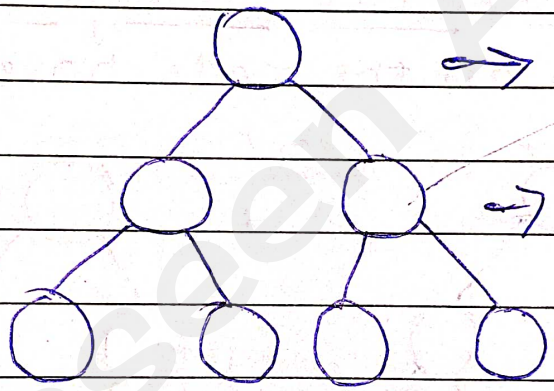
→ Binary trees: It is a non-linear data structure which stores data in a hierarchial form.
A tree which has almost 2 children is called a binary tree.

* properties of Binary tree:

1. Maximum nodes at level L = 2^L

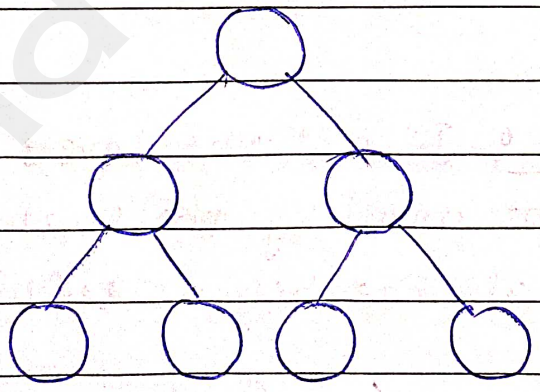


→ Level-0 : $2^0 = 1$

→ level-1 : $2^1 = 2$

→ Level-2 : $2^2 = 4$.

2. Maximum nodes in a tree of height H is $2^H - 1$.



∴ $2^3 - 1 = 8 - 1 = 7$ nodes

Height H: 3

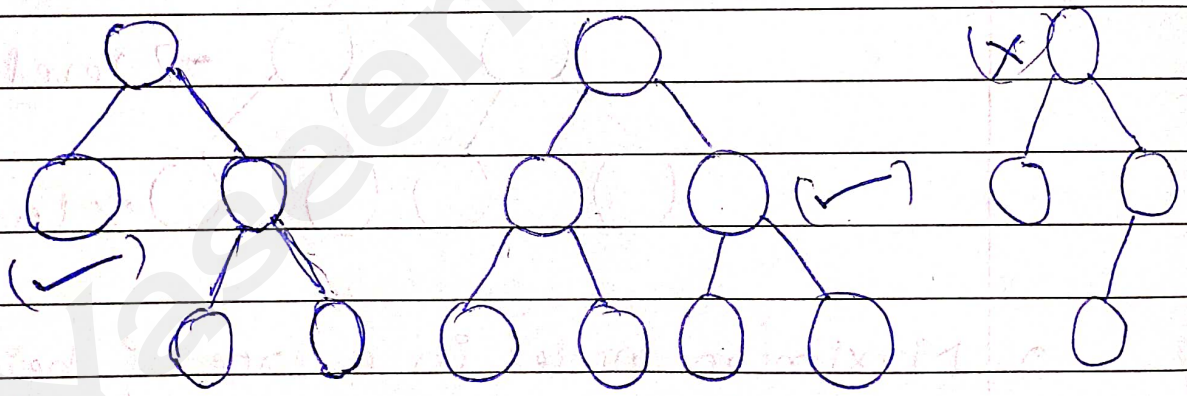
3. For N nodes, minimum possible height or the minimum number of levels is $\log_2(N+1)$

⇒ Types of Binary Trees :

