Binary Trees

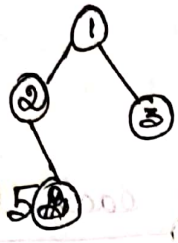
#1) Binary Tree Paths

Given the root; return all root to leaf paths.

— x —

O/P:- ["1→2→5"; "1→3"].

Input :-



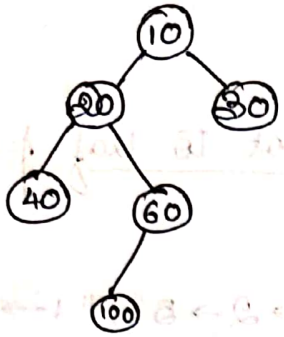
Code

```
TreeNode*
vector<string> binaryTreePaths(TreeNode* root){
    vector<string> paths;
    if(root == NULL) return path;
    solve(root, "", paths);
    return paths;
}

void solve(TreeNode* root, string currPath,
                    vector<string>& paths)
    if(root == NULL)
        return;

    currPath += to_string(root->val);
    if(root->left == NULL && root->right == NULL){ // leaf leaves
        paths.push_back(currPath);
        return;
    }
    if(root->left)
        solve(root->left, currPath+"→", paths);
    if(root->right)
        solve(root->right, currPath + "→", paths)
}
```

#2) Left View of a Binary Tree

O/P:- 10 20 40 100

Input



Algorithm

① Print ~~each~~ first node of each level while pushing left first followed by right.

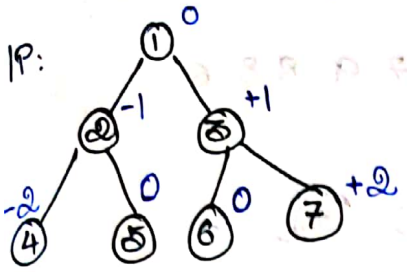
Code

```
Vector<int> leftView(Node *root){
    vector<int> res;
    if(root==NULL)
        return res;
    queue<Node*> q;
    q.push(root);
    while(!q.empty()){
        int size = q.size();
        for(int i=0; i<size; i++){
            Node *temp = q.front();
            if(i==0) // Push 1st element of each level
                res.push_back(temp->data);
            if(temp->left)
                q.push(temp->left)
            if(temp->right)
                q.push(temp->right)
            q.pop();
        }
    }
    return res;
}
```

// For Right View. first push right than left
Print 1st element of each level.

3) Top View of a Tree

I/P:



O/P:- 4 2 1 3 7

If we move left (-1)
move right (+1).

Algorithm (Do level Order Traversal)

① Store the first element of vertical traversal val in hashmap. 2d.

② Once a node exists for particular value do not replace with next value.

Code

```
vector<int> topView(Node *root){
    vector<int> res;
    if(root == NULL) return res;
    map<int,int> hashmap;                    // Storing vals of 0,-1,+1,-2,+2..
    queue<pair<Node*, int>> q;
    q.push({root, 0});
    while(!q.empty()){
        int size = q.size();
        Node *temp = q.front().first;
        int height = q.front().second;
        // Only for first one
        if(hashmap.find(height) == hash.end())
            hashmap[height] = temp->data;
        if(temp->left)
            q.push({temp->root, height-1});
        if(temp->right)
            q.push({temp->right, height+1});
        q.pop();
    }
    for(auto it: hashmap) res.push_back(it.second);
    return res;
}
```

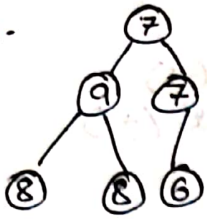
Initialisation

// Bottom View - Continuously update values in hashmap. for each height.

4)

Zigzag Traversal of a Binary Tree

I/P:-



O/P:- 7 9 7 8 8 6.

Algorithm

① Odd levels traversed left to right (Normally)
② Even levels right to left (Reversed).

