Binary Tree Notes:

1️⃣ Level Order Traversal (BFS)

Type: Basic BFS / Level Order

Problem Solved: “Print all levels of a tree”

Approach / Logic:

1. Use queue → push root
2. Loop while queue not empty:
    - Record `size = queue.size()`
    - Loop over level size: pop node, push left & right children
3. For storing per level → vector inside vector

Tricks / Observations:

- Level-based problems → BFS
- i==0 → first node of level → Left View
- i==size-1 → last node → Right View

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

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

```
        }
        ans.push_back(level);
    }
    return ans;
}
```

2️⃣ Zigzag Level Order

Type: BFS + Alternating Direction

Problem Solved: LeetCode Zigzag Level Order

Approach / Logic:

- BFS like normal
- Maintain level count → if level%2==0 → push_back, else → insert at begin
- Collect vector of vectors

Tricks / Observations:

- level%2 → alternate order
- Queue still normal BFS

```
vector<vector<int>> zigzagLevelOrder(TreeNode* root){
    vector<vector<int>> ans;
    if(!root) return ans;
    queue<TreeNode*> q; q.push(root);
    int level=0;

    while(!q.empty()){
        int size = q.size();
        vector<int> levelAns;
        for(int i=0;i<size;i++){
            TreeNode* node = q.front(); q.pop();
            if(level%2==0) levelAns.push_back(node->val);
            else levelAns.insert(levelAns.begin(), node->val);
            if(node->left) q.push(node->left);
```

```
            if(node->right) q.push(node->right);
        }
        ans.push_back(levelAns);
        level++;
    }
    return ans;
}
```

3️⃣ Height of Binary Tree

Type: DFS / Recursion

Problem Solved: Compute height

Approach / Logic:

- Base case: root==NULL → return 0
- Else: 1 + max(height(left), height(right))

Trick:

- Simple post-order DFS
- Height = number of edges (or nodes based on definition)

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

4️⃣ Diameter of Binary Tree

Type: DFS + Post-order

Problem Solved: Max distance between 2 nodes

Approach / Logic:

1. Use height function recursively

2. Diameter = max(diameter of left subtree, diameter of right subtree, leftHeight+rightHeight)

Trick / Shortcut:

- Post-order recursion → height calculation
- Combine left + right heights at each node → candidate diameter
- Use global variable to track max (more optimal)

```
int diameter(TreeNode* root, int &maxDia){
    if(!root) return 0;
    int lh = diameter(root→left, maxDia);
    int rh = diameter(root→right, maxDia);
    maxDia = max(maxDia, lh+rh);
    return 1 + max(lh,rh);
}

int diameterOfBinaryTree(TreeNode* root){
    int maxDia=0;
    diameter(root,maxDia);
    return maxDia;
}
```

5️⃣ Maximum Path Sum

Type: DFS + Global variable

Problem Solved: Max sum of any path in BT

Approach / Logic:

- Recursive DFS → bottom-up
- For each node:
    - max path including left + node
    - max path including right + node
    - max path through node = left+node+right

- Update global max

Tricks / Shortcut:

- Return max of (node + left) or (node + right) for recursion
- Keep global max for total

```
int maxPath(TreeNode* root, int &res){
    if(!root) return 0;
    int left = max(0, maxPath(root→left,res));
    int right = max(0, maxPath(root→right,res));
    res = max(res, left + right + root→val);
    return root→val + max(left,right);
}

int maxPathSum(TreeNode* root){
    int res = INT_MIN;
    maxPath(root,res);
    return res;
}
```

6️⃣ Left / Right View

Type: BFS + Level order

Right View Logic: i==size-1

Left View Logic: i==0

Code Snippet (Right View):

```
if(i==size-1) ans.push_back(node→val);
```

Code Snippet (Left View):

```
if(i==0) ans.push_back(node→val);
```

Memory Trick:

- BFS skeleton same → only index condition changes

7️⃣ LCA (Lowest Common Ancestor)

Type: DFS / Bottom-up recursion

Approach / Logic:

- If root==p or q → return root
- left = dfs(left), right = dfs(right)
- Both left & right non-null → root is LCA
- Else return non-null

```
TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode
* q){
    if(!root) return NULL;
    if(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;
}
```

8️⃣ Top / Bottom View (High-level)

Type: BFS + Horizontal Distance

Logic / Trick:

- Top View → first node per HD
- Bottom View → last node per HD
- Map<int, val> → store nodes per HD

Skeleton Code:

```
queue<pair<TreeNode*, int>> q; // node, HD
map<int,int> mp;
```