Tree Data Structures

Tree data structures are versatile and powerful ways to organize and manipulate hierarchical information. In this presentation, we'll explore their many applications and learn how to use them effectively.

Dr. Gopal Chandra Jana
Assistant Professor

Definition and Characteristics of Trees

1. **Node-Based Structure** 🌳

   A tree is a collection of nodes connected by edges, with a single root node and no cycles or loops.

2. **Hierarchical Relationship**

   Trees represent a natural hierarchical structure, making them ideal for organizing data that has parent-child relationships.

3. **Multiple Branches** 🌱

   A tree can have multiple branches, with each node having at most one parent and zero or more children.

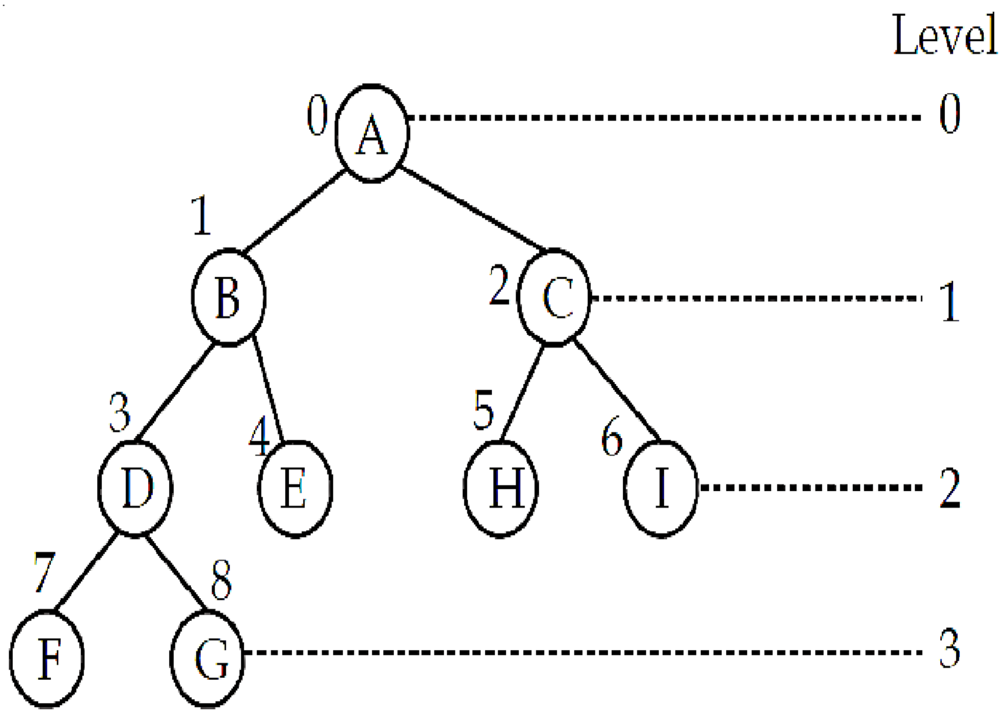
4. **Recursive Nature**

   The recursive structure of trees allows for efficient algorithms with logarithmic time complexity.

Binary Tree

•A tree is a non-linear data structure. It is considered as a powerful and very flexible data structure that can be used for a wide variety of applications.

•A Tree is said to be a Binary Tree if all of its nodes have at-most 2 children. That is, all of its node can have either no child, 1 child, or 2 child nodes.



Root: Root is the first node of the hierarchical data structure. It is present at the 0th level and this node represents the base address of the tree. As node A is the root of the tree.

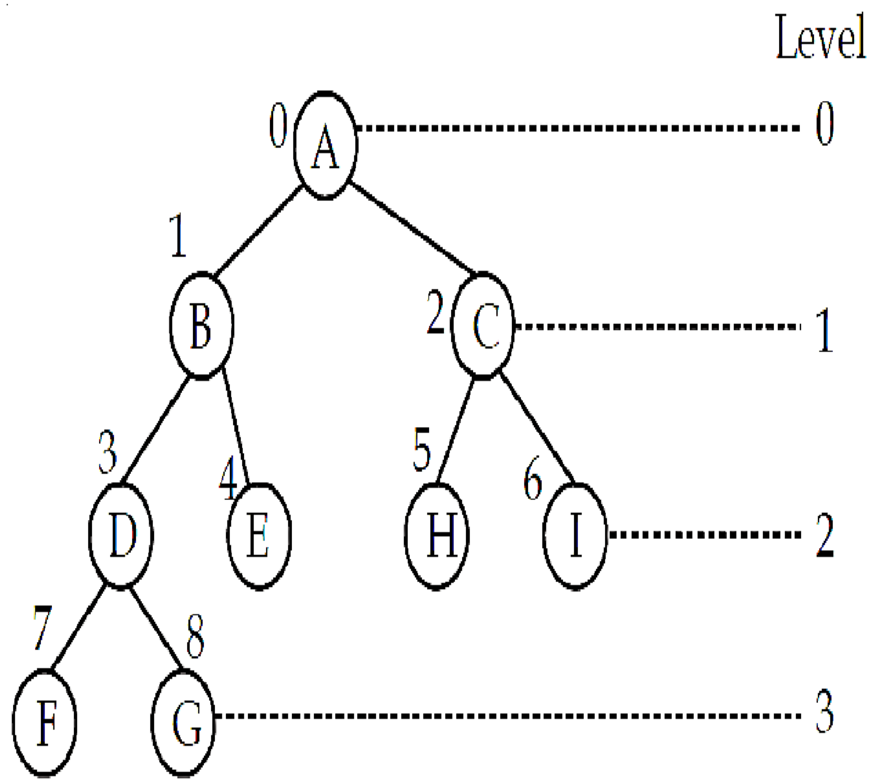
Nodes: Each data item in the tree is called a node like a node in a graph. It specifies the information about the data and the links to other items. A, B, C, D, E, and F are the nodes of the tree.

Degree: The number of nodes connected to a particular node 'A' is called the degree of that node 'A'. The degree of the leaf node is always one. As deg(A)=2, deg(B)= deg(C)= deg(D)=3, deg(E)= deg(H)=1,

The degree of the tree is the maximum value of the degree of any node in the tree. As deg(tree) =3.

Level: The root node of the tree has level 0. The level of any other node is one more than the level of its parent. As Level(A)=0, Level(B)=Level(C)=1 and so on.

Depth: The maximum level of any leaf is called the depth of the binary tree. As the depth of the tree would be 3 here.



Height: Height of the tree is the total number of levels from the root node to the terminal/leaf node. As Height of the tree would be 4 here.

Ancestor: A node "A" is said to be an ancestor of node "B" if A is either the father of B or the father of some ancestor of B.

For example; A is an ancestor of B, E, etc.

Descendent: A node "B" is said to be a left descendant of node "A" if B is either the left son of A or the descendant of the left son of A.

For example; G, F, D, B, E are the left descendent to node A.

A node "B" is said to be a right descendant of node "A" if B is either the right son of A or the descendant of the right son of A.

For example; C, H, I are the right descendent of node A.

Climbing: Is the process of traversing the tree from the leaf node to the root node, also known as Bottom-up traversal.

Descending: Is the process of traversing the tree from the root node to the leaf node, also known as Top-down traversal.

Properties of a Binary Tree

The maximum number of nodes at level 'l' of a binary tree is (2^{l-1}). Level of root is 1. This can be proved by induction. For root, l = 1, number of nodes = 2^{1-1} = 1 Assume that the maximum number of nodes on level l is 2^{l-1}. Since in Binary tree every node has at most 2 children, next level would have twice nodes, i.e. 2 * 2^{l-1}.

Maximum number of nodes in a binary tree of height 'h' is (2^h – 1). Here height of a tree is the maximum number of nodes on the root to leaf path. The height of a tree with a single node is considered as 1. This result can be derived from point 2 above. A tree has maximum nodes if all levels have maximum nodes. So maximum number of nodes in a binary tree of height h is 1 + 2 + 4 + .. + 2^h-1. This is a simple geometric series with h terms and sum of this series is 2^h – 1. In some books, the height of the root is considered as 0. In that convention, the above formula becomes 2^{h+1} – 1.

In a Binary Tree with N nodes, the minimum possible height or the minimum number of levels is $Log_2(N+1)$. This can be directly derived from point 2 above. If we consider the convention where the height of a leaf node is considered 0, then above formula for minimum possible height becomes $Log_2(N+1)$ – 1.

A Binary Tree with L leaves has at least (Log_2L + 1) levels. A Binary tree has maximum number of leaves (and minimum number of levels) when all levels are fully filled. Let all leaves be at level l, then below is true for number of leaves L. L <= 2l-1 [From Point 1] l = Log_2L + 1 where l is the minimum number of levels.

In a Binary tree in which every node has 0 or 2 children, the number of leaf nodes is always one more than the nodes with two children. L = T + 1 Where L = Number of leaf nodes T = Number of internal nodes with two children

Type of a Binary Tree

Full Binary Tree: A Binary Tree is full if every node has either 0 or 2 children. The following are examples of a full binary tree. We can also say that a full binary tree is a binary tree in which all nodes except leave nodes have two children. ***In a Full Binary, the number of leaf nodes is number of internal nodes plus 1*.**

```
        18
     /       \
   15         30
  /  \        /  \
40    50    100    40
```

```
       18
    /     \
  40       30
          /  \
       100    40
```

```
          18
        /    \
      15      20
     /  \
   40    50
  /   \
30    50
```

```
          18
      /        \
    15          30
   /  \        /  \
 40    50    100    40
```

```
          18
      /        \
    15          30
   /  \        /  \
 40    50    100    40
 /  \   /
8    7  9
```

