(Binary Tree) → each node can have at most two children nodes

⇒ Types :-

1) full Binary Tree :- every node has either zero or two children.

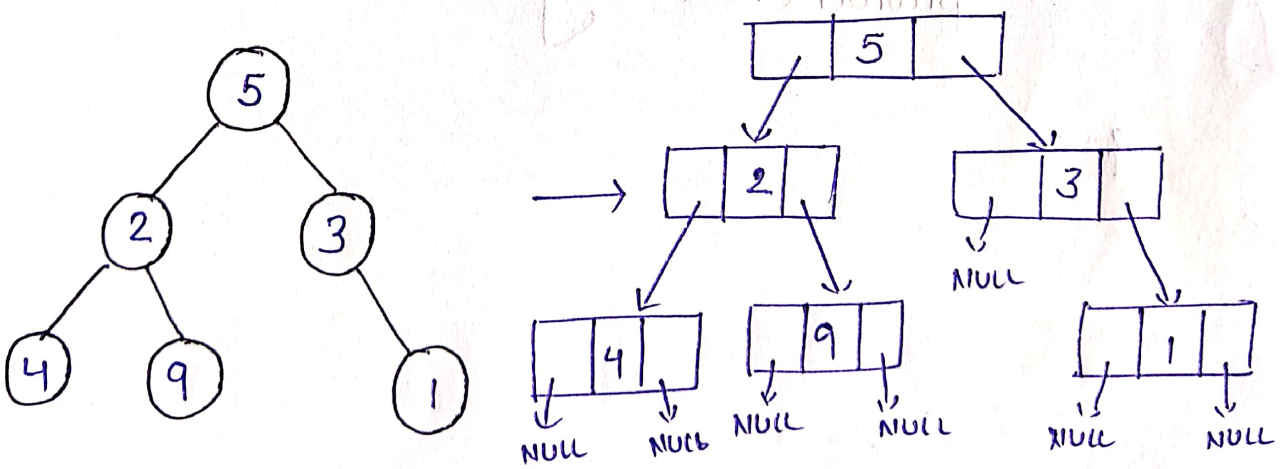
2) complete Binary Tree :- all levels are filled completely except possibly the last level, which is filled from left to right

3) Perfect Binary Tree :- all leaf nodes are at the same level & the no. of leaf nodes is maximised for that level.

4) Balanced Binary Tree :- height of the two subtrees of any node differ by at most one.

5) Degenerate Tree :- nodes are arranged in a single path leaning to the right or left

⇒ In code —

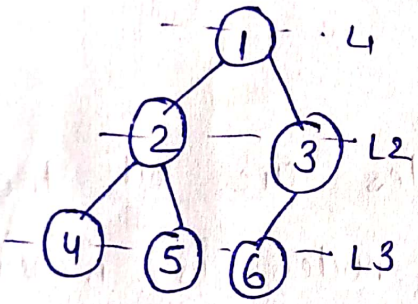


Traversal Techniques (BFS / DFS) :-

⇒ Depth first Search —

1) Preorder (root left right)
2) Postorder (left right root)
3) Inorder (left root right)

⇒ Breadth first Search —



⇒ 1 2 3 4 5 6

⇒ 1) Preorder Traversal :- (Root left Right)

```
void preorder(node) {
    if (node == null)   return;
    print(node → data)
    preorder(node → left)
    preorder(node → right)
}
```

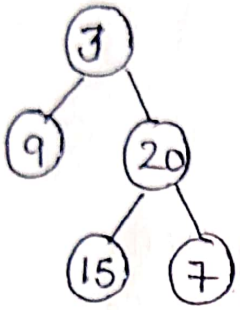
2) Inorder traversal :-

```
void Inorder (node) {
    if (node == null) return;
    Inorder (node → left);
    printf (node → data);
    Inorder (node → right);
}
```

3) Postorder Traversal :-

```
void postorder (node) {
    if (node == null) return;
    postorder (node → left);
    postorder (node → right);
    printf (node → data);
}
```

⇒ Level order Traversal :- (Queue)



I/P = [3, 9, 20, null, null, 15, 17]

O/P = [[3], [9, 20], [15, 7]]

⇒ Code :-

```
if (root == null)  return ans
queue<TreeNode*> q
q.push(root)
while (!q.empty()) {
        for(i=0 to size of q) {
            node = q.front
            q.pop()
            if(node->left!=null)  q.push(node->left)
            if(node->right!=null) q.push(node->right)
            level.push_back(node->val)
        }
    ans.push_back(level);
}
return ans
```

TC :- O(n)

SC :- ≤ O(n)