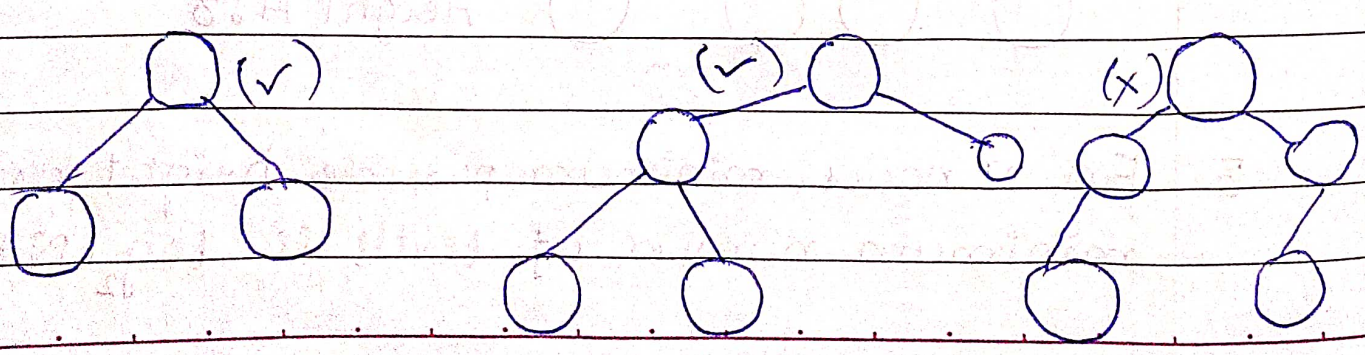
1. Full / proper / strict Binary tree.
2. Complete Binary tree
3. Perfect Binary tree.
4. Balanced Binary tree
5. Degenerate Binary tree.

* **Full / proper / strict Binary tree :** It is a Binary in which all the node have 0 or 2 children.

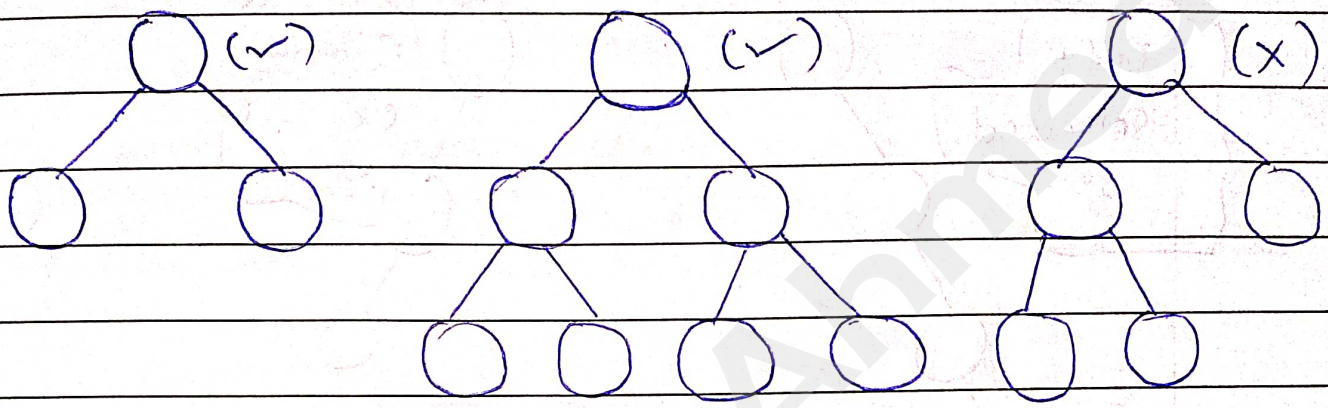
ex:



* **Complete Binary tree :** If all the levels are completely filled except possibly the last level & the last level has all the key as left as possible

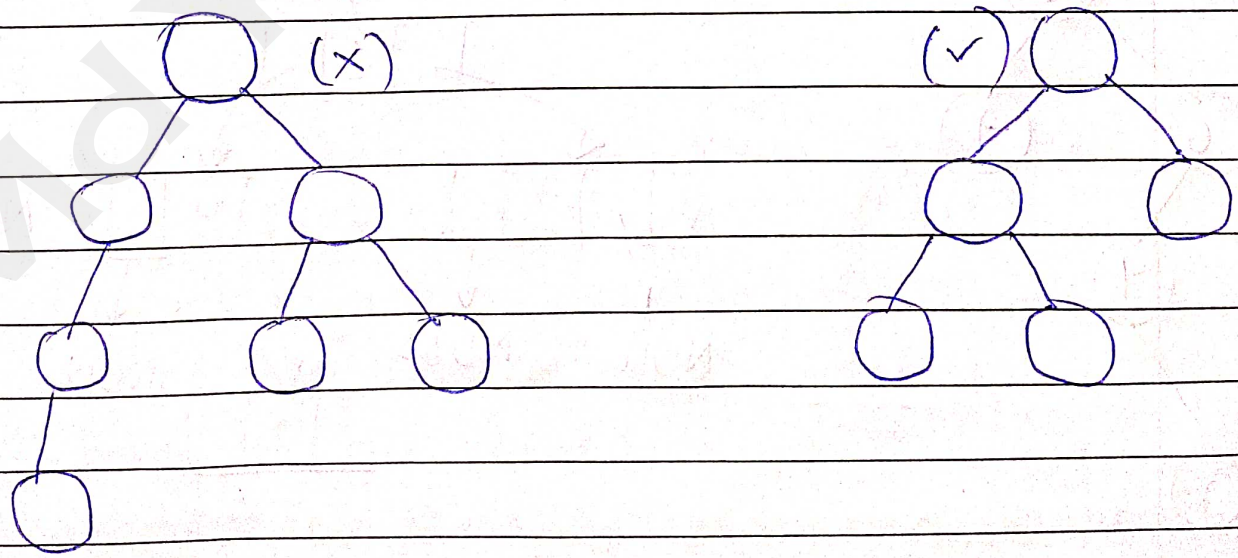


* Perfect Binary tree: A tree in which all the internal nodes have two children and all the leafs are at same level.

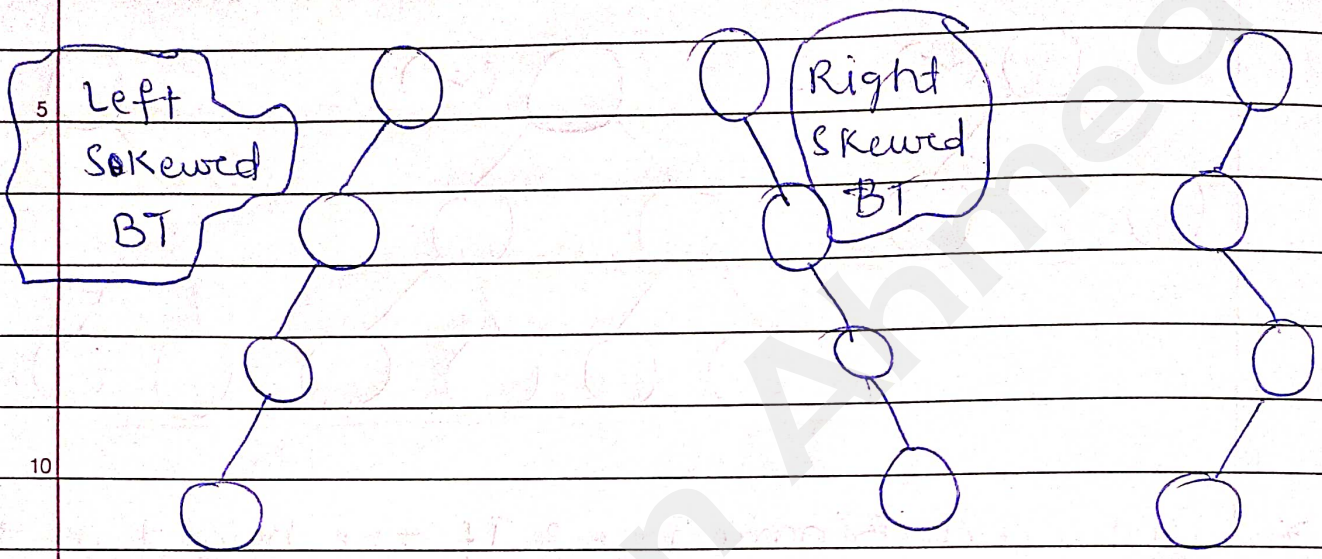


* Balanced Binary Tree: If the height of tree is O(log N) where, N is the number of nodes