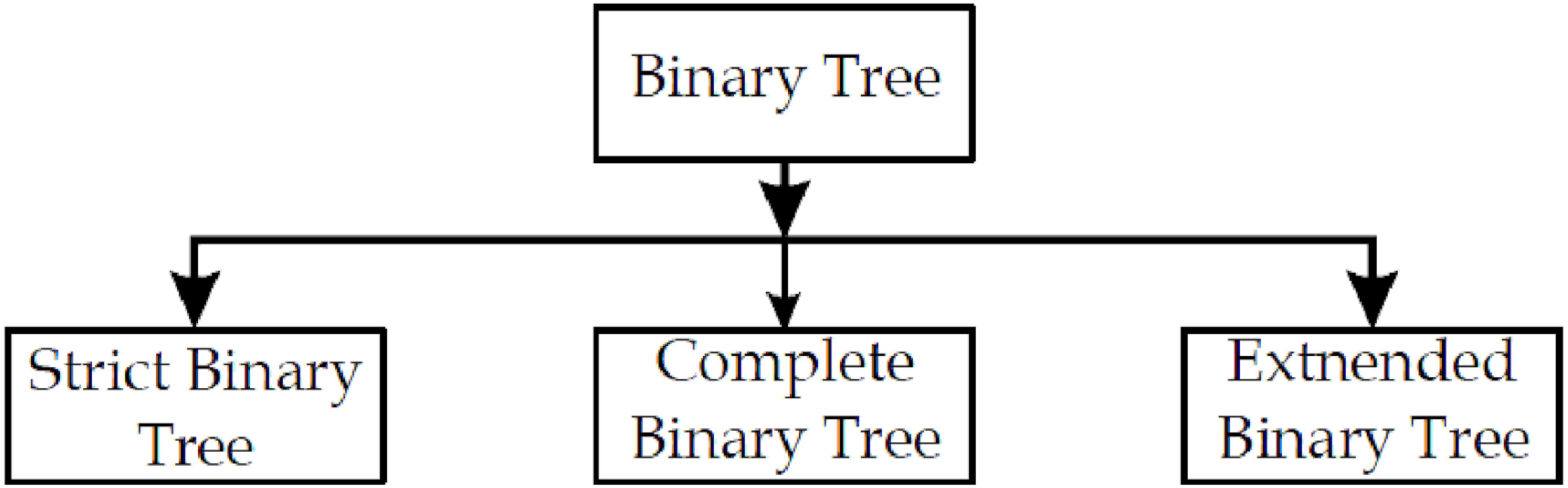
Complete Binary Tree: A Binary Tree is a complete Binary Tree if all levels are completely filled except possibly the last level and the last level has all keys as left as possible Following are the examples of Complete Binary Trees:

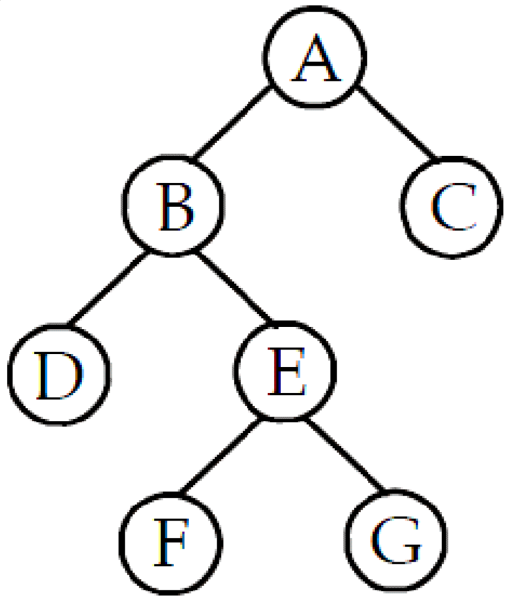
Perfect Binary Tree: A Binary tree is a Perfect Binary Tree when all internal nodes have two children and all the leave nodes are at the same level. Following are the examples of Perfect Binary Trees: A Perfect Binary Tree of height h (where height is the number of nodes on the path from the root to leaf) has $2^h – 1$ node.

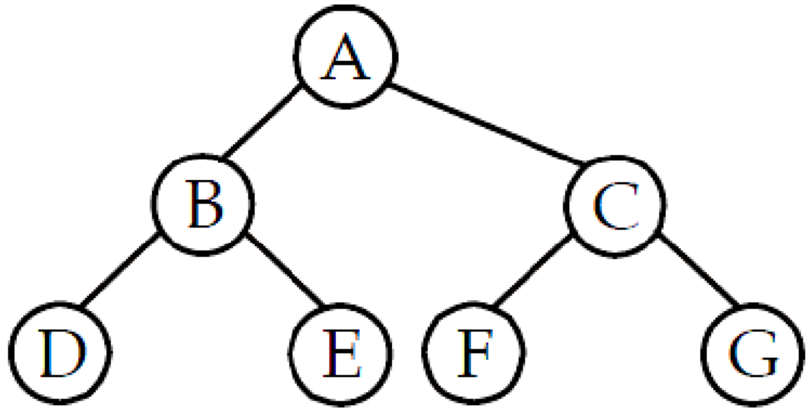
```
          18
      /        \
    15          30
   /  \        /  \
 40    50    100    40
```

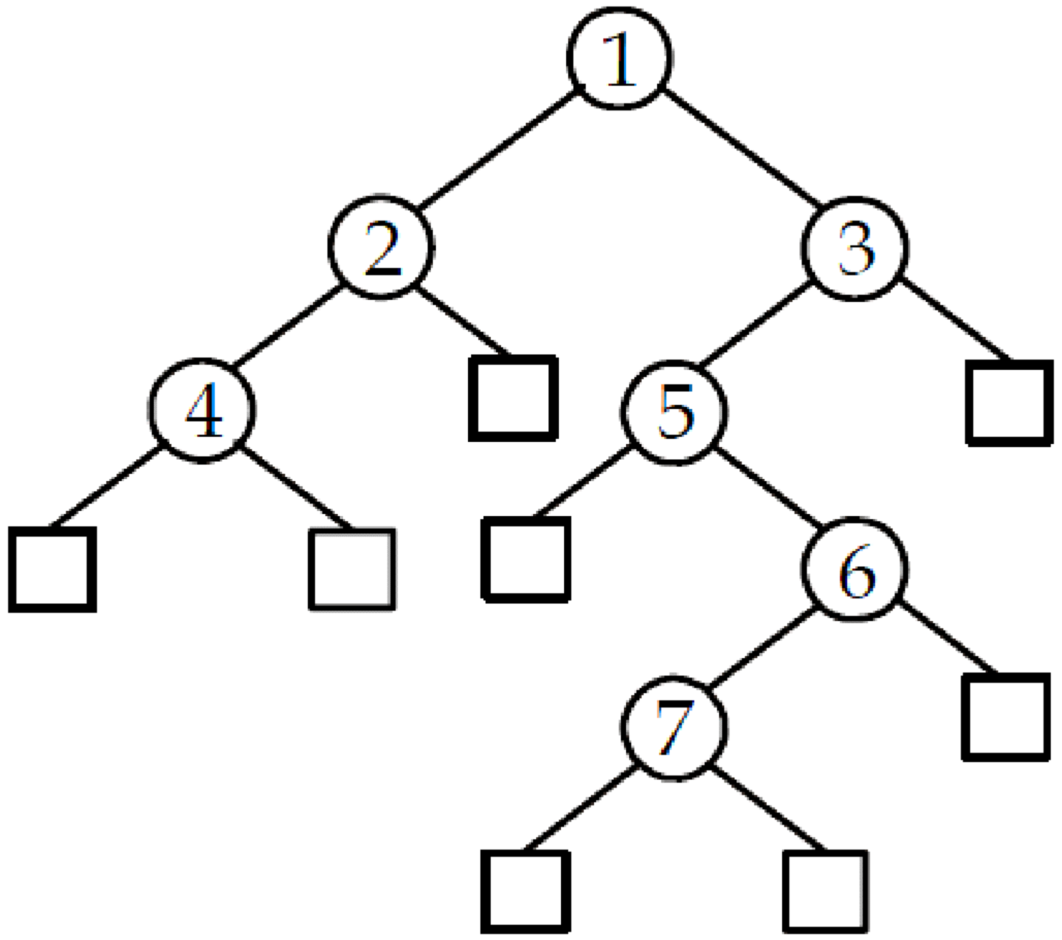
```
       18
   /        \
15           30
```