* Iterative Traversal using stack :-

1) Preorder :-

```
stack s
if(root == null) return ans
s.push(root)
While(!s.empty()){
    root = s.top();
    s.pop();
    ans.push_back(node->val);
    if(root->right != null) s.push(root->right)
    if(root->left != null) s.push(root->left)
}
return ans
```

TC :- O(n)

SC :- O(2n)

2) Inorder :-

```
stack st
node = root
while(true){
    if(node != null){
        st.push(node)
        node = node->left
    }
    else{
        if(st.empty() == true) break
        node = st.top()
        st.pop()
        ans.push_back(topNode->val);
        node = node->right;
    }
}
```

TC :- O(n)

SC :- O(2n)

3) Postorder :-

a) using Two stack :-

```
stack s1, s2;
s1.push(root)
while(!s1.empty()){
    node = s1.top();
    s1.pop();
    s2.push(node);
    if(node→left) s1.push(node→left);
    if(node→right) s1.push(node→right);
}
// s2 se elements pop krke ans me dalo
while(!st2.empty()){
    ans.push_back(st2.top()→val);
    st2.pop();
}
return ans
```

TC :- O(2n)
SC :- O(2n)

b) using one stack :- (using ~~recu~~ reverse())

```
if(root==null) return ans;
stack s1;
s1.push(root)
while(!s1.empty()){
    node = s1.top();
    s1.pop();
```

TC :- O(2n)
SC :- O(n)

```
ans.push_back(node->val);
if(node->left) st1.push(node->left);
if(node->right) st2.push(node->right);

// reverse the order of elements to get postorder
reverse(ans.begin(), ans.end());
}
```

⇒ Without reverse using one stack :-

```
curr = root
stack st;
while(curr!=null || st.empty()){
    if(curr!=null){
        st.push(curr)
        curr = curr->left
    }
    else{
        temp = st.top()->right;
        if(temp==null){
            temp = st.top();
            st.pop();
            ans.push_back(temp->data);
            while(!st.empty() && temp==st.top()->right)
            {
                temp = st.top();
                st.pop();
                ans.push_back(temp->data);
            }
        }
        else{
            curr = temp;
        }
    }
}
```

TC :- O(n)

SC :- O(n)

⇒ Postorder, Inorder & Preorder Traversal in one Traversal :-

⇒ a node is visit three times in any Traversal.

Let, a node 'x' — located deep in the leftmost part of binary Tree.

first visit: This is the point where we can consider 'x' to be in the preorder phase (because root is before left & right).

second visit: When left subtree is null return to 'x' Now, we process the current node which is inorder (left → root → right)

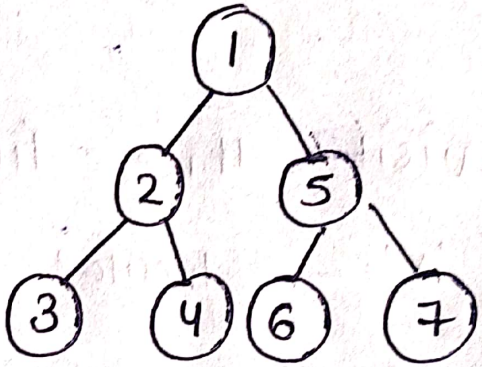
Third visit :- When right subtree is null return to 'x'. the node has finished processing both children, & we consider it postorder (because left → right → root)

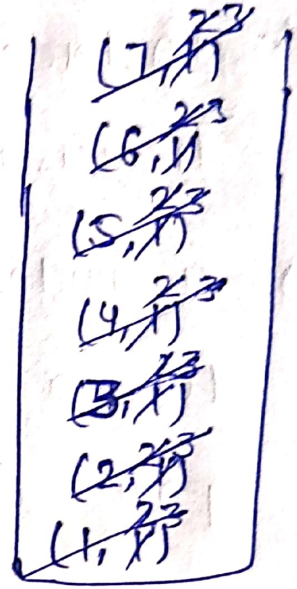
⇒ Recursive :-

```
trav(TreeNode, in, pre, post) {
    if(!root) return
    pre.push(root → data)
    trav(root→left, in, pre, post)
    in.push(root→data)
    trav(root→right, in, pre, post)
    post.push(root→data)
}
```

⇒ Iterative :-

```
if (num==1) {
    preorder
    ++
    left }
if (num==2) {
    inorder
    ++
    right
}
if (num==3)
    postorder
```





(node, num)

Preorder – 1 2 3 4 5 6 7

Inorder – 3 2 4 1 6 5 7

Postorder – 3 4 2 6 7 5 1

TC :- O(3n)

SC :- O(n)