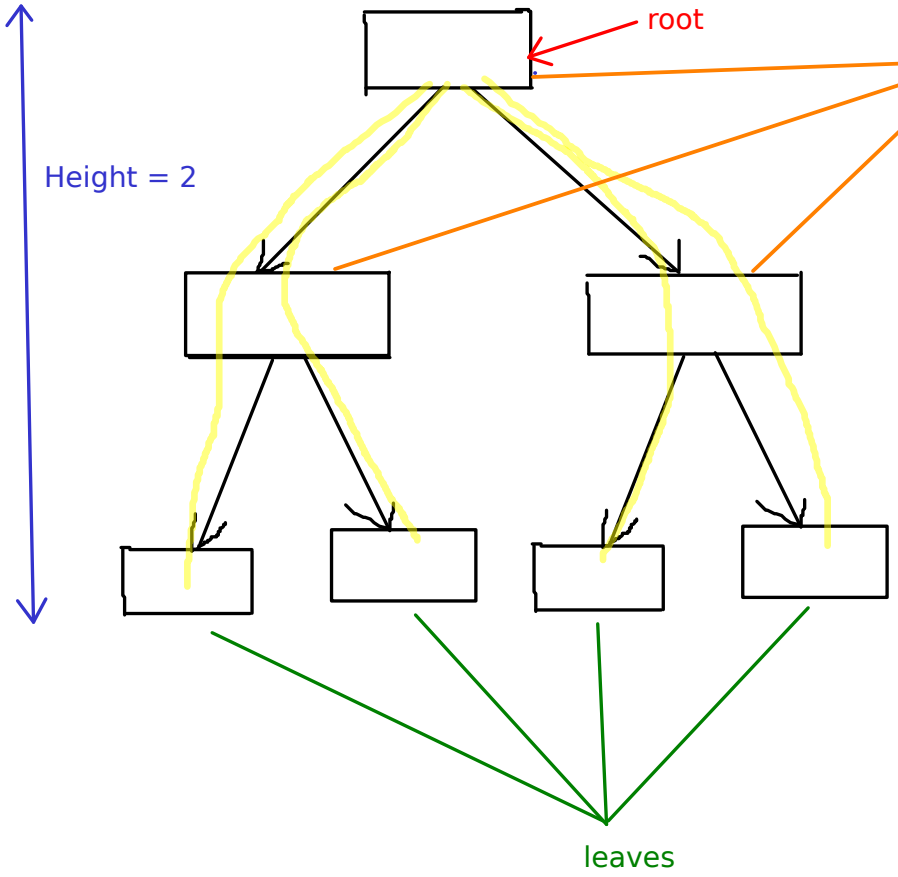
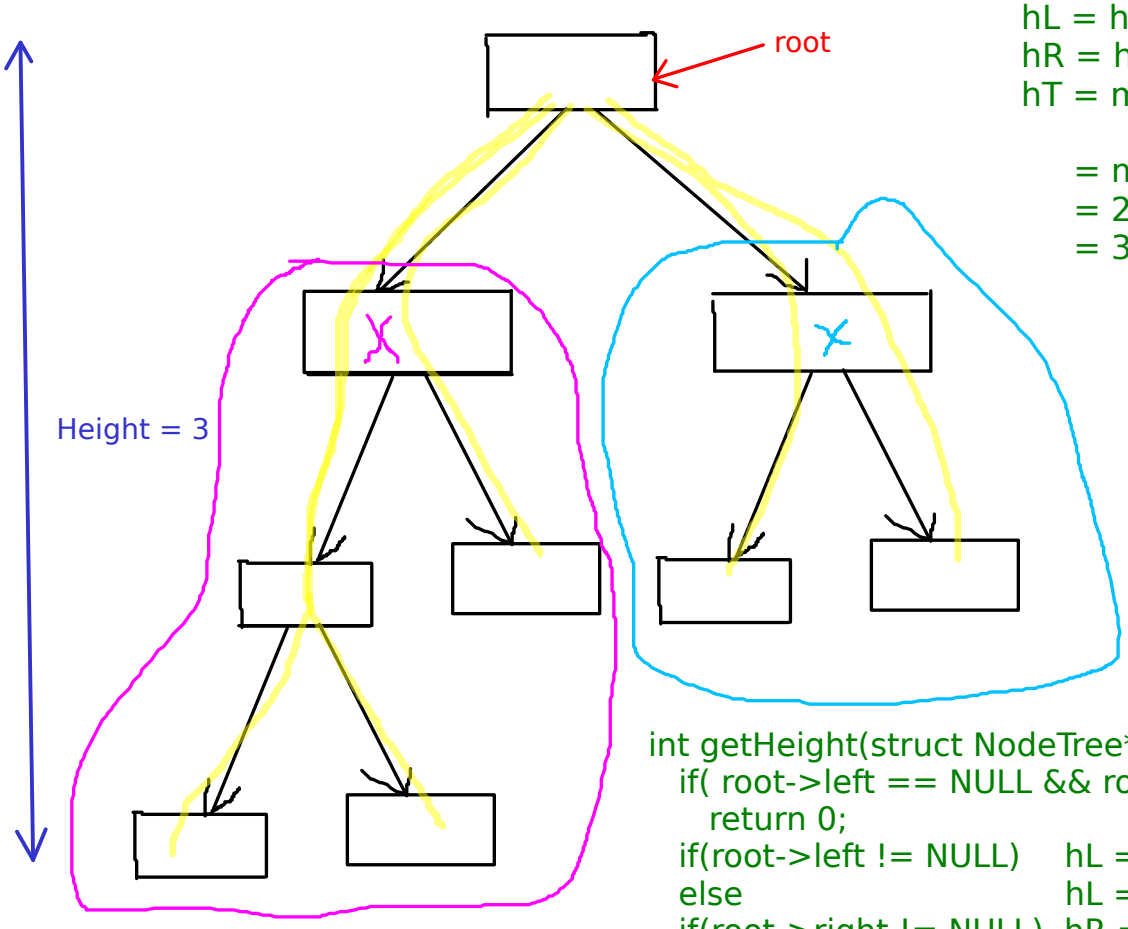