TYPES OF BINARY TREES

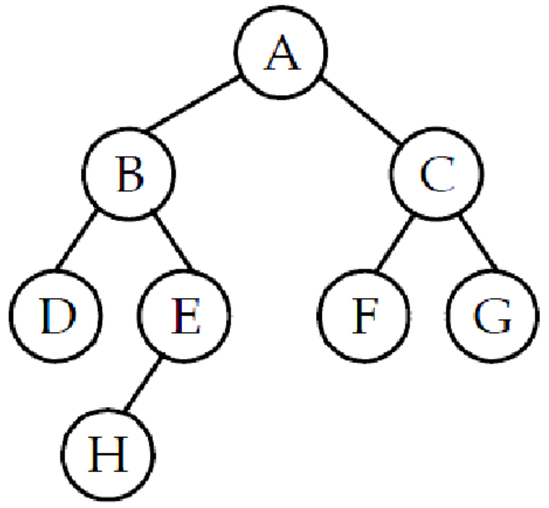



A
B
C
D
E
F
G

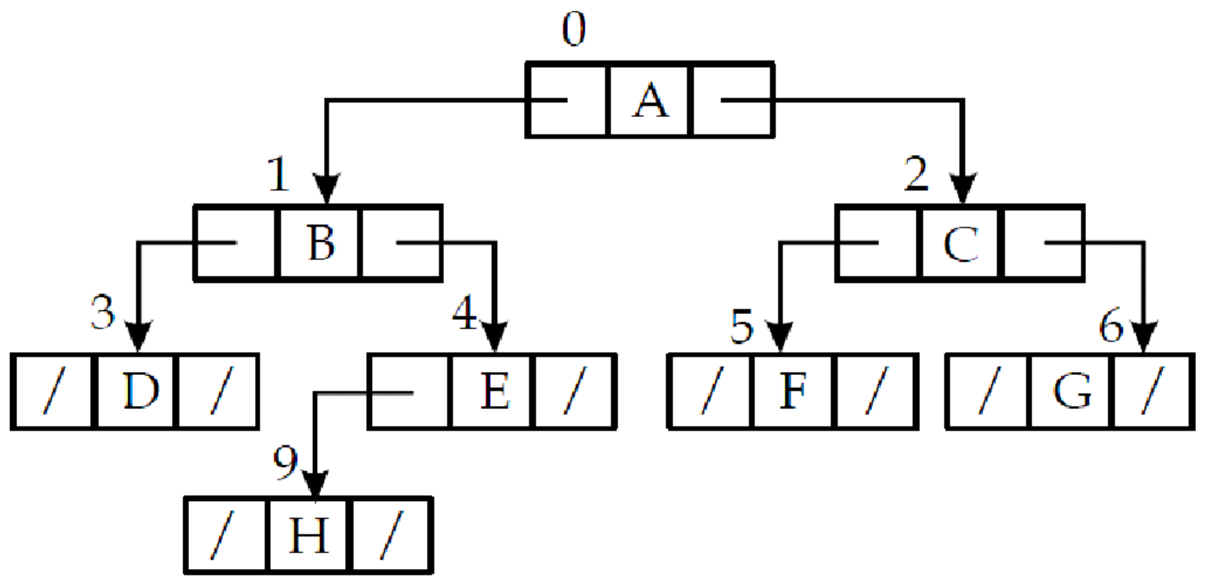

A
B
C
D
E
F
G


1
2
3
4
5
6
7

REPRESENTATION OF BINARY TREE IN MEMORY



(a) Binary tree



(b) Linked list representation of tree

	0	1	2	3	4	5	6	7	8	9
TREE	A	B	C	D	E	F	G	'/0'	'/0'	H
LC	1	3	5	-1	9	-1	-1	-1	-1	-1
RC	2	4	6	-1	-1	-1	-1	-1	-1	-1

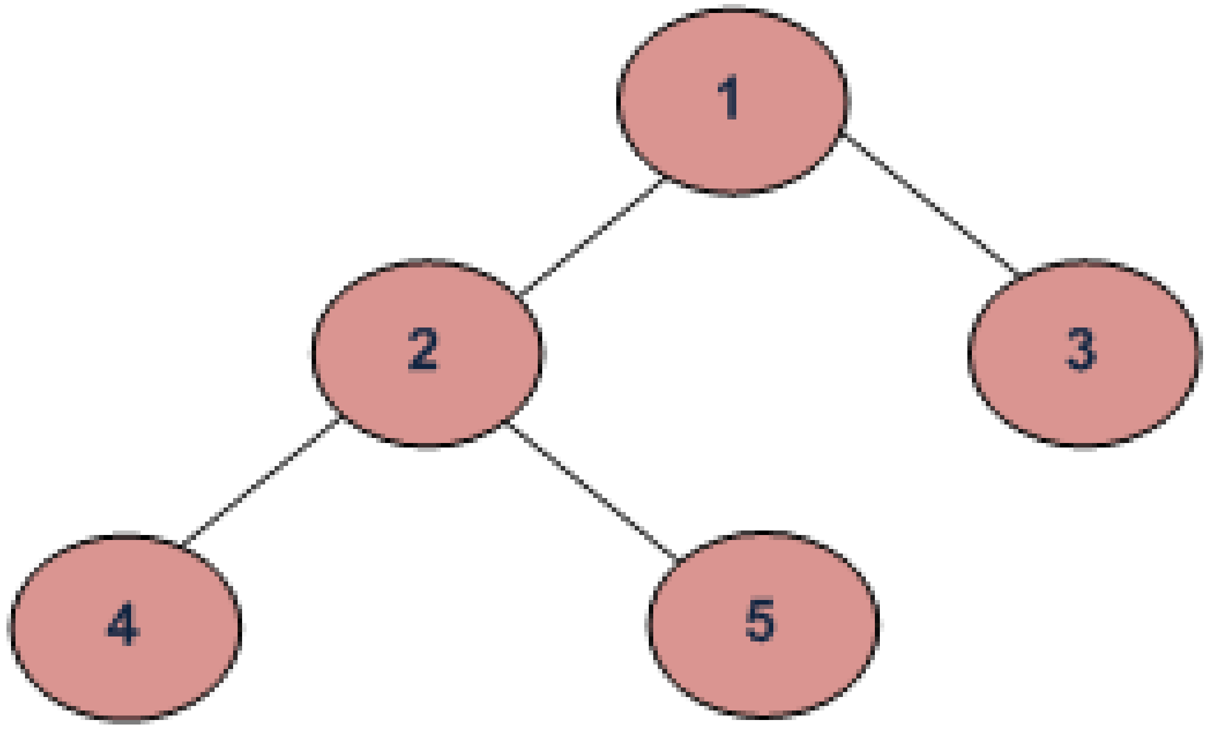
TRAVERSING BINARY TREE

Pre-order: N-L-R (node(root)-left-right)

In-order: L-N-R (left-node(root)-right)

Post-order: L-R-N (left-right-node(root))

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc.), which have only one logical way to traverse them, trees can be traversed in different ways. Following are the generally used ways for traversing trees:



Inorder (Left, Root, Right) : 4 2 5 1 3
Preorder (Root, Left, Right) : 1 2 4 5 3.
Postorder (Left, Right, Root) : 4 5 2 3 1

```cpp
#include <iostream>
using namespace std;

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct Node
{
    int data;
    struct Node* left, *right;
    Node(int data)
    {
        this->data = data;
        left = right = NULL;
    }
};
```

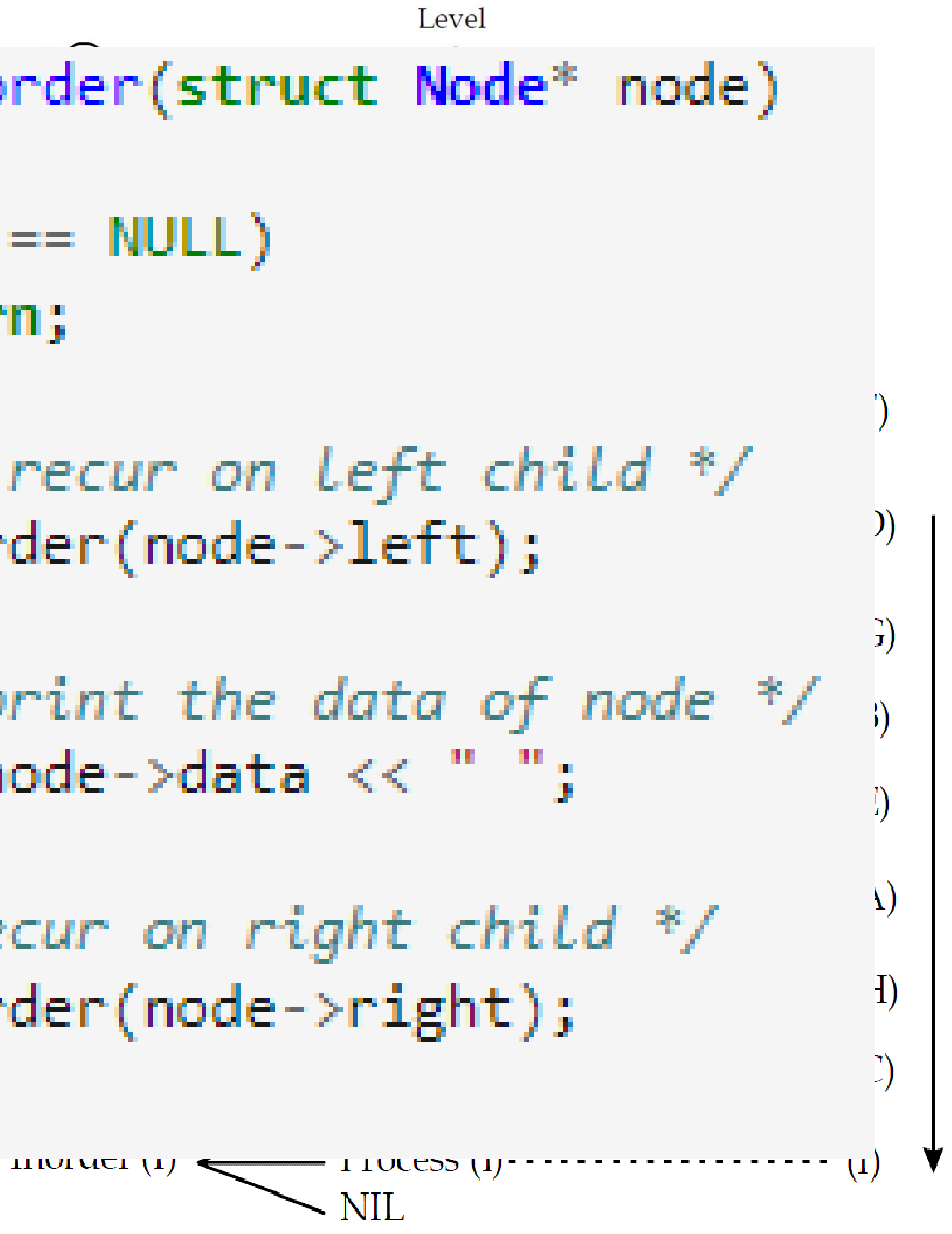
```cpp
// Driver Code
int main()
{
    // Contrust the Tree
    //       1
    //      /  \
    //     2    3
    //    / \
    //   4   5

    struct Node *root = new Node(1);
    root->left     = new Node(2);
    root->right     = new Node(3);
    root->left->left = new Node(4);
    root->left->right = new Node(5);

    cout << "Preorder traversal of binary tree is \n";
    printPreorder(root);

    cout << "\nInorder traversal of binary tree is \n";
    printInorder(root);

    cout << "\nPostorder traversal of binary tree is \n";
    printPostorder(root);

    return 0;
}
```