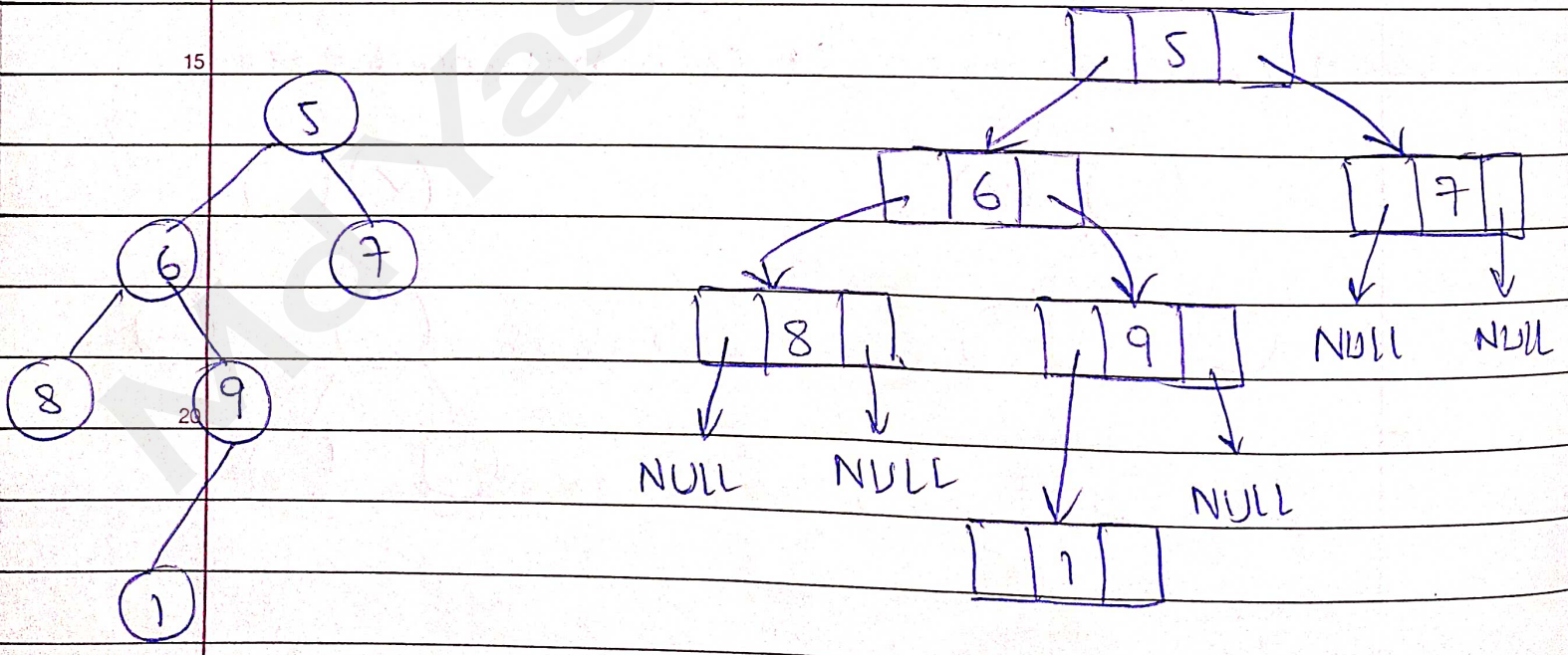
n=8, $\log_2 8 = 3$, If The maximum height of the tree is 3 then it is a balanced tree.



*. Degenerate tree: A tree in which every internal node has only 1 child. Such trees are similar to the linked list performance wise.