Code

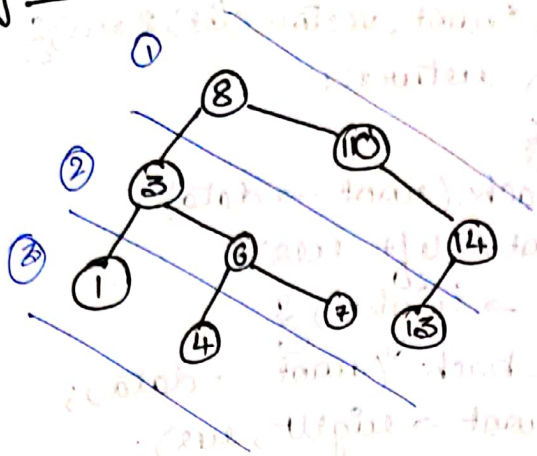
```
Vector<int> Zigzag (Node *root){
    vector<int> res;
    if (root == NULL)  return res;

    queue<Node*> q;
    q.push(root);
    int levelNow = 1;

    while(!q.empty()){
        vector<int> temp_res;
        for(int i=0; i<q.size(); i++){
            Node *temp = q.front();
            temp_res.push_back(temp->data);
            if(temp->left)
                q.push(temp->left);
            if (temp->right)
                q.push(temp->right);
            q.pop();
        }
        if(levelNow%2==0)  reverse(temp_res)
        for(int i=0; i<temp_res.size(); i++)
            res.push_back(temp_res[i]);
    }  levelNow++;
```

Write syntax.

return res

#5) Diagonal Traversal

I/P:-



O/P:- 8 10 14 3 6 7 13 1 4

Algorithm

① Push left nodes ~~or~~ in queue.
② Push right nodes to result

Code

```
vector<int> diagonal(Node *root){
    vector<int> res;
    if(root == NULL)
        return res;
    queue<Node*> q;
    q.push(root);
    while(!q.empty()){
        Node *temp = q.front();
        while(temp != NULL){
            if(temp->left)
                q.push(temp->left);
            res.push_back(temp)
            temp = temp->right;
        }
        q.pop();
    }
    return res;
}
```

#6) Boundary Traversal of a BT

(Left + Leaves + Right)
(Bottom to U

Code

```
void leftTree(Node *root, vector<int>& res){
    if(root==NULL) return;
    if(root->left){
        res.push_back(root->data);
        leftTree(root->left, res);
    }else if(root->right){
        res.push_back(root->data);
        leftTree(root->right, res);
    }
}
```

Prints left Tree Nodes. left View.

```
void leaf(Node *root, vector<int>& res){
    if(root==NULL) return;
    if(root->left==NULL && root->right==NULL)
        res.push_back(root->data);
    leaf(root->left, res);
    leaf(root->right, res);
}
```

Prints leaf nodes.

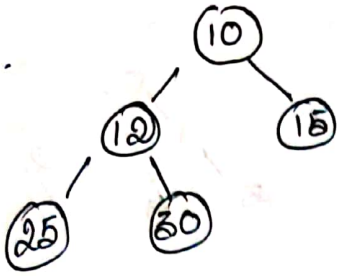
```
void rightTreeBottomUp(Node*root, vector<int>& res){
    if(root==NULL) return;
    if(root->right){
        rightTreeBottomUp(root->right, res);
        res.push_back(root->data);
    }else if(root->left){
        rightTreeBottomUp(root->left, res);
        res.push_back(root->data);
    }
}
```

Prints right view nodes bottom up.

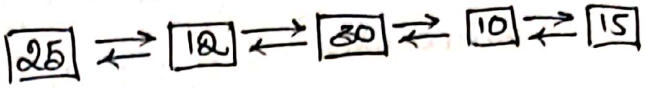
```
vector<int> printBoundary(Node*root){
    vector<int> res;
    if(root==NULL) return res;
    res.push_back(root->data);
    leftTree(root->left, res);
    leaf(root, res)
    rightTree(root->right, res);
    return res;
```

#7) Convert a Binary Tree into a Doubly Linked List

I/P:-



O/P:-



Code (Recursively).

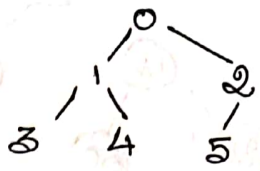
```
Node * btToDLL(Node * root){
    Node *head = NULL
    Node * prev = NULL
    int checkFirst = 0
    Util(root, head, prev, checkfirst);
    return head;
}
```

```
void util(Node * root, Node *&head, Node *&prev,
                                    int &checkFirst){
    if (root == NULL)
        return;
    util(root->left, head, prev, checkFirst)
                        // recursive sol^n for left.
    // Do only for root node.
    if (checkFirst == 0){
        head = root
        prev = root
        checkFirst = 1;
    } else {   // Already nodes present
        prev->right = root;
        root->left = prev;
        prev = root;
    }
    util(root->right, head, prev, checkFirst);
                        // Recurse for right subtree
}
```

#8) Construct Binary Tree from Inorder and Preorder

I/P: inorder = {3, 1, 4, 0, 5, 2}
preorder = {0, 1, 3, 4, 2, 5}

O/P:



inStart

Inorder :- left (x) root right

Preorder :- root left right
preStart

x = index of root in order.

root = preOrder[start]

LeftInorder = InStart → (x-1)
RightInorder = (x+1) → InEnd.