Binary Tree

N-ary Tree

Theorem: For any binary tree, no. of leaf nodes is always 1 more than the no. of internal nodes.

```
struct TreeNode {
      int data;
      struct TreeNode* left;
      struct TreeNode* right;
      // struct TreeNode* parent;
}
```



hL = height of left subtree
hR = height of right subtree
hT = max(hL, hR) + 1

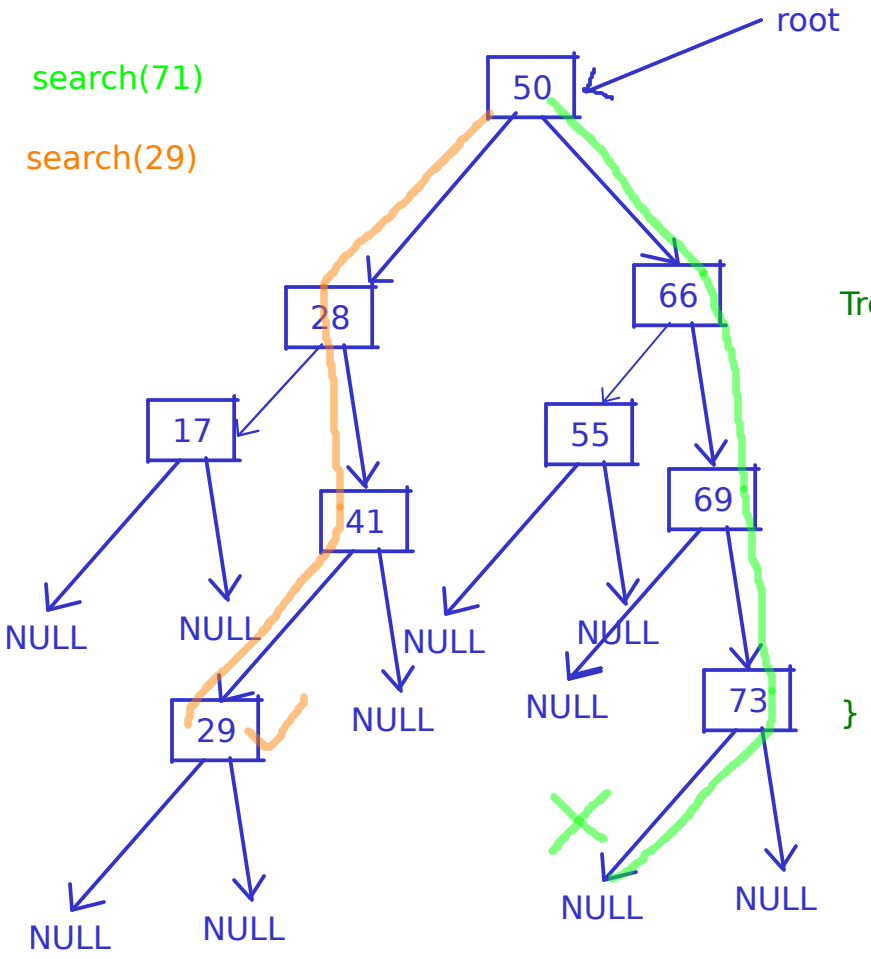
= max(2,1) + 1
= 2 + 1
= 3

h	#leaves	#total
0	1	1
1	2	3
2	4	7
3	8	15

```
int getHeight(struct NodeTree* root) {
   if( root->left == NULL && root->right == NULL)
      return 0;
   if(root->left != NULL)    hL = getHeight( root->left );
   else                      hL = 0;
   if(root->right != NULL)   hR = getHeight( root->right );
   else                      hR = 0;
   if (hL > hR)          return (hL + 1);
   else                  return (hR + 1);
}
```

Theorem: For a binary tree having height h, the maximum possible no. of leaf nodes is 2^h, and the total no. of nodes can be at most 2^(h+1)-1.