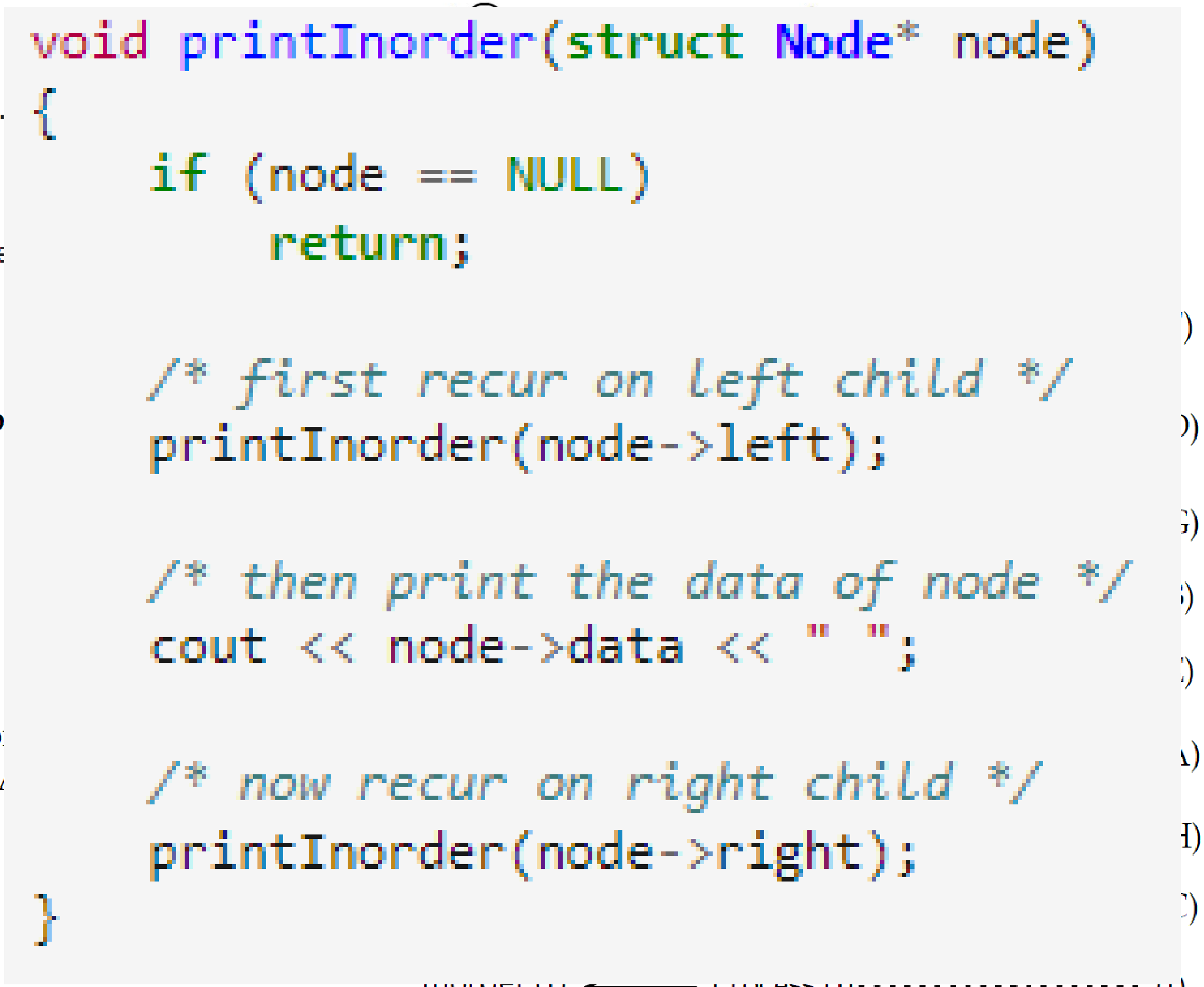
```
Algorithm Inorder(tree)
   1. Traverse the left subtree, i.e.
      call Inorder(left->subtree)
   2. Visit the root.
   3. Traverse the right subtree, i.e
      call Inorder(right->subtree)
```

Ino

Ino

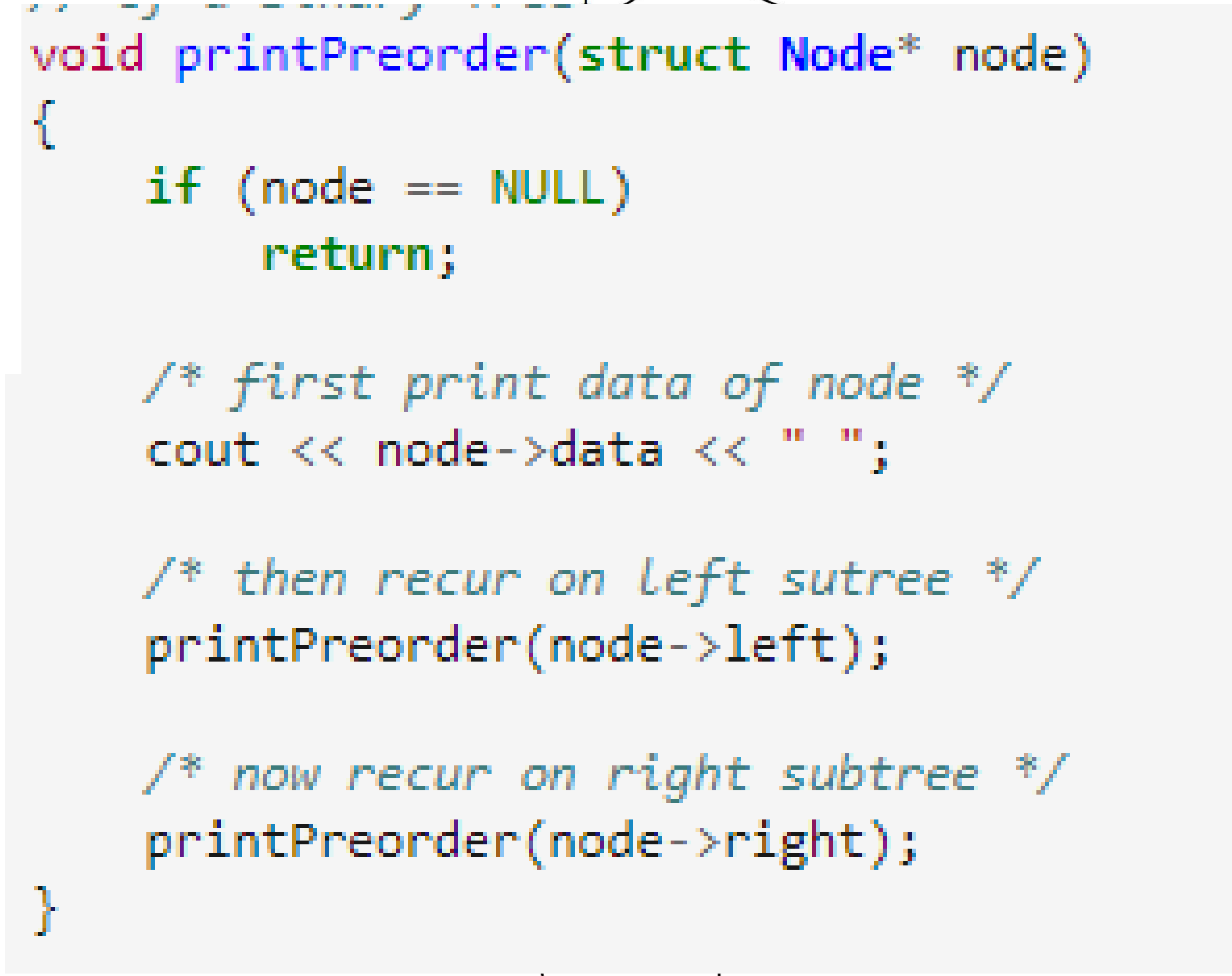
(A



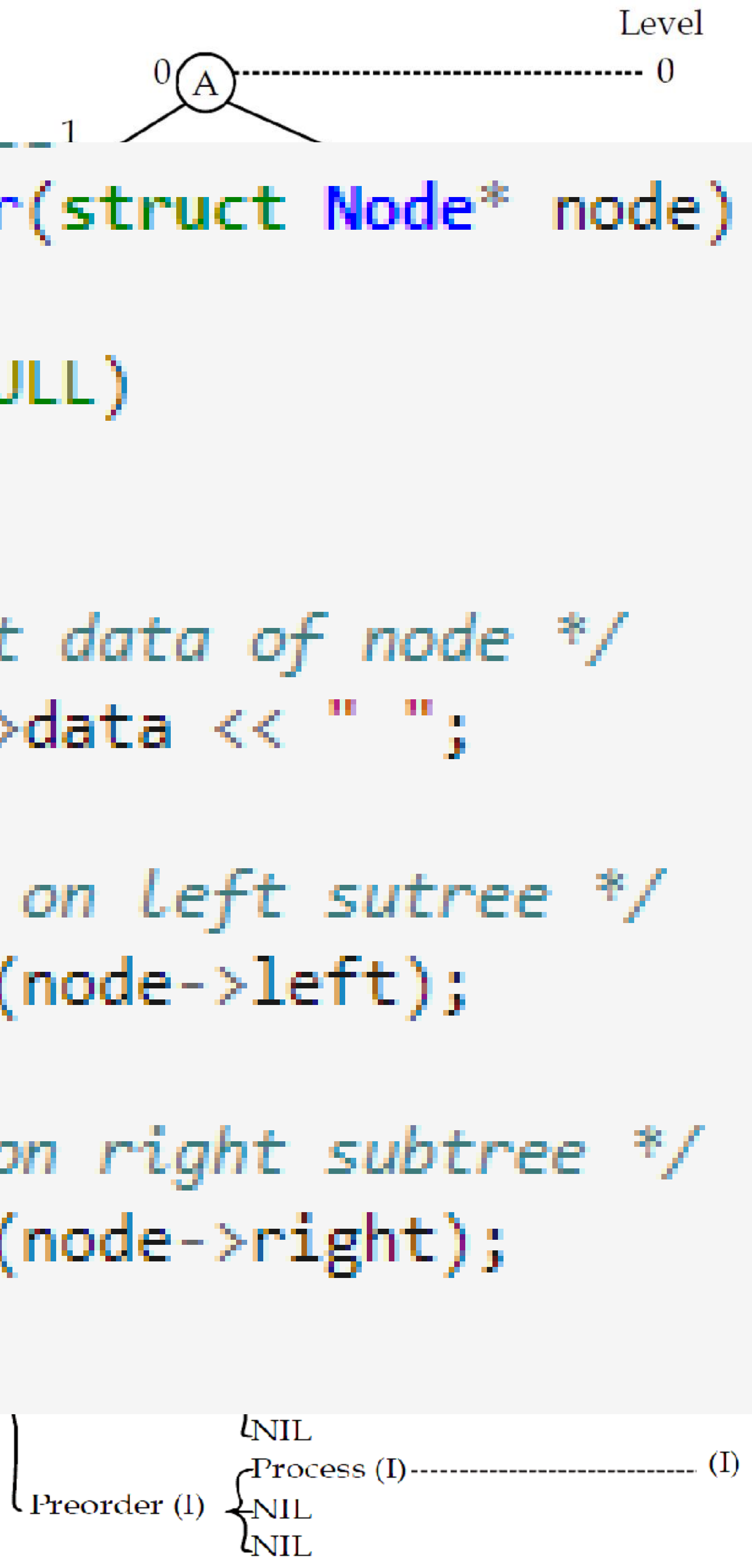


```
void printInorder(struct Node* node)
{
    if (node == NULL)
        return;

    /* first recur on left child */
    printInorder(node->left);

    /* then print the data of node */
    cout << node->data << " ";

    /* now recur on right child */
    printInorder(node->right);
}
```