LeftPreend - LeftPreStart
= LeftInEnd - LeftInSta
(x-1) InStart.

∴ LeftPreend = preStart + (x - Insta

LeftPreorder = (PreStart + 1) → (preStart + (x - InStart))
RightPreorder = (preStart + (x - InStart) + 1) → preEnd)

Code

```
Node* buildTree(int in[], int pre[], int n) {
    int inStart = 0; inEnd = n-1;
    int preStart = 0, preEnd = n-1;

    return construct(pre, preStart, preEnd,
                     in, inStart, inEnd)
}

Node *construct(int *pre, int preStart, int preEnd,
                int *in, int inStart, int inEnd){

    if(preStart > preEnd || inStart > inEnd)
        return NULL;

    Node *root = new Node(preOrder[preStart])

    // Find index x of rootVal in Inorder array.
    // Run loop from inStart to inEnd.

    root->left = construct(pre, preStart+1, preStart +
                           (x - inStart), in, inStart, (x-1));
    root->right = construct(pre, preStart + (x - inStart)+1,
                            preEnd, in, (x+1), inEnd);
    return root;
}
```

Imp.

#9) Find minimum no. of swaps to convert BT into BST

Inorder traversal of a BST always gives a sorted array.

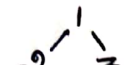
Question is same as minimum of swaps to convert unsorted array into a sorted array.

Idea:- First sort and then swap with original to count no. of swaps.

Code

```
int minSwaps(vector<int>& nums){
    int size = nums.size();
    vector<pair<int, int>> v(size);
    for(int i=0; i<size; i++)
        v[i] = {nums[i], i};   // Insert in vector with orig. index.
    sort(v.begin, v.end);
    int countSwaps = 0;
    for(int i=0; i<size; i++){
        if(i == v[i].second)   // At it's original position.
            continue;
        else{
            countSwaps++;
            swap(v[i], v[v[i].second]);
            i--;   // Recheck; the newly swapped element is in its correct position.
        }
    }
    return countSwaps;
}
```

#10) Check whether leaf nodes of the BT. are at the same level

I/P:- 

O/P: True

① Assign height to first leaf node.

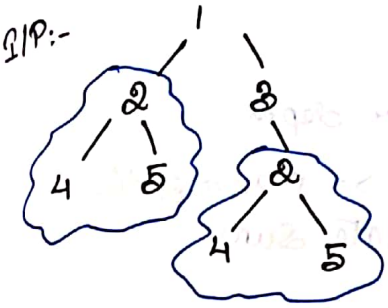
② Check recursively whether all other leaf nodes have same height.

Code

```
bool check(Node* root){
    int checkHeight = -1, height = 0;
    int ans = 1;
    solve(root, checkHeight, height, ans);
    return ans;
}

void solve(Node *root, int &checkHeight, int height,
                                        int &ans){
    if(root == NULL)
        return;
    if (ans==0)          // leaves not at same level
        return;
    if(root->left == NULL && root->right == NULL){
        if(checkHeight == -1)   // for first leaf node
            checkHeight = height;
        else{
            if(checkHeight != height){
                ans = 0;
                return;
            }
        }
    }
    if(root->left)
        solve(root->left, checkHeight, height+1, ans);
    if(root->right)
        solve(root->right, checkHeight, height+1, ans);
}
```

#11) **Check if a Binary Tree contains duplicate subtrees of size 2 or more

I/P:-



O/P:- True.

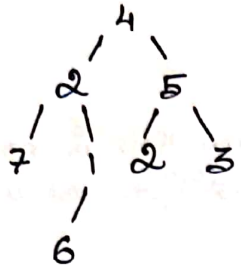
Catch - Whenever there is no node at a position use a delimiter "$".

**Code

```
unordered_map <string, int> hashmap;

string solve(Node* root){
    string s = "";
    if(root == NULL)
        return s + "$";          // return $ at pts where there is no node.
    if(root->left == NULL && root->right == NULL){
        s = to_string(root->data)    // Leaf Node
        return s;
    }

    string sl = solve(root->left);
    string sr = solve(root->right);

    s = s + to_string(root->data) + sl + sr;

    hashmap[s]++;                // if not working write if else condition.
    return s;
}

bool dupSub(root){
    hashmap.clear();
    solve(root);
    for(auto it: hashmap){
        if(it.second >=2)
            return true;
    }
    return false;
}
```

#12) Sum of ~~Leaf~~ Nodes on Longest Path from root to Leaf Nodes.

