Data Structures and Algorithms

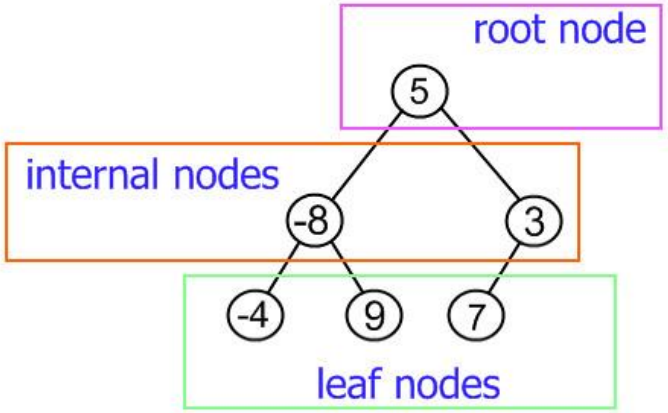
Trong-Hop Do

University of Information Technology, HCM city

Binary tree

Tree introduction

- Unlike Arrays, Linked Lists, Stack and queues, which are linear data structures, trees are hierarchical data structures.
- Tree vocabulary:
  - The topmost node is called root of the tree.
  - The elements that are directly under an element are called its children.
  - The element directly above something is called its parent.
  - Finally, elements with no children are called leaves.



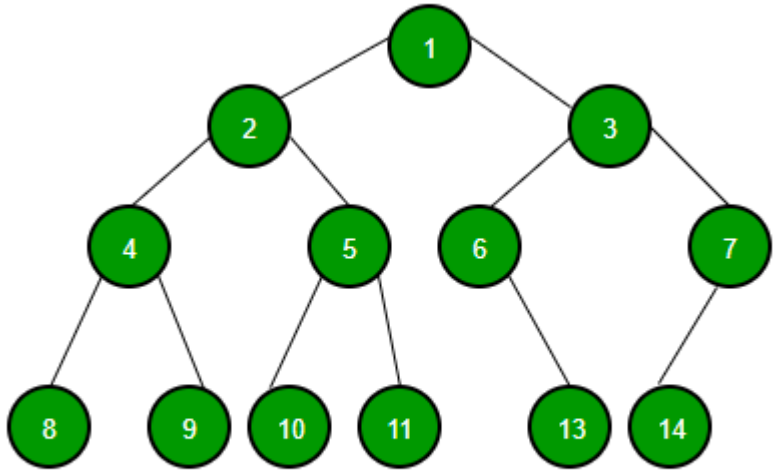
Tree applications

Main applications of trees include:

1. Manipulate hierarchical data.
2. Make information easy to search (see tree traversal).
3. Manipulate sorted lists of data.
4. As a workflow for compositing digital images for visual effects.
5. Router algorithms
6. Form of a multi-stage decision-making (see business chess).

Binary tree

- A tree whose elements have at most 2 children is called a binary tree. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.

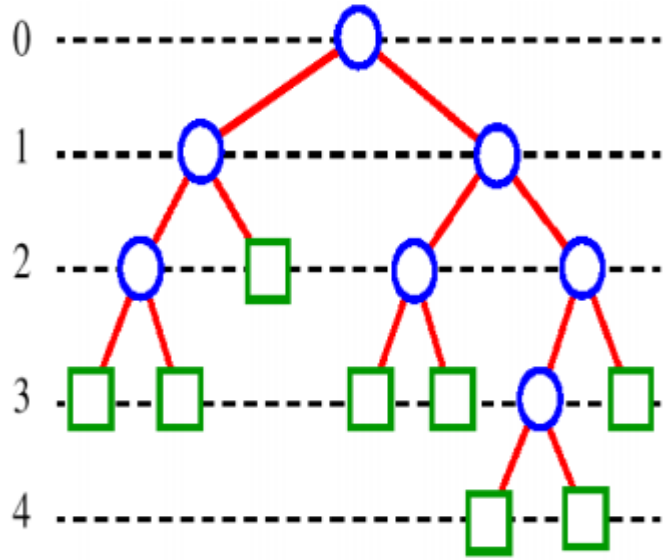


- A Binary Tree node contains following parts

    1. Data
    2. Pointer to left child
    3. Pointer to right child

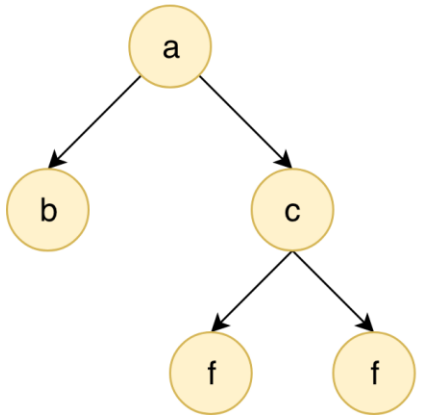
Properties

- The maximum number of nodes at level ‘l’ of a binary tree is 2^l.
- The Maximum number of nodes in a binary tree of height ‘h’ is $2^h - 1$
- In a Binary Tree with N nodes, minimum possible height or the minimum number of levels is Log2(N+1)
- A Binary Tree with L leaves has at least? Log2L + 1 levels
- In Binary tree where every node has 0 or 2 children, the number of leaf nodes is always one more than nodes with two children.