=> Binary tree Representation:



*: Structure definition in CPP:

```
Struct Node
{
    int data;
    Struct Node* left;
    Struct Node* right;

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

int main()
{
  Struct Node* root = new Node(5);
  root->left = new Node(6);
  root->right = new Node(7);
  root->left->left = new Node(8);
  root->left->right = new Node(9);
  root->left->right->left = new Node(1);

  return 0;
}
```

Binary tree traversals:

(1) DFS: [Depth First Search]:

1. preorder traversal.
2. Inorder traversal
3. postorder traversal.

(2) BFS: [Breadth First Search]

1. Level order traversal.

Level order traversal:

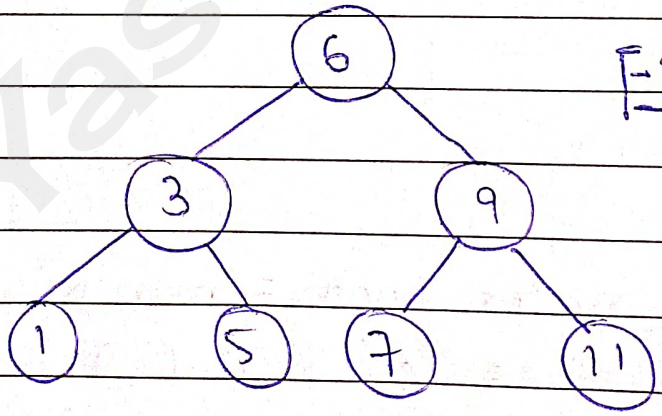


Fig: Tree (1.1)

Level order: 6, 3, 9, 1, 5, 7, 11

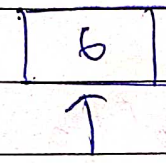
```
vector<int> levelOrderTraversal(Node* root)
{
    vector<int> res;
```

```
if (root == NULL)
    return res;
queue<Node*> q;
q.push(root);

while(!q.empty())
{
    int n = q.size();
    for(int i=0; i<n; i++)
    {
        Node* tempNode = q.front();
        q.pop();
        if(tempNode->left != NULL)
            q.push(tempNode->left);
        if(tempNode->right != NULL)
            q.push(tempNode->right);
        res.push_back(tempNode->data);
    }
}
return res;
}
```

Ex: let us visualize how the above function works on Fig(1.1)

(1) Root is Not null, then root node is inserted into queue q.

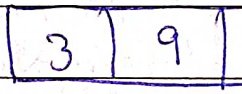
Now, q.size() = 1



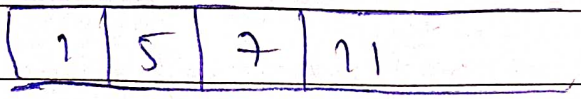
(2) tempNode = 6

Now, it checks is 6->left != NULL yes it is so, it add the node to the queue, similarly for the right of root (i.e., 3 and 9 are added to the queue)

q.size() = 2.



(3) Now, in this iteration, 3 & 9 are stored in the res vector & their childrens are pushed/inserted into the queue.



(4) In this iteration, no node has a child so, simply they all are inserted into the res vector & q is also becomes empty so, it will come out of the while loop & returns the res vector.

-> Minimum depth of a Binary tree : The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node.

```
int minDepth(Node* root)
{
    if (root == NULL) // Empty tree
        return 0;
    // only Root node is present then,
    if (root->left == NULL && root->right == NULL)
        return 1;

    int l = INT_MAX, r = INT_MAX;

    if (root->left)
        l = minDepth(root->left);

    if (root->right)
        r = minDepth(root->right);

    return 1 + min(l, r);
}
```

Maximum depth or height of a Binary tree:

A maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

```
int maxDepth(Node* root)
{
    if (root == NULL)
        return 0;
    else
        return 1 + max(maxDepth(root->left),
                       maxDepth(root->right));
}
```

Iterative Pre-order Traversal:

[Root, Left, Right]: The idea is simple first if root is not empty then create a stack of type Node* push root on top of stack. Now, If or while stack is not empty do the following:

1. pop the node from the top of stack & store it in a temporary variable say tempNode
2. Now, check if tempNode has a right child if yes then push it on top of stack.
3. Similarly, check if tempNode has a left child if yes, then push it on top of stack.
4. push the tempNode → data into the resultant vector.