I/P:-



O/P:- 13.

① check for depth.
② if depth >= maxDepth update sum.

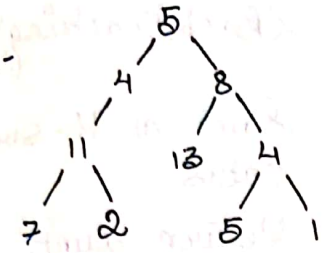
Code

```
int sumRoot_to_Leaf (Node* root){
    int maxSum=0, sum=0;
    int depth=0, maxDepth=0;
    solve(root, sum, depth, maxSum, maxDepth);
    return maxSum;
}

void solve (Node *root, int sum, int depth,
            int &maxSum, int &maxDepth){
    if(root == NULL)
        return;
    // Leaf Node
    if(root->left==NULL && root->right==NULL
                        && depth >= maxDepth){
        maxSum = sum + root->data;
        maxDepth = depth;
        return;
    }
    if(root->left)
        solve(root->left, sum+root->data, depth+1, maxSum, maxDepth);
    if(root->right)
        solve(root->right, sum+root->data, depth+1, maxSum, maxDepth);
}
```

#18) Path Sum - II (LeetCode)

Given the root of a BT and an integer targetSum. Return all root-to-leaf paths where each path's sum = targetSum

I/P:-



targetSum = 22.

O/P:- [[5,4,11,2], [5,8,4,5]]

Similar, all "K" sum paths.

Code

```
vector<vector<int>> pathSum(Node *root, int targetSum){
    if(root == NULL)
        return {};
    vector<vector<int>> paths;
    vector<int> currentPath;
    findPaths(root, targetSum, paths, currentPath);
    return paths;
}
void findPaths(Node *root, int targetSum, vector<vector<int>
                &paths, vector<int> currPath){
    if(root == NULL)
        return;
    currPath.push_back(root->val);
    if(root->left == NULL && root->right = NULL
                && root->val == targetSum);
        paths.pushback(currPath);
        return;
    }
    if(root->left)
        findPaths(root->left, targetSum - root->val, paths
                currentPath);
    if(root->right)
        findPaths(root->right, targetSum - root->val, paths,
                currPath);
}
```

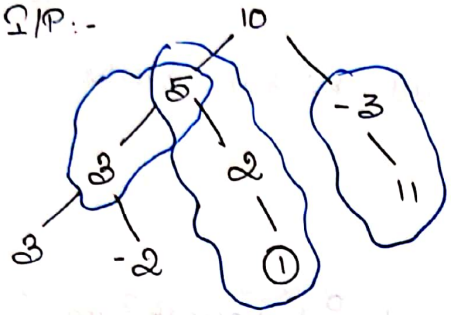
Leaf Node

#14) Path Sum - III

Return the count of paths where sum of the values along the path equals targetSum.

<Backtracking>

I/P:-



targetSum = 8

O/P:- 3

Same as k-sum paths.
Return count.

Code

```
int count = 0;
vector<int> vec;
int countPath(Node* root, int targetSum){
    helper(root, targetSum);
    int count;
}
```

Util function

```
//void helper(Node *root, int targetSum){
    if(root == NULL)
        return;
    vec.push_back(root->val);
```

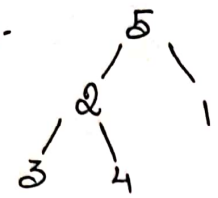
Recurse

```
//helper(root->left, targetSum);
helper(root->right, targetSum);
    int currSum = 0
    for(int i = vector.size()-1; i>=0; i--){
        currSum += vec[i];
        if(currSum == targetSum)
            count++;
    }
    vec.pop_back(); //Backtrack.
}
```

// If return path asked pushback in a vector

#15) Lowest Common Ancestor

I/P :-

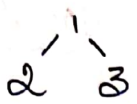


n1 = 3 n2 = 4. O/P - 2

Code

```
Node * lca (Node * root, int n1, int n2) {
    if (root == NULL)
        return NULL

    // If n1 or n2 matches with root -> val; root is LCA
    if (root -> data == n1 || root -> data == n2)
        return root;

    // Recurse
    Node * leftLCA = lca (root -> left, n1, n2)
    Node * rightLCA = lca (root -> left, n1, n2).

    // If both leftLCA and rightLCA exists; n1 and n2
    are on opposite sides. So the root is LCA.
    if (leftLCA and rightLCA)
        return root;

    // If only leftLCA; return it.
    if (leftLCA)
        return leftLCA;
    else
        return rightLCA;
}
```

#16) Distance b/w 2 nodes in a binary tree.

