->right, 2 * idx + 2});
        }

        maxWidth = max(maxWidth, int(last - first + 1));
    }

    return maxWidth;
}
```

Time Complexity: O(N)
Space Complexity: O(N)

---

6. Check Children Sum Property

Problem:

Check if each node equals the sum of its children (NULL children count as 0).

Code:

```
bool checkChildrenSum(TreeNode* root) {
    if (!root || (!root->left && !root->right)) return true;
    int leftVal = root->left ? root->left->val : 0;
    int rightVal = root->right ? root->right->val : 0;
    return (root->val == leftVal + rightVal) &&
           checkChildrenSum(root->left) &&
           checkChildrenSum(root->right);
}
```

Time Complexity: O(N)
Space Complexity: O(H)

---

7. Modify Tree to Satisfy Children Sum Property

Problem:

Modify tree in-place so every node becomes equal to sum of its children by only increasing node values.

Approach:

- Use post-order traversal.
- Push parent value down if needed.
- Update back up from fixed children.

Code:

```
void convertToChildrenSum(TreeNode* root) {
    if (!root || (!root->left && !root->right)) return;

    convertToChildrenSum(root->left);
    convertToChildrenSum(root->right);

    int leftVal = root->left ? root->left->val : 0;
    int rightVal = root->right ? root->right->val : 0;
    int childSum = leftVal + rightVal;

    if (childSum >= root->val) {
        root->val = childSum;
    } else {
        if (root->left) root->left->val = root->val;
        if (root->right) root->right->val = root->val;
    }

    convertToChildrenSum(root->left);
    convertToChildrenSum(root->right);

    root->val = (root->left ? root->left->val : 0) +
                (root->right ? root->right->val : 0);
}
```

Time Complexity: O(N)
Space Complexity: O(H)

Summary Table

Problem	Approach	TC	SC
Symmetric Tree	DFS	O(N)	O(H)
Root to Node Path	Backtracking DFS	O(N)	O(H)
All Root to Leaf Paths	DFS	O(N)	O(H)
LCA in Binary Tree	Post-order	O(N)	O(H)
Width of Binary Tree	Level-order BFS	O(N)	O(N)
Check Children Sum Property	Post-order DFS	O(N)	O(H)
Modify Tree to Children Sum Prop	Post-order Fix	O(N)	O(H)