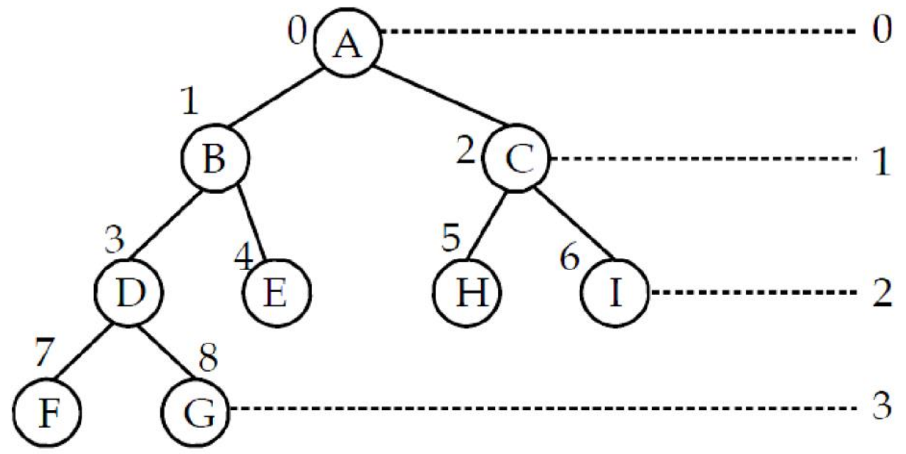
```
Algorithm Preorder(tree)
   1. Visit the root.
   2. Traverse the left subtree, i.e.,
      call Preorder(left-subtree)
   3. Traverse the right subtree, i.e.,
      call Preorder(right-subtree)
```



```
void printPreorder(struct Node* node)
{
    if (node == NULL)
        return;

    /* first print data of node */
    cout << node->data << " ";

    /* then recur on left sutree */
    printPreorder(node->left);

    /* now recur on right subtree */
    printPreorder(node->right);
}
```

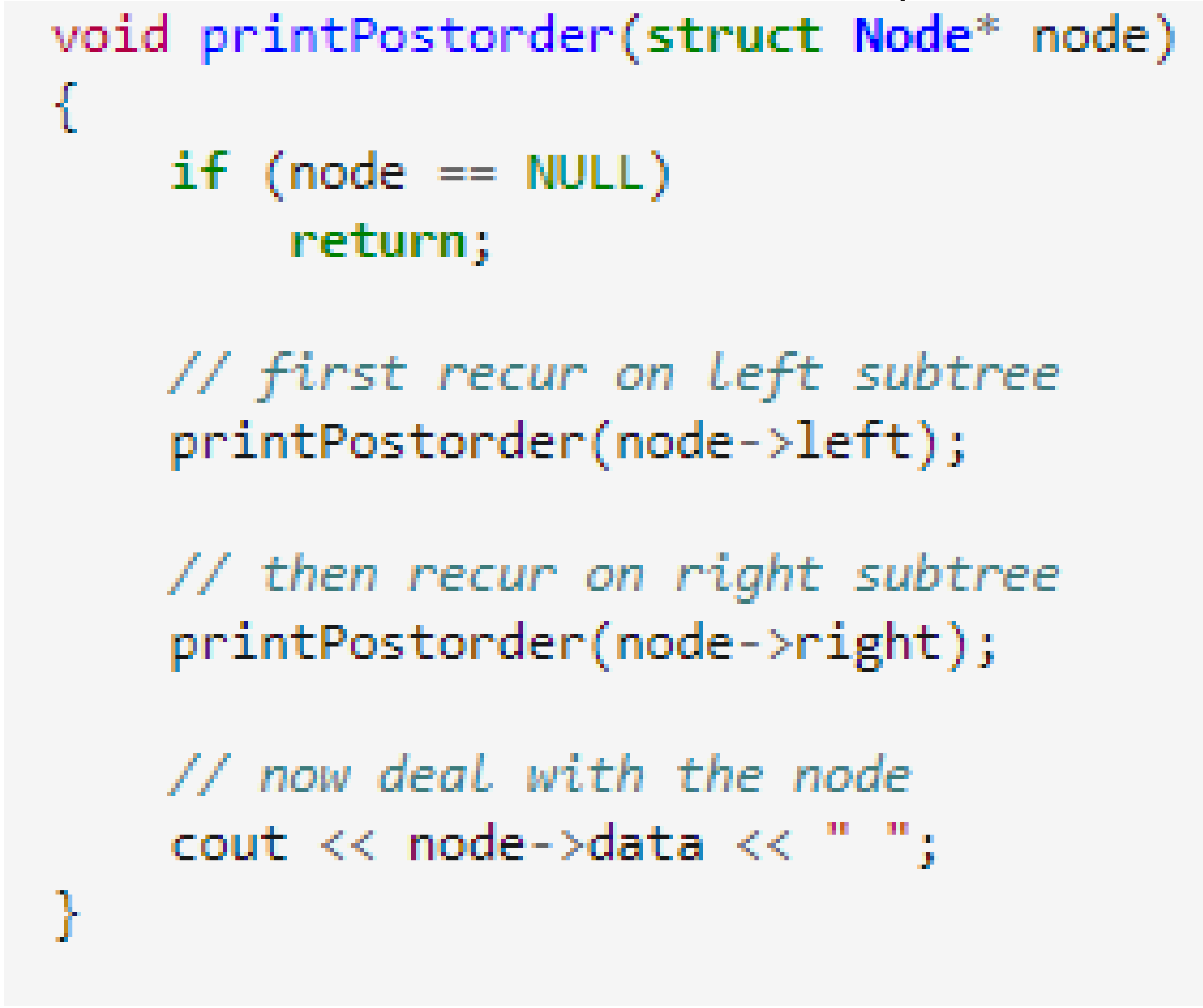


```
Algorithm Postorder(tree)
    1. Traverse the left subtree, i.e.,
       call Postorder(left-subtree)
    2. Traverse the right subtree, i.e.,
       call Postorder(right-subtree)
    3. Visit the root.
```



Post order Traversal:

Nill



```
void printPostorder(struct Node* node)
{
    if (node == NULL)
        return;

    // first recur on left subtree
    printPostorder(node->left);

    // then recur on right subtree
    printPostorder(node->right);

    // now deal with the node
    cout << node->data << " ";
}
```

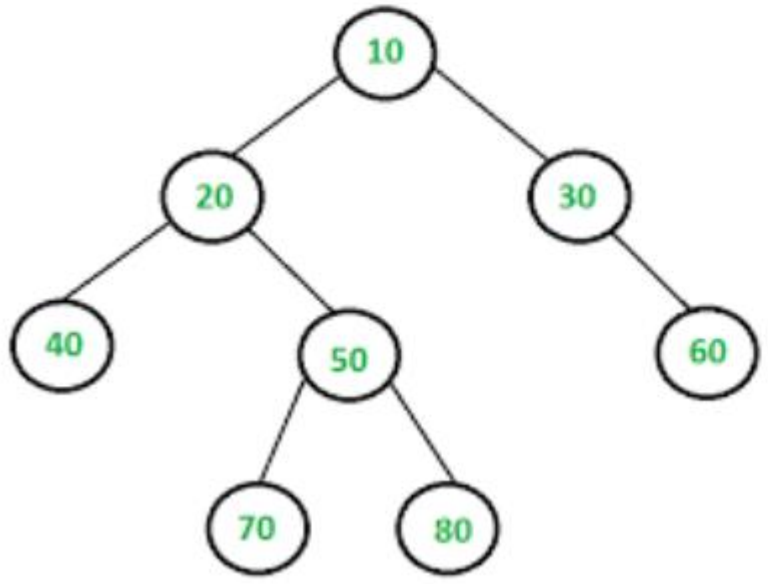
Level Order Traversal of a Binary Tree

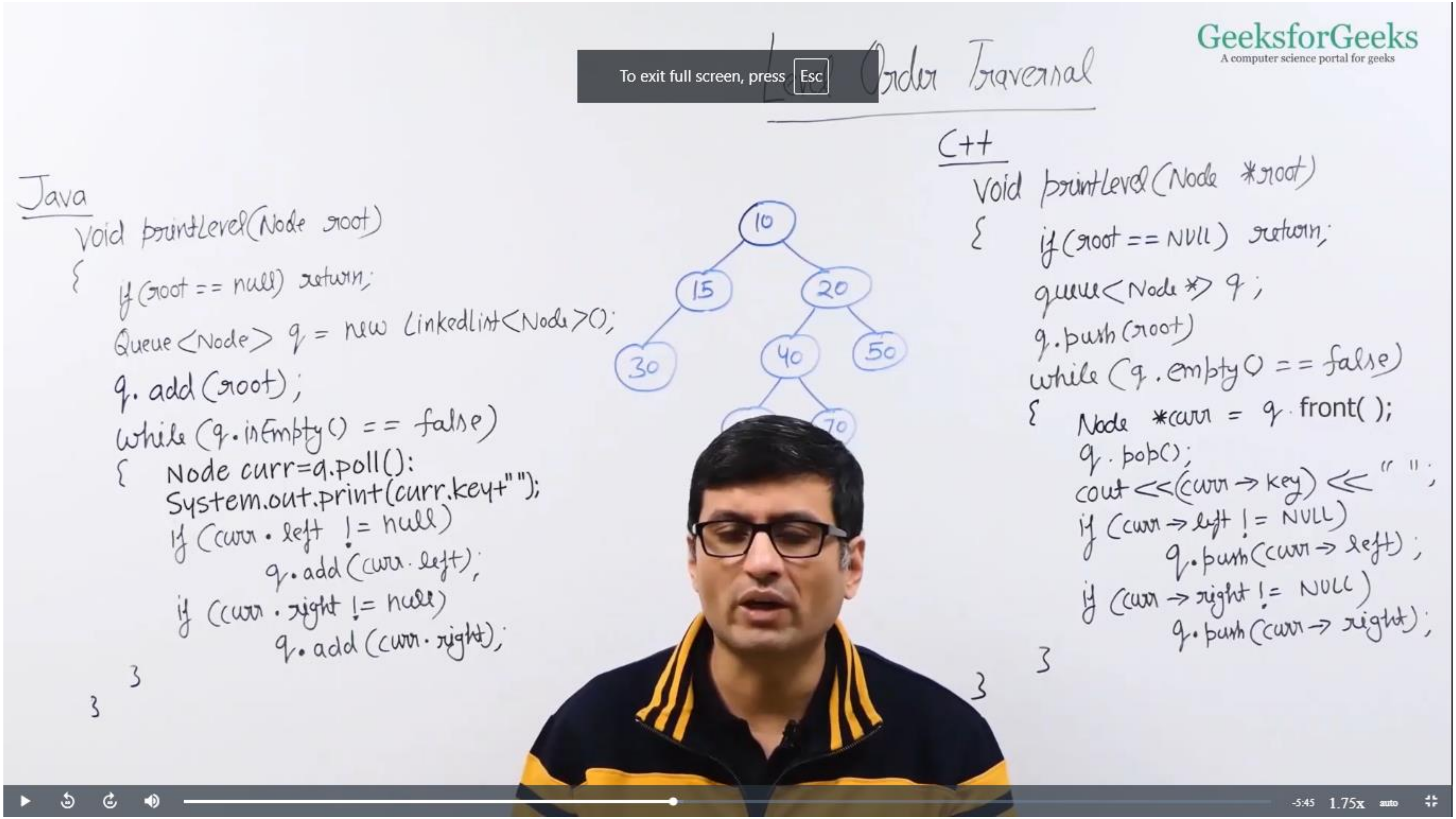
We have seen the three basic traversals(Preorder, postorder, and Inorder) of a Binary Tree. We can also traverse a Binary Tree using the *Level Order Traversal*.