I/P:- 

a = 2 ; b = 3 ;

O/P:- 2.

① find LCA

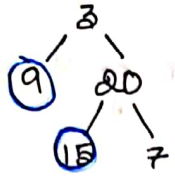
② distance 1 + distance 2
(LCA and a) (LCA and b)

Code

```
int findDist (Node *root, int a, int b) {
    Node* LCA = lca(root, a, b);
    int dist1 = dis(LCA, a, 0);   // Dist. b/w LCA and a;
    int dist2 = dis(LCA, b, 0);   // LCA and b;
    return dist1 + dist2;
}

int dis (Node* root, int a, int hgt) {
    if (root == NULL)   // If "a" not found in a subtree.
        return -1;
    if (root -> data == a)
        return hgt;
    int lh = dist(root -> left, a, hgt+1);
    int rh = dist(root -> right, a, hgt+1);
    return max(lh, rh);   // Either lh or rh will be -1. (not found)
}                         // So, we are taking max
```

#17) Sum of left leaves

I/P:-



O/P:- 24

Code

```
int sumofleftleaves (Node * root) {
    if (root == NULL)
        return 0;
    else if (root -> left != NULL && root -> left -> left == NULL
                                  && root -> left -> right == NULL)
        return root -> left -> val + sum of (root -> right)
    else
        return sumof(root -> left) + sum of (root -> right);
}
```

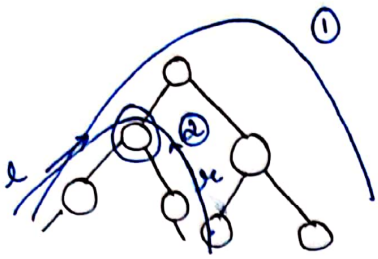
#18) Tree Isomorphism Problem

Code

```
bool isIsomorphic (Node * root1, Node * root2) {
    if (root1 == NULL && root2 == NULL)  return true;
    if (root1 == NULL || root2 == NULL)  return false;
    if (root1 -> data != root2 -> data)  return false.

    bool check1 = isIsomorphic (root1 -> left , root2 -> left)
                                  &&
                  isIsomorphic (root1 -> right, root2 -> right);
    bool check2 = isIsomorphic (root1 -> left, root2 -> right)
                                  &&
                  isIsomorphic (root1 -> right, root2 -> left);

    return (check1 || check2);
}
```

DP on Trees

#19) Maximum Path Sum from any node to any node



① Max path sum can go through the root.

OR

② Max. path sum can go through left subtree alone

① If max sum path is through root node. intermediate node;

max { < max(ls, rs) + root → val > passes through root.
or
< root → val > → if both ls, rs are negative

② If left subtree is along path of answer.

(ls + root → val + rs).

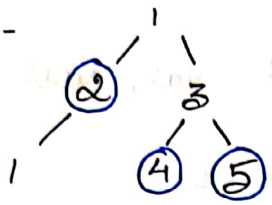
Code

```
int maxPathSum (Node* root) {
    if (root == NULL)
        return 0;
    int res = INT_MIN;
    solve(root, res);
    return res;
}

int solve (Node* root, int & res) {
    if (root == NULL)
        return 0;
```

```
int ls = solve(root->left, res);
int rs = solve(root->right, res);

// Max sum path through root.
int temp = max(max(ls, rs) + root->val, root->val);

// Max sum path through left subtree.
int ans = max(temp, ls + root->val + rs);

int res = max(ans, res);

return temp;  // Pass on temp to higher node.
}
```

#20) Max. Sum of Nodes in a BT such that no 2 are adjacent (DP on trees)

I/P -



O/P - 11

① If root node included. → take sum of grandchildren
② If root node excluded → sum of children.

Code

```
unordered_map<node*, int> dp;
int maxSum(Node * root){
    if(root == NULL)
        return 0;

    // Memoization
    if(dp[root])
        return dp[root];

    int inc = root->data;   // root included
    if(root->left)
        inc += maxSum(root->left->left) + maxSum(root->left
                                                 ->right);
    if(root->right)
        inc += maxSum(root->right->left) + maxSum(root->right
                                                  ->right);
    // Root Not included
    int exc = maxSum(root->left) + maxSum(root
                                          ->right);

    dp[root] = max(inc, exc);

    return dp[root];
}
```