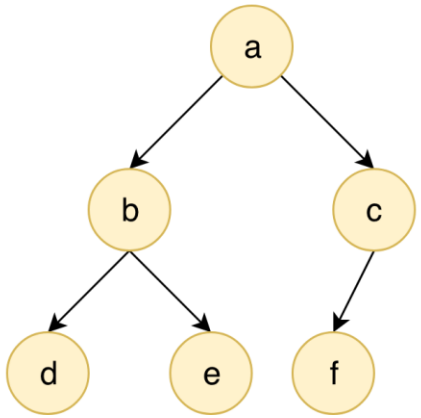


Types of binary tree

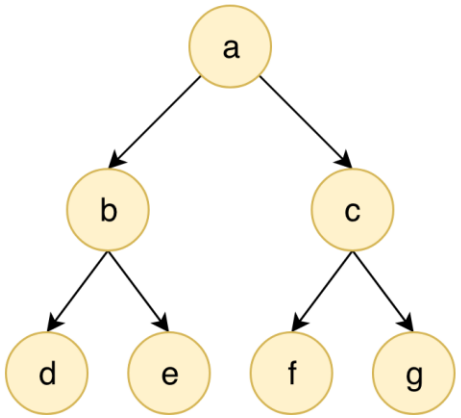
Full Binary Tree
(also balanced)



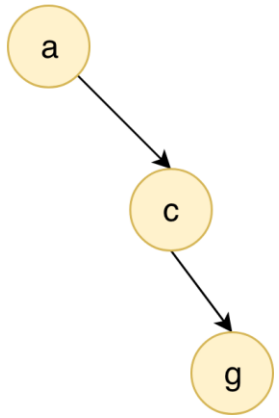
Complete Binary Tree
(also balanced)



Perfect Binary Tree
(also balanced)



Pathological Binary Tree
(not balanced)



Types of Binary Tree

- **Full Binary Tree** A Binary Tree is a full binary tree if every node has 0 or 2 children. The following are the examples of a full binary tree. We can also say a full binary tree is a binary tree in which all nodes except leaf nodes have two children.

Types of Binary Tree

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

Types of Binary Tree

- **Perfect Binary Tree** A Binary tree is a Perfect Binary Tree in which all the internal nodes have two children and all leaf nodes are at the same level.

Types of Binary Tree

- **A degenerate (or pathological) tree** A Tree where every internal node has one child. Such trees are performance-wise same as linked list.

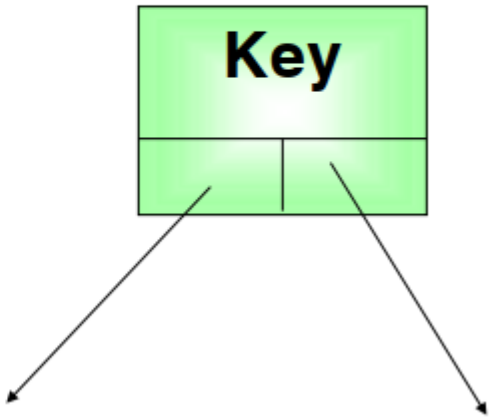
Types of Binary Tree

- **Balanced Binary Tree** is a Binary tree in which height of the left and the right sub-trees of every node may differ by at most 1.

Binary tree data structure

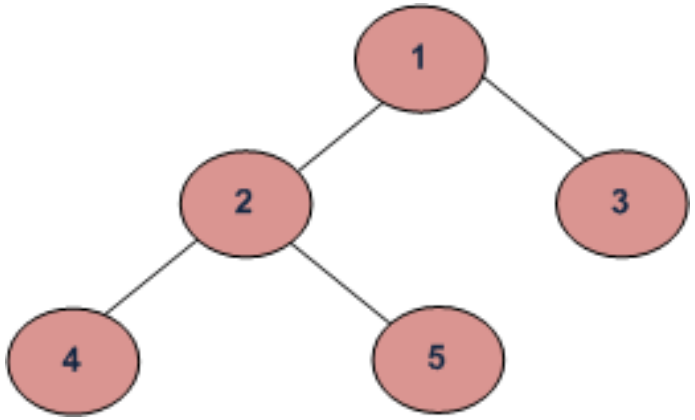
```
typedef struct tