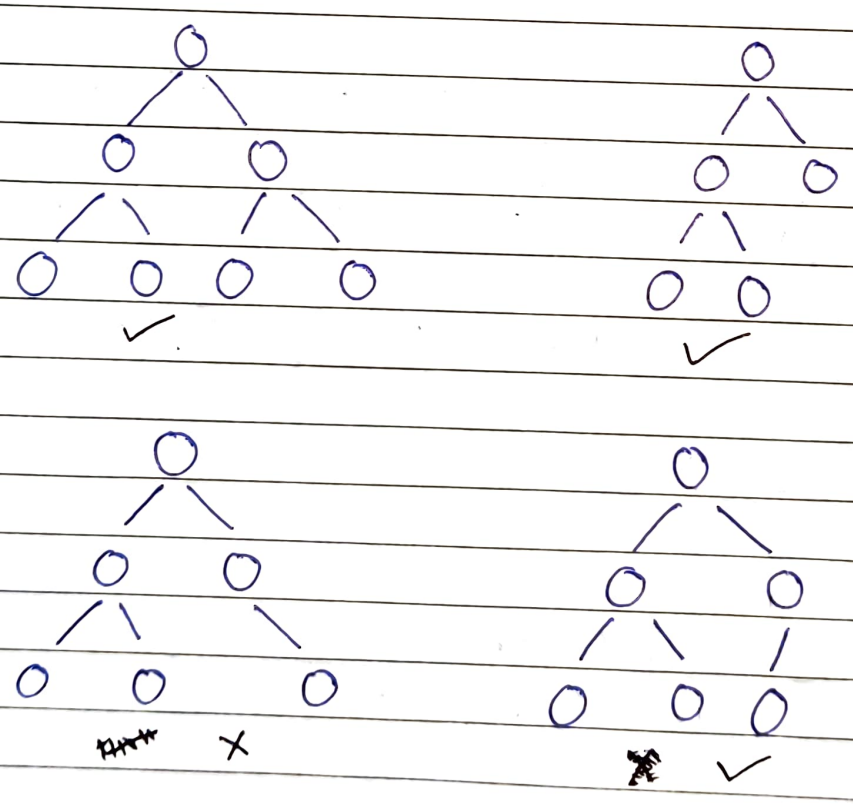
* Types of Binary Trees

→ Full Binary Tree
Either has 0 or 2 children

(ii) Complete Binary Tree
→ All levels are completely filled except the last level

→ The last level has all nodes in left as possible.

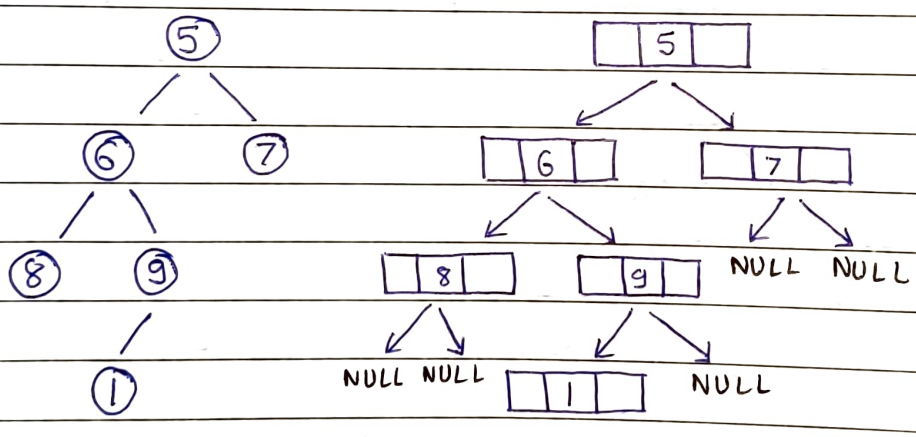


(iii) Perfect Binary Tree
All the leaf nodes are at same level.

(iv) Balanced Binary Tree
Height of tree at max log (n)

(v) Degenerate / Skewed Binary Tree
Each node has single children

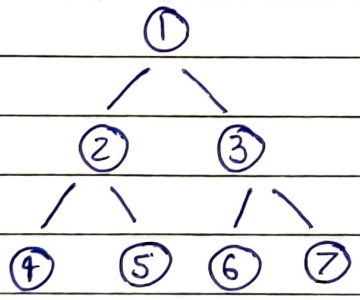
Binary Tree Representation




```
class Node {
public :
    int data;
    class Node* left;
    class Node* right;

    Node (int val) {
        data = val;
        left = right = NULL;
    }
};
```

```
int main () {

    class Node * root = new Node (1);

    root -> left = new Node (2);
    root -> right = new Node (3);
    root -> left -> right = new Node (4);

    return 0;
}
```

* DFS → Inorder
* DFS → Preorder
* DFS → Postorder



Inorder (Left Root Right)
4 2 5 1 6 3 7

Preorder (Root Left Right)
1 2 4 5 3 6 7

Postorder (Left Right Root)
4 5 2 6 7 3 1

Preorder Traversal

```
if (node == NULL) return;

print (node → data);

preOrder (node → left);
preOrder (node → right);
```

Inorder Traversal

```
inOrder (node → left);

print (node → data);

inOrder (node → right);
```

Postorder Traversal

```
postOrder (node → left);
postOrder (node → right);

print (node → data);
```

Level Order Traversal

```
2d vector ans ;
if (root == NULL) return ans ;

queque < Node * > q ;
q. push (root) ;

while (! q. empty ()) {

    int size = q. size () ;

    vector < int > level ;

    for ( 0 → size -1) {
        Node* node = q. front () ;
        q. pop () ;

        if (node → left != NULL)
            q. push (node → left) ;
        if (node → right != NULL)
            q. push (node → right) ;

        level. pb (node → val) ;
    }

    ans. pb (level) ;

}
```

Auxiliary Space in Recursive DFS

→ If we donn't consider the size of the stack for f^n calls then O(1) otherwise O(h) where h is the height of the tree.

→ The height of Balanced Tree is (log n) so the best time complexity is O(log n).

→ The height of skewed tree is n (no. of elements) so the worst time complexity is O(n).

Iterative Preorder Traversal

```
while (! s.empty ()) {

    Node * node = s. top ();
    s. pop ();

    if (node → right) != NULL)
        s. push (node → right);
    if (node → left != NULL)
        s. push (node → left);
}
```

Iterative Inorder Traversal

```
TreeNode * node = root;

while (!s.empty())

while (true) {

    if (node != NULL) {
        s.push (node);
        node = node -> left;
    }

    else {
        if (s.empty ()) break;

        node = s.top();
        s.pop();

        ans.pb (node -> val);

        node = node -> right;
    }

}
```

Iterative Postorder Traversal using 2 stacks

```
while (! s1. empty()) {

    Node * node = s1. top();
    s1. pop ();

if (node -> left != NULL)
    s1. push (node -> left);

if (node ->right != NULL)
    s1. push (node -> right);

    s2. push (node);

}

while (! s2. empty()) {

    print (s2. top() -> val);
    s2. pop ();

}
```

Iterative Postorder Traversal

```
Node * node = root -> left ;

while (node != NULL || !s.empty ()){

    if (node != NULL) {
        s.push (node);
        node = node -> left;
    }
    else {

    Node * temp = s.top () -> right;

    if (temp == NULL) {

        temp = s.top ();
        s.pop ();
        ans.push_back (temp -> val);

        while (!s.empty && temp
        == s.top -> right) {

            temp = s.top ();
            s.top ();
            ans.push_back (temp -> val);
        }

    } else node = temp;
}
```

All traversal in one traversal

```
stack <pair < Node*, int >> s;
s. push ({root, 1});

while (! s. empty ()) {

    auto it = s. top ();
    s. pop ();

    if (it. ss == 1) {
        pre. pb (it.ff -> val);
        it. ss ++;
        s. push (it);

        if (it.ff -> left != NULL){
            s. push ({it. ff -> left, 1});
        }
    }
    else if (it. ss == 2) {
        in. push_back (it. ff -> val);
        it. ss ++;
        s. push (it);

        if (it. ff -> right != NULL) {
            s. push ({it. ff -> right, 1});
        }
    }
    else post. pb (it. ff -> val);
```

Maximum Depth in Binary Tree

```
1 + max (left, right);
```

Balanced Binary Tree

→ The left and right subtree of every node differ in height by no more than 1.

Diameter of a binary Tree

→ length of longest path betn any two nodes

```
ans = max (ans, left + right);
```

Maximum Path Sum

```
left = max (0, left);
right = max (0, right);

curr = (root → val) + left + right;
ans = max (ans, curr);

return (root → val) + max (left, right);
```

Zig - Zag or Spiral Traversal

```
vector <int> level (size);
for ( 0 → size -1) {
    index = (right To Left) ? size -i -1
                : i ;
    level [index] = node →val;
}
```

Vertical Order Traversal of Binary Tree

* Approach I

→ Run Normal DFS

→ Store value in vectors of
vector << col, row >, root →val >;

→ Sort the vector.

* Approach II

→ map < int, map <int, multiset <int>> store;
queque < pair < TreeNode*, pp <int, int>> q;

→ store [x][y]. insert (node → val);

For left : x-1, y+1
For right : x+1, y+1

→ For (auto x : store) {
    vector <int> level;
    for (auto y : x.ss) {
        For (auto z : y.ss)
            level. pb (z);
    }
    ans. pb (level);
}

Boundary Traversal (Anti Clock)

→ Root should be included in main fn.

Otherwise, if root has only left or right child, then, in this case ans will be wrong.

In recursive, for right boundary, call the next fn first, since we are storing it from bottom to top.

In iterative, check every time before storing the value, whether it is leaf node or not.

Top / Bottom view of Binary Tree

→ For this approach, we have to use level order traversal.

→ Whether, it is top view or down / bottom view, for a particular level or height, we need extreme node's value, i.e extreme left and extreme right.

→ queue <pair < Node*, int >> q;
(col)

→ Why only 1 int in q ⇒ Not 2 int i.e for row and col.
Because of this, we used DFS over BFS.

* For top view,

```
if (m.count(col)) m[col] = curr->val;
```

* For bottom view,

```
m[col] = curr->val;
```

Right / Left view of Binary Tree

→ solve (Node* root, int height);

→ height = row;

→ For a particular height, the root which comes 1st, is it's answer.

if (ans.size() == heigh) ans.pb(root->val);

* For right view;

solve (root->right ; height+1);
solve (root -> left, height+1);

* For left view;

solve (root -> left, height+1);
Solve (root -> right, height->1);

Date
Page

Same tree and Symmetric Tree

* **Base condition** for both are same

```
if (root1 == NULL && root2 == NULL)
        return true;
if (root1 == NULL || root2 == NULL)
        return false;
if (root1 -> val != root2 -> val);
        return false;
```

* **Same Tree**

```
ans1 = solve (root1 -> left, root2 -> left);
ans2 = solve (root2 -> right, root2 -> right);
```

* **Symmetric Tree**

```
ans1 = solve (root1 -> right, root2 -> left);
ans2 = solve (root2 -> left, root2 -> right);
```

* Again, **return** is same for both;

```
retun (ans1 && ans2);
```

Binary Tree Paths

* root is given return all root to leaf paths

* if (root == NULL) return;

```
if (s.empty()) s += tostring(root->val);
else s += "->" + tostring(root->val);

if (isLeaf(root)) {
    ans.push_back(s);
    return;
}

solve(root->left, s, ans);
solve(root->right, s, ans);
```

→ By careful, don't pass string s by pass by reference.

* For pass by reference, s;

```
void change(string &s){
    while(!s.empty()){
        if(s.back() != '>') s.pop_back();
        else break;
    }
```

```
    if (!s.empty && s.back() == '>') {
        s.pop_back();
        s.pop_back();
    }
}
```

```
if (isLeaf(root)){
    ans.push_back(s);
    change(s);
    return;
}

solve();
solve();

change(s);
```

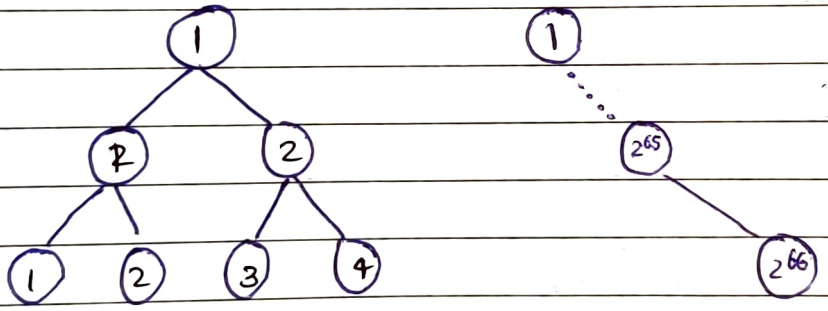
Lowest Commont Ancestor

```
if (root == NULL || p == root || q == root)
                    return root;

Node* left  = solve(root->left, p, q);
Node* right = solve(root->right, p, q);

if (root == NULL) return right;
else if (root == NULL) return left;
return root;
```

Maximum width of Binary Tree

→ We can assign value for nodes as (2*index +1) or (2*index +2) just like segment trees.
But, this gonna overflow

→ After little optimising, now try to assign values from 1, 2, 3... from left to right for every level
But, this still gonna overflow.



→ Now, let
curr Index = Index - prevMin Index;

Now, do (2*curr Index +1) and (2*curr Index +2)

For, this we have to use 'DFS' ⇒ we required prevMin Index

For this, in skew Trees, curr Index is 0.

Children Sum Property

→ For a partical node,

value ≥ max (parent, left + right);

→ if (root == NULL) return 0;

int curr = max (root → data, prev);

int left = update (root → left, curr);
int right = Updata (root → right, curr);

(root → data) += max (curr, left + right) -
(root → data);

return (root → data);

All Nodes Distance K in Binary Tree

* The main problem in the question that here, for any particular node, we cann't tell who is it's parent.

Approach I

→ Run a BFS ⇒ For a particular node ⇒ Store who it it's parent and child ⇒ store the node which is connected to it ⇒ Just like CP Problem in 2D vector

→ Now, using DFS, calculate the distance of every node from target.

Approach II

→ map < Node*, Node* > m;

Store the parent of every node in this map.

→ map < Node*, int > dis;

Now, run DFS ⇒ Along with children, now insert both parent and children for particular node.

→ Before inserting or updating the distance, check whether it is present in "dis" or not.
Otherwise, it will run forever.

Minimum Time Taken to BURN Binary Tree

* Similar Approach ⇒ Print all the Nodes at a distance K in Binary Tree

Construct Total Nodes in Complete Binary Tree

```
int leftHeight (node * root) {
    int height = 0;
    while (root != NULL) {
        height ++;
        root = root -> left;
    }
    return height;
}

int count (node* root) {
    if (root == NULL) return 0;

    int lh = leftHeight (root);
    int rh = rightHeight (root);

    if (lh == rh) return pow (2, lh) - 1;

    int left = count (root -> left);
    int right = count (root -> right);

    return left + 1 + right;
}
```

→ if (lh == rh)

This means that node is a parent of that subtree whose child has two childs except the leaf node.

This is because the given tree is complete binary Tree.

→ $TC = (\log N)^2$

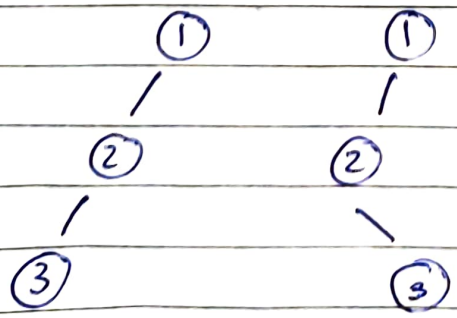
Time complexity to calculate the height = $(\log N)$

No. of nodes at particular level = $(\log N)$

Q. Can we construct a UNIQUE Binary Tree with following?

(a) Pre Order & Post Order

→ NO



(Root L R)

PreOrder = 1 2 3

Post Order = 3 2 1

(L R Root)

(b) Inorder & Pre Order

→ Yes

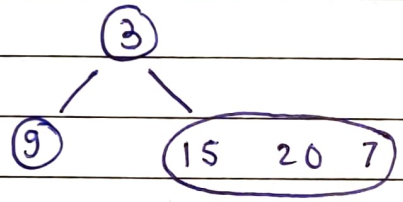
→ 3 must be the root of the tree.

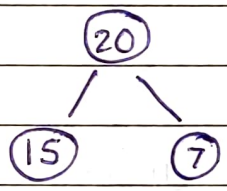
(L Root R)

In Order = 9 3 15 20 7

Pre Order = 3 9 20 15 7

(Root L R)





(c) In Order & Post Order

→ Yes

Construct Binary Tree from Inorder and Postorder

* Initial value of k = n-1 ⇒ Not 0 and k-- in every step

*
```
if (mid + 1 <= j) {
    root -> right = new TreeNode;
    build ();
}
```

Here, right is called first.

Construct Binary Tree from inorder and preorder

```
void build ( ) {
    root -> val = preorder [k];
    if (i == j) {
        k++;
        return;
    }

    int mid = index (preorder [k]];

    k++;
    if (i <= mid-1) {
        root -> left = new TreeNode ();
        build (root -> left, i, mid-1, k );
    }
    if (mid+1 <= j) {
        root -> right = new TreeNode();
        build (root -> right, mid+1, j, k);
    }
}

TreeNode * build ( ) {
    TreeNode * root = new TreeNode();
    int n = preOrder. size (), k = 0;

    for (0 -> n-1) index [inorder [i]] = i;

    build (root, 0, n-1, k, index );
}
```

Serialize and Deserialize Binary Tree

* Level Order Traversal ⇒ For Both Serializing and deserializing
* During Serializing ⇒ IF NULL insert "#" (otherwise " * val ".
* During Deserializing ⇒ Pay attention to negative values.

i.e * -12 # # * -13 * 15 #
↓
minus

Morris Traversal

* If curr == NULL ⇒ Break the loop
* If curr → left == NULL
  → Print curr
  → curr = curr → right

If the curr is not leaf ⇒ curr → right is already connected with root of that subtree.

* Else
  next = curr → left

Find the rightmost node of next. Suppose it is 'x'.

(i) If x → right == NULL

→ x → right = curr

curr = curr → left;

(ii) Else

x → right == NULL;

curr = curr → right

→ remove the thread and move to right because left part is already traversed.

* The algorithm is such that intial state of the tree is restored after traversing.

* Each node is visited 3 times

→ to visit

→ to find its rightmost node

→ make rightmost node == NULL

```
vector <int>  preOrder (Node *  root) {

    vector <int> preorder;
    if (root == NULL) return  preorder;

    while (root != NULL) {

        if (root -> left == NULL) {
            preorder. pb (root -> val);
            root = root -> right;
        } else {

            Node *  next = root -> left;

            while (next -> right != NULL &&
                   next -> right != root) {
                next = next -> right;
            }

            if (next -> right == NULL) {
                next -> right = root;
                preorder. pb (root -> val);
                root = root -> left;
            } else {
                next -> right = NULL;
                root = root -> right;
            }
        }
    } return  preorder;
}
```

Inorder

```
if (next -> right == NULL){
        next -> right = root;
        root = root -> left;
} else {
        next -> right = NULL;
        inorder. pb (root -> val);
        root = root -> right;
}
```

Flatten a Binary Tree to Linked List

(i) Stack

→ Run normal preorder

→ if (curr → right != NULL) s.push (curr → right);
if (curr → left != NULL) s.push (curr → left);

curr → left = NULL;
if (!s.empty()) curr → right = s.top();
else curr → right = NULL;

(ii) Recursion

```
Node * prev = NULL;
void flattern (Node * root) {
    if (root == NULL) return;

    flattern (root → right);
    flattern (root → left);

    root → left = NULL;
    root → right = prev;
    prev = root;
}
```

(iii) Morris Traversal

```
void flatten (Node* root) {

    Node * curr = root;

    while (curr != NULL){
        if (curr -> left != NULL) {

            Node* next = curr -> left;

            while (next -> right != NULL)
                next = next -> right;

            next -> right = curr -> right;
            curr -> right = curr -> left;
            curr -> left = NULL;
        }
        curr = curr -> right;
    }
}
```