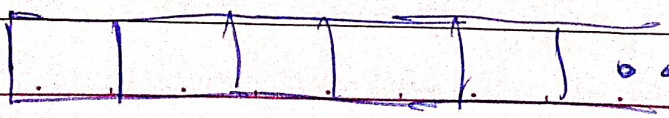
As soon as the stack becomes empty, come out of the loop and return the resultant vector.

```
* vector<int> preorder (Node* root)
{
    if (root == NULL)
        return 0;
```

```
vector<int> res;
stack<int> st;
st.push(root);

while (!st.empty())
{
  Node* tempNode = st.top();
  st.pop();

  if (tempNode->right)
    st.push(tempNode->right);
  if (tempNode->left)
    st.push(tempNode->left);
  res.push_back(tempNode->data);
}

return res;
}
```

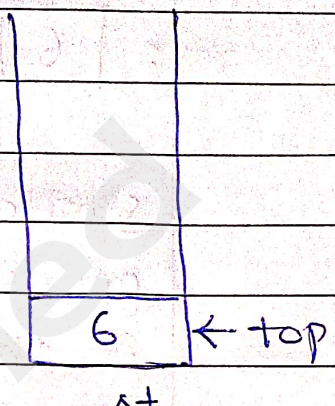
Ex: Let us visulize the above function on Fig(1.1) on page No. (6)

(1) Root is not empty create a stack push root Node on the top of stack.

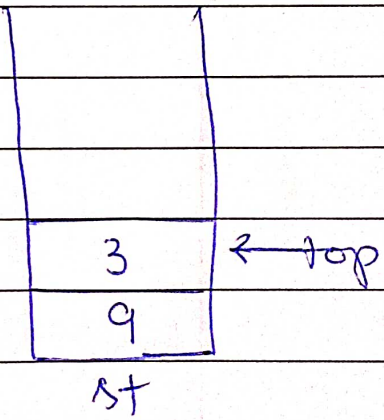
res: 

(2) Now, Enters into while loop as the stack st is not empty.

tempNode = root (st.top()).



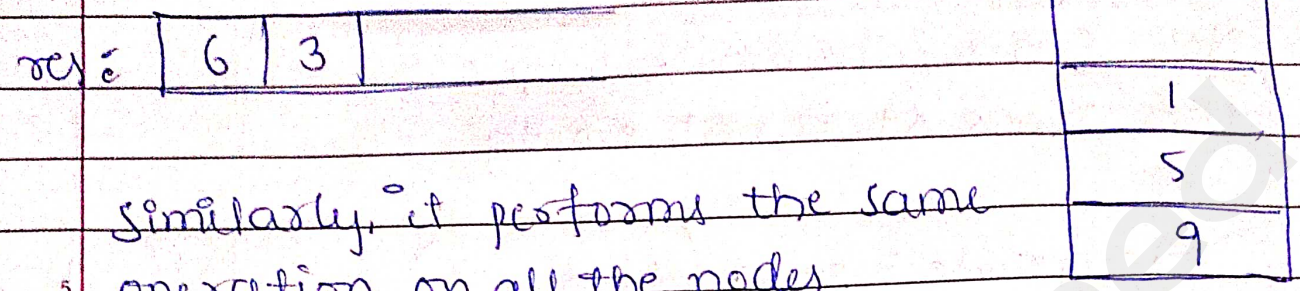
Now, pops the top of stack & checks if it has a right child it has a right child. so, push it on to the stack, similarly it have the left child also so, put that also on the top of stack. and finally, push the data value of tempNode on the res vector. After this iteration on the contents of stack st and res vector are as follows:



res: | 6 |

Now, it removes 3 from stack & inserts the children of 3 ie. first (5) and then (1) and after that pushes the value 3 into the res vector.

After this iteration the contents of stack st and res vector is.....



Similarly, it performs the same operation on all the nodes
At the end when Stack St, becomes empty, the res vector contains the preorder of the tree nodes