In the Level Order Traversal, the binary tree is traversed level-wise starting from the first to last level sequentially.

Consider the below binary tree:
The Level Order Traversal of the above Binary Tree will be: **10 20 30 40 50 60 70 80**.
Algorithm: The Level Order Traversal can be implemented efficiently using a Queue.

- Create an empty queue q.
- Push the root node of tree to q. That is, q.push(root).
- Loop while the queue is not empty:
  - Pop the top node from queue and print the node.
  - Enqueue node's children (first left then right children) to q
  - Repeat the process until queue is not empty.





GeeksforGeeks
A computer science portal for geeks

Order Traversal

Java

```
void printLevel(Node root)
{
    if(root == null) return;
    Queue<Node> q = new LinkedList<Node>();
    q.add(root);
    while(q.isEmpty() == false)
    {   Node curr=q.poll():
        System.out.print(curr.key+" ");
        if (curr.left != null)
                q.add(curr.left);
        if (curr.right != null)
                q.add(curr.right);
    }
}
```

C++

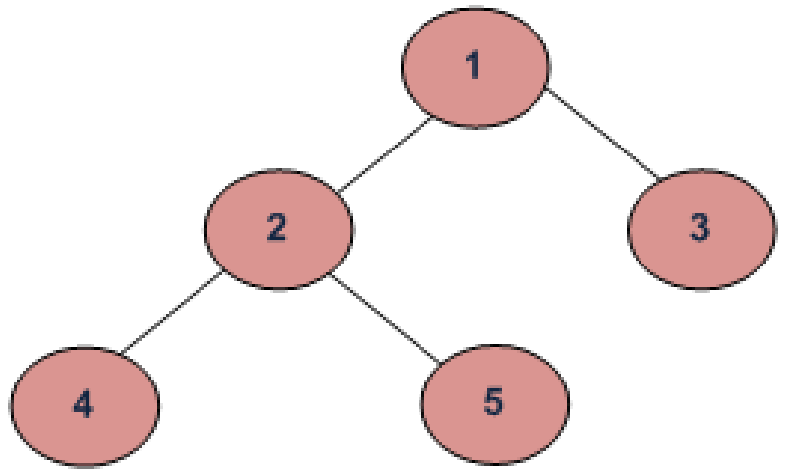
```
void printLevel(Node *root)
{   if(root == NULL) return;
    queue<Node *> q;
    q.push(root)
    while(q.empty() == false)
    {   Node *curr = q.front();
        q.pop();
        cout<<(curr->key)<<" ";
        if (curr->left != NULL)
                q.push(curr->left);
        if (curr->right != NULL)
                q.push(curr->right);
    }
}
```

Height of Binary Tree

Given a binary tree, the task is to find the height of the tree. Height of the tree is the number of edges in the tree from the root to the deepest node, Height of the empty tree is **0**.

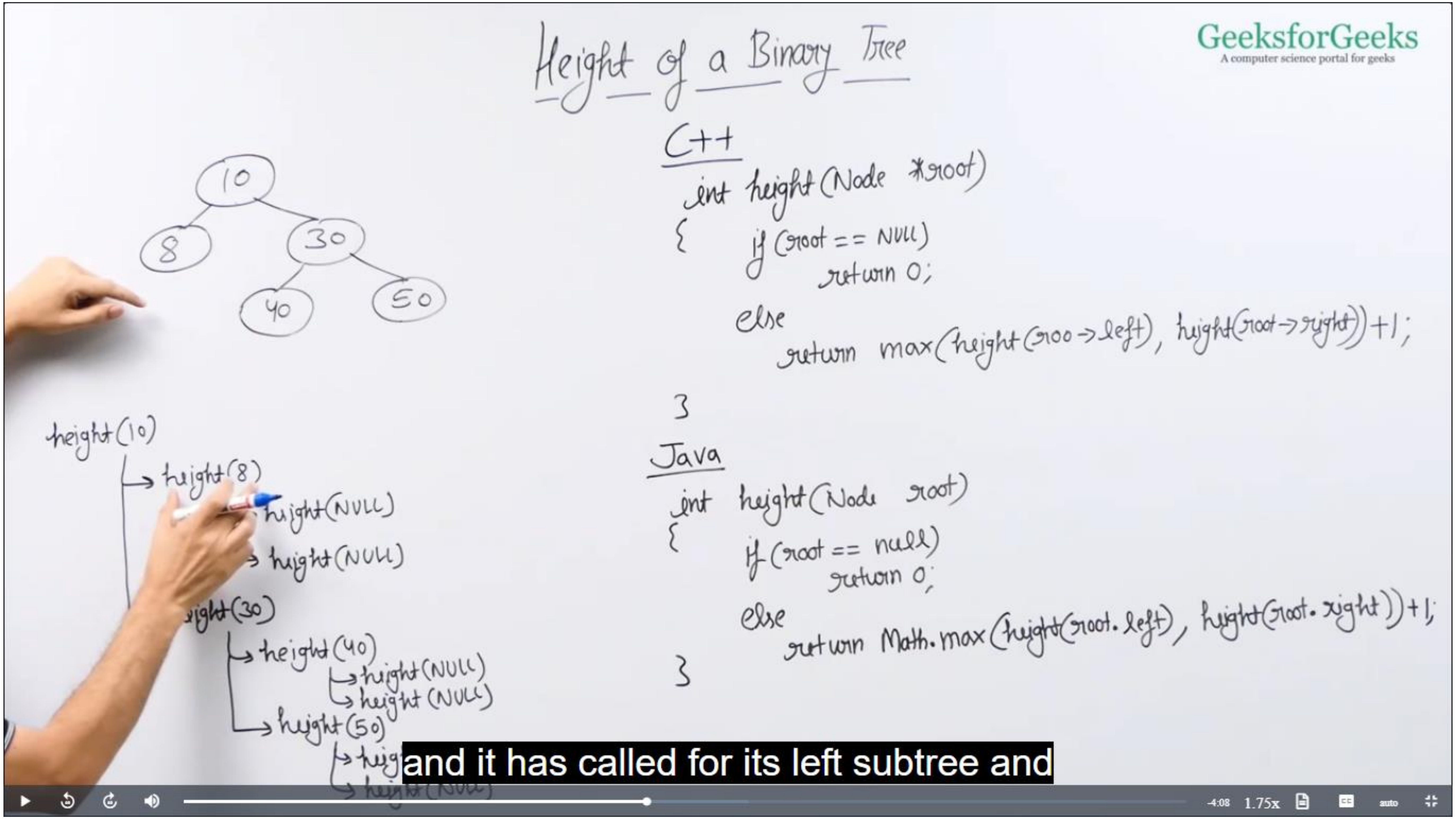
Recursively calculate height of **left** and **right** subtrees of a node and assign height to the node as **max of the heights of two children plus 1**. See below pseudo code and program for details.

maxDepth('1') = max(maxDepth('2'), maxDepth('3')) + 1 = 2 + 1
because recursively
maxDepth('2') = max (maxDepth('4'), maxDepth('5')) + 1 = 1 + 1 and (as height of both '4' and '5' are 1)
maxDepth('3') = 1



Follow the below steps to Implement the idea:

- Recursively do a Depth-first search.
- If the tree is empty then return -1
- Otherwise, do the following
    - Get the max depth of the left subtree recursively i.e. call maxDepth(tree->left-subtree)
    - Get the max depth of the right subtree recursively i.e. call maxDepth(tree->right-subtree)
    - Get the max of max depths of **left** and **right** subtrees and **add 1** to it for the current node.
- Return max_depth.



Height of a Binary Tree

C++

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

Java

```
int height(Node root)
{
        if (root == null)
                return 0;
        else
                return Math.max(height(root.left), height(root.right))+1;
}
```

```
height(10)
    |-> height(8)
    |       -> height(NULL)
    |       -> height(NULL)
    |-> ight(30)
            |-> height(40)
            |       |-> height(NULL)
            |       |-> height(NULL)
            |-> height(50)
                    |-> heig
                    |-> height(NULL)
```

and it has called for its left subtree and

```cpp
// C++ program to find height of tree
#include <bits/stdc++.h>
using namespace std;

/* A binary tree node has data, pointer to left child
and a pointer to right child */
class node {
public:
    int data;
    node* left;
    node* right;
};

/* Compute the "maxDepth" of a tree -- the number of
    nodes along the longest path from the root node
    down to the farthest leaf node.*/
int maxDepth(node* node)
{
    if (node == NULL)
        return 0;
    else {
        /* compute the depth of each subtree */
        int lDepth = maxDepth(node->left);
        int rDepth = maxDepth(node->right);

        /* use the larger one */
        if (lDepth > rDepth)
            return (lDepth + 1);
        else
            return (rDepth + 1);
    }
}
```

```cpp
/* Helper function that allocates a new node with the
given data and NULL left and right pointers. */
node* newNode(int data)
{
    node* Node = new node();
    Node->data = data;
    Node->left = NULL;
    Node->right = NULL;

    return (Node);
}

// Driver code
int main()
{
    node* root = newNode(1);

    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    cout << "Height of tree is " << maxDepth(root);
    return 0;
}
```

Output

```
Height of tree is 3
```

Time Complexity: O(N)

Auxiliary Space: O(N) due to recursive stack.

Operations on Trees

1

Traversing and Searching 🔍

There are multiple ways to traverse a tree, such as Inorder, Preorder, Postorder and Level-Order. Searching can be done recursively or iteratively, depending on the tree's structure.

2

Insertion and Deletion ➕➖

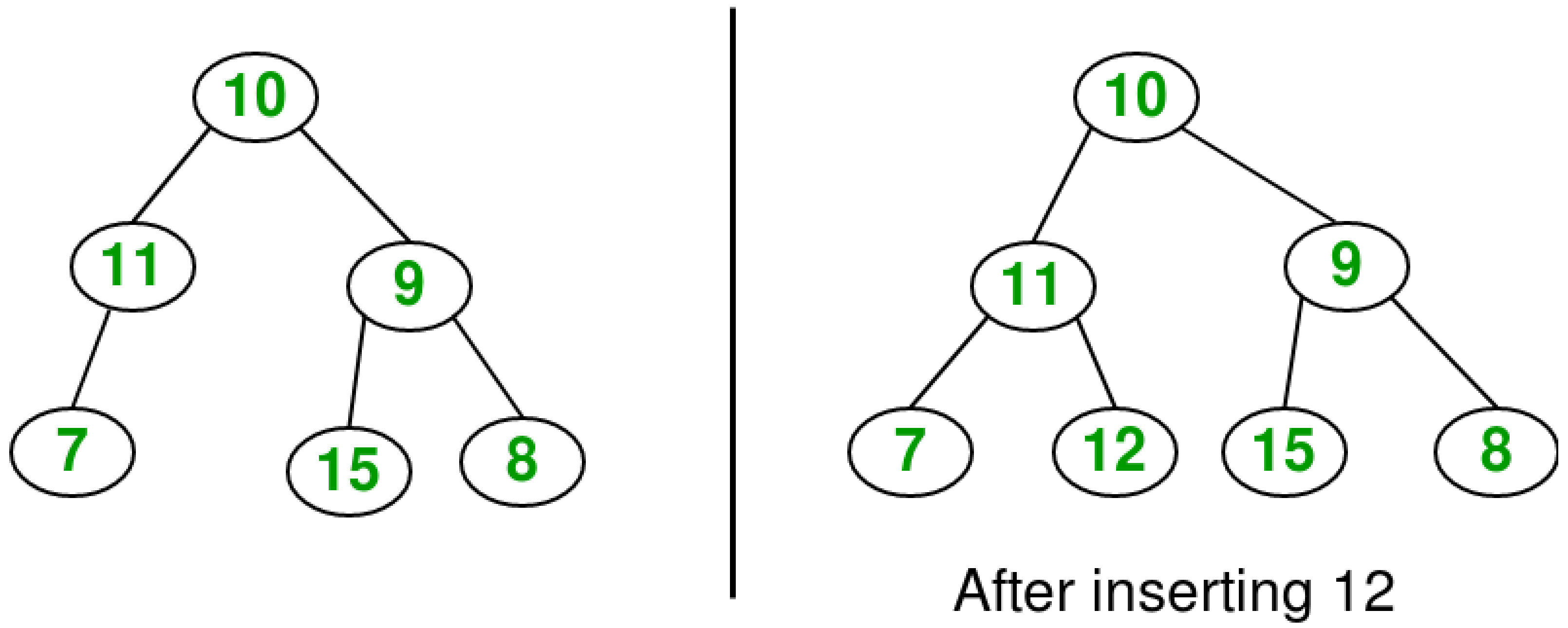
New nodes can be inserted into a tree in an ordered manner, while existing nodes can be deleted using their position or value. These actions are crucial for sorting and manipulating hierarchical data.

Insertion in a Binary Tree

Problem: Given a **Binary Tree** and a **Key**. The task is to insert the *key* into the binary tree at first position available in level order.

The idea is to do iterative level order traversal of the given tree using a queue. If we find a node whose left child is empty, we make new key as the left child of the node. Else if we find a node whose right child is empty, we make new key as the right child of that node. We keep traversing the tree until we find a node whose either left or right child is empty.

```
// Function to insert a new element in a Binary Tree
void insert(struct Node* temp, int key)
{
    queue<struct Node*> q;
    q.push(temp);

    // Do level order traversal until we find
    // an empty place.
    while (!q.empty()) {
        struct Node* temp = q.front();
        q.pop();

        if (!temp->left) {
            temp->left = newNode(key);
            break;
        } else
            q.push(temp->left);

        if (!temp->right) {
            temp->right = newNode(key);
            break;
        } else
            q.push(temp->right);
    }
}
```



Deletion in a Binary Tree

Problem: Given a Binary Tree and a node to be deleted from this tree. The task is to delete the given node from it.

Examples:

Delete 10 in below tree

```
   10
 /    \
20     30
```

Output :

```
    30
  /
20
```

Delete 20 in below tree

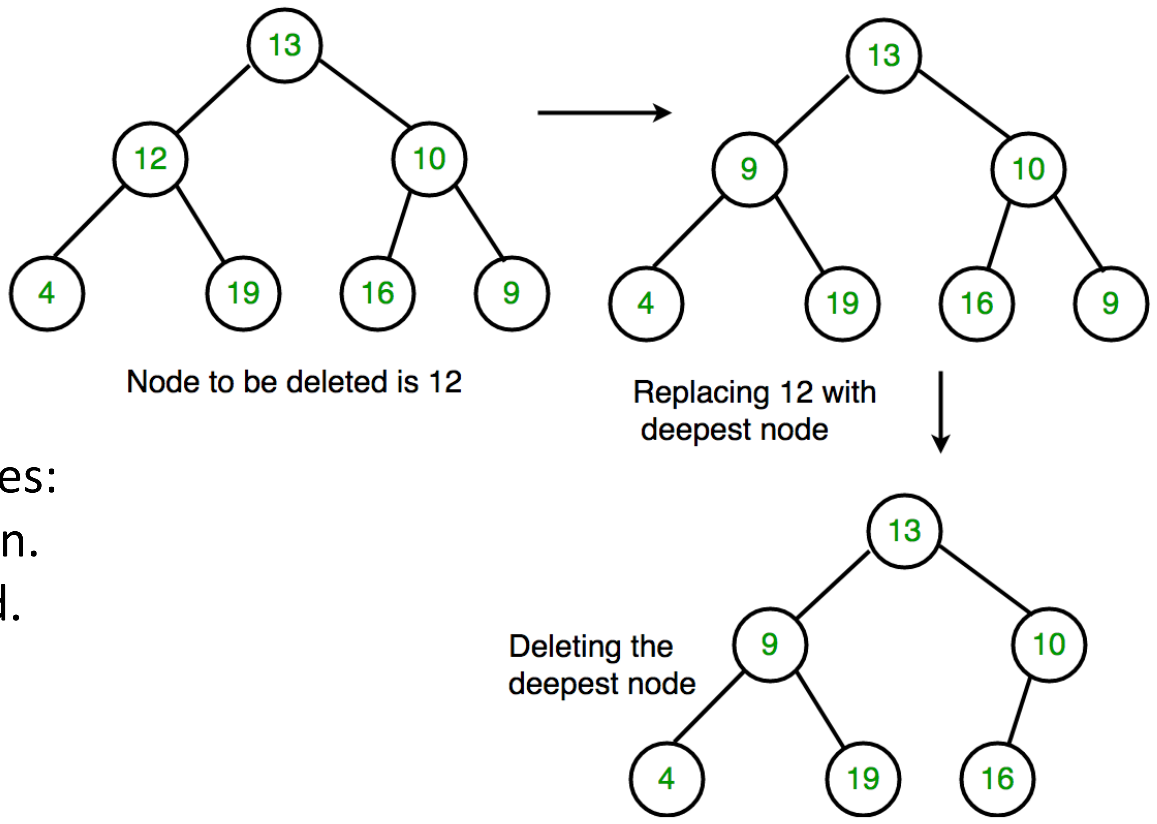
```
   10
 /    \
20     30
         \
          40
```

Output :

```
   10
 /   \
40    30
```

While performing the delete operation on binary trees, there arise a few cases:

- The node to be deleted is a leaf node. That is it does not have any children.
- The node to be deleted is a internal node. That is it have left or right child.
- The node to be deleted is the root node.



In the first case 1, since the node to be deleted is a leaf node, we can simply delete the node without any overheads. But in the next 2 cases, we will have to take care of the children of the node to be deleted.

In order to handle all of the cases, one way to delete a node is to:

- Starting at the root, find the deepest and rightmost node in binary tree and node which we want to delete.
- Replace the deepest rightmost node's data with the node to be deleted.
- Then delete the deepest rightmost node.

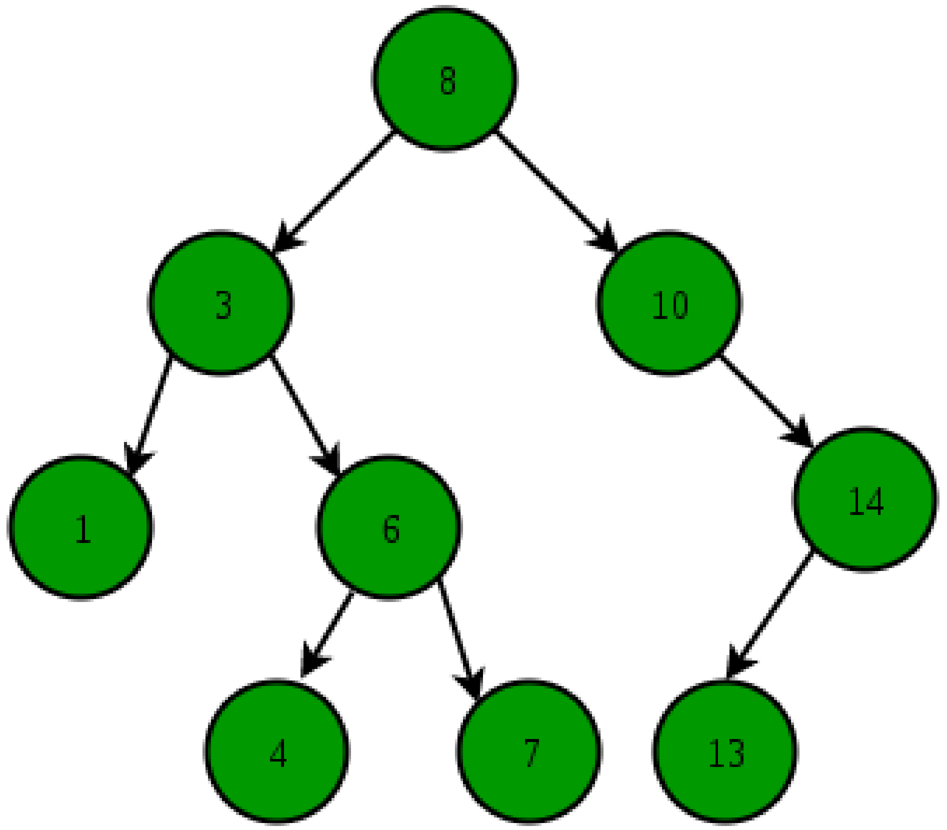
Introduction to Binary Search Trees

Binary Search Tree is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than or equal to the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree. There must be no duplicate nodes.