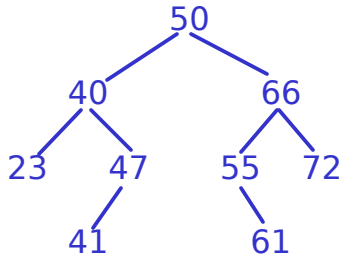
Create a Binary Search Tree (BST) by inserting the following elements (in sequence):
50, 28, 41, 66, 69, 73, 55, 17, 29




```
struct TreeNode {
        int data;
        struct TreeNode* left;
        struct TreeNode* right;
}
```

```
TreeNode* search(TreeNode* root, int k) {
        if( root == NULL ) {
                print("Not found!");
                return NULL;
        }
        if( k == root->data )
                return root;
        if( k < root->data )
                return search( root->left, k );
        if( k > root->data )
                return search( root->right, k );
}
```

```
Treenode* insert(TreeNode* root, int k) {
  if( root == NULL ) {
        struct TreeNode* newNode = ...;
        newNode->data = k;
        newNode->left = NULL;
        newNode->right = NULL;
        return newNode;
  }
  else {
        if( k <= root->data )
                root->left = insert( root->left, k );
        if( k > root->data )
                root->right = insert( root->right, k );
        return root;
  }
}
```



Pre-order: ROOT, LEFT Subtree, RIGHT Subtree

Post-order: LEFT Subtree, RIGHT Subtree, ROOT

In-order: LEFT Subtree, ROOT, RIGHT Subtree

(...) 50 (...)

((...) 40 (...)) 50 (...)

((23) 40 (...)) 50 (...)

((23) 40 ((...) 47)) 50 (...)

((23) 40 ((41) 47)) 50 (...)

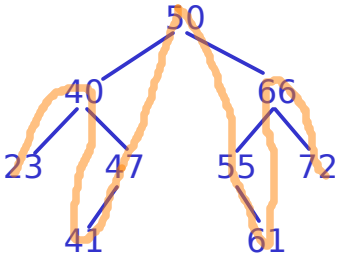
((23) 40 ((41) 47)) 50 ((...) 66 (...))

((23) 40 ((41) 47)) 50 ((55 (...)) 66 (...))

((23) 40 ((41) 47)) 50 ((55 (61)) 66 (...))

((23) 40 ((41) 47)) 50 ((55 (61)) 66 (72))

```
void inorder(struct TreeNode* root) {
        if( root == NULL ) {
                print("Tree is empty!");
                return;
        }
        if( root->left != NULL )
                inorder( root->left );
        print( root->data );
        if( root->right != NULL )
                inorder( root->right );
}
```



(...) (...) 50

((...) (...) 40) (...) 50

((23) (...) 40) (...) 50

((23) ((...) 47) 40) (...) 50

((23) ((41) 47) 40) (...) 50

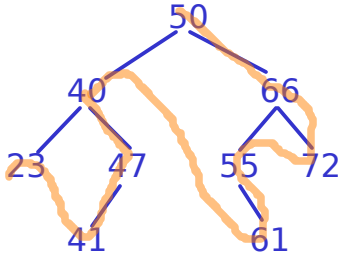
((23) ((41) 47) 40) ((...) (...) 66) 50

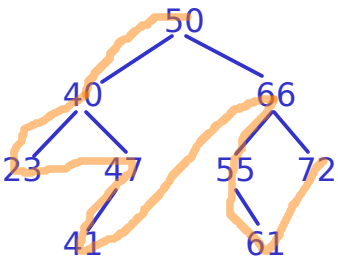
((23) ((41) 47) 40) (((...) 55) (...) 66) 50

((23) ((41) 47) 40) (((61) 55) (...) 66) 50

((23) ((41) 47) 40) (((61) 55) (72) 66) 50

```
void postorder(struct TreeNode* root) {
        if( root == NULL ) {
                print("Tree is empty!");
                return;
        }
        if( root->left != NULL )
                postorder( root->left );
        if( root->right != NULL )
                postorder( root->right );
        print( root->data );
}
```






```
void preorder(struct TreeNode* root) {
        if( root == NULL ) {
                print("Tree is empty!");    return;  }
        print( root->data );
        if( root->left != NULL )          preorder( root->left );
        if( root->right != NULL )         preorder( root->right );
}
```