Sample Binary Search Tree:

The above properties of Binary Search Tree provide an ordering among keys so that the operations like search, minimum and maximum can be done fast in comparison to normal Binary Trees. If there is no ordering, then we may have to compare every key to search a given key.



Searching a Key

Using the property of Binary Search Tree, we can search for an element in O(h) time complexity where **h** is the height of the given BST.

To search a given key in Binary Search Tree,

- first compare it with root,
- if the key is present at root, return root.
- If the key is greater than the root's key, we recur for the right subtree of the root node.
- Otherwise, we recur for the left subtree.

```
struct node* search(struct node* root, int key)
{
// Base Cases: root is null or key is present at root
if (root == NULL || root->key == key)
        return root;
if (root->key < key)
        // Key is greater than root's key
        return search(root->right, key);

    // Key is smaller than root's key
    return search(root->left, key);
}
```

Insertion of a Key

Inserting a new node in the Binary Search Tree is always done at the leaf nodes to maintain the order of nodes in the Tree. The idea is to start searching the given node to be inserted from the root node till we hit a leaf node. Once a leaf node is found, the new node is added as a child of the leaf node.

For Example:

```
     100                               100
    /   \        Insert 40            /    \
  20     500    --------->          20      500
 /  \                              /  \
10   30                           10   30
                                           \
                                          40
```

```c
// A utility function to insert a new node
// with given key in BST
struct node* insert(struct node* node, int key)
{
    /* If the tree is empty, return a new node */
    if (node == NULL) return newNode(key);

    /* Otherwise, recur down the tree */
    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);

    /* return the (unchanged) node pointer */
    return node;
}
```

Deletion in a Binary Search Tree

That is, given a Binary Search Tree and a node to be deleted. The task is to search that node in the given BST and delete it from the BST if it is present.

When we delete a node, three cases may arise:

- ***Node to be deleted is leaf:*** Simply remove from the tree.

```
       50                            50
    /     \         delete(20)      /   \
   30      70       --------->    30     70
  /  \    /  \                     \    /  \
20   40  60   80                   40  60   80
```

- ***Node to be deleted has only one child:*** Copy the child to the node and delete the child.

```
    50                            50
 /     \         delete(30)      /   \
30      70       --------->    40     70
  \    /  \                           /  \
  40  60   80                       60   80
```

- ***Node to be deleted has two children:*** Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.

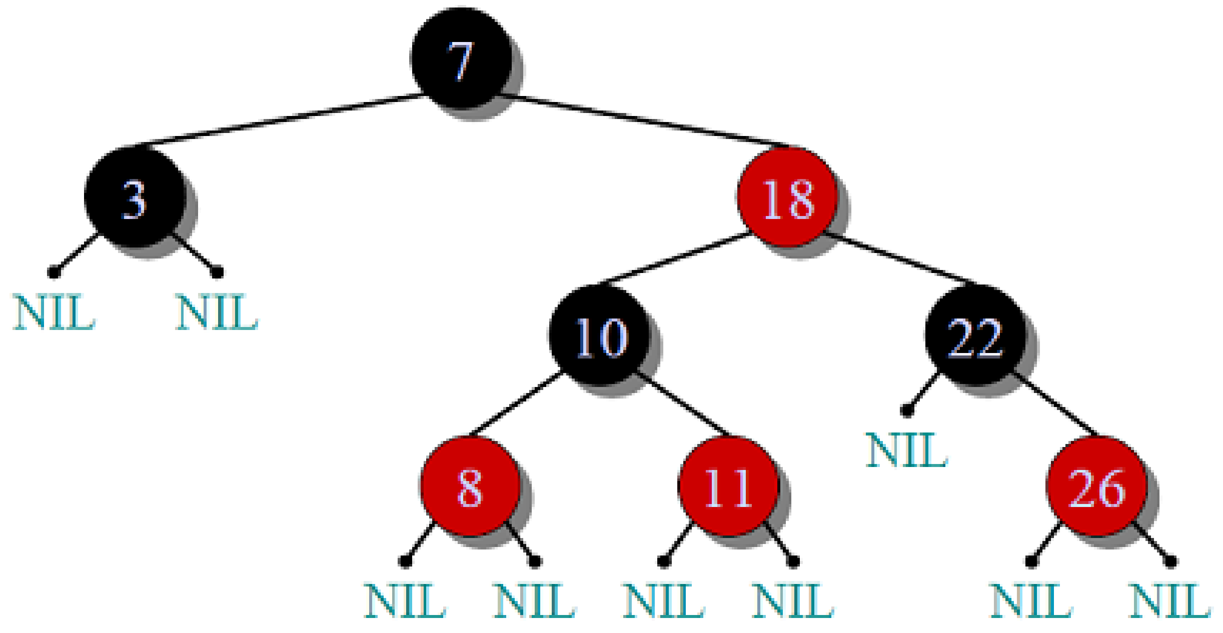
```
    50                             60
 /     \         delete(50)       /   \
40      70       --------->     40    70
       /  \                             \
      60   80                            80
```

Note: *The inorder successor can be obtained by finding the minimum value in the right child of the node.*

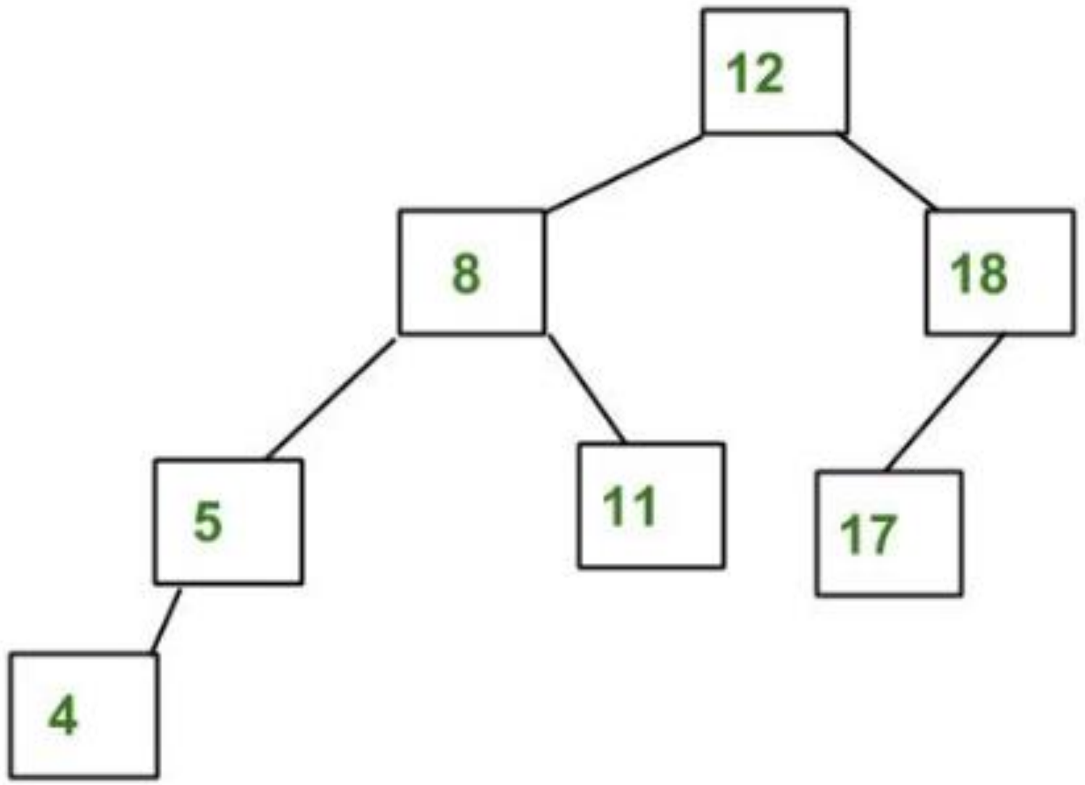
Self Balancing BST

Self-Balancing Binary Search Trees are ***height-balanced*** binary search trees that automatically keeps height as small as possible when insertion and deletion operations are performed on tree. The height is typically maintained in order of Log n so that all operations take O(Log n) time on average.

Examples :
Red Black Tree



AVL Tree:



Applications of BST

Applications of Binary Search tree:

- BSTs are used for indexing.
- It is also used to implement various searching algorithms.
- IT can be used to implement various data structures.

Real-time Application of Binary Search tree:

- BSTs are used for indexing in databases.
- It is used to implement searching algorithms.
- BSTs are used to implement Huffman coding algorithm.
- It is also used to implement dictionaries.

Advantages of Binary Search Tree:

- BST is fast in insertion and deletion when balanced.
- BST is efficient.
- We can also do range queries – find keys between N and M (N <= M).
- BST code is simple as compared to other data structures.

Disadvantages of Binary Search Tree:

- The main disadvantage is that we should always implement a balanced binary search tree. Otherwise the cost of operations may not be logarithmic and degenerate into a linear search on an array.
- Accessing the element in BST is slightly slower than array.
- A BST can be imbalanced or degenerated which can increase the complexity.

Applications of Tree Data Structures

File Systems and Directory Structures 📁

Trees are a natural way to represent the filesystem hierarchy and the folders inside it, allowing fast searches, and efficient manipulation of files and directories.

Representing Hierarchical Relationships 👨‍👩‍👧‍👦

Trees are used to represent the structure of an organization, the hierarchy of a family, or the relationships between entities in various domains.

Sorting and Searching Algorithms 🕵️‍♂️

Popular sorting algorithms such as Heap sort use trees to move elements to their final order, while searching algorithms like Tries use trees to efficiently store and retrieve words or keys.

Advantages and Disadvantages of Tree Data Structures

Advantages 👍

- Efficient and Fast
- Scalable and Adaptable
- Flexible and Easy to Implement

Disadvantages 👎

- Memory Consumption
- Complexity and Maintenance
- Tree Imbalance and Degeneracy

Efficiency and Complexity Analysis: BST

	Average Time Complexity	Worst-Case Time Complexity	Space Complexity
Search	O(log n)	O(n)	O(1)
Insert	O(log n)	O(n)	O(n)
Delete	O(log n)	O(n)	O(1)

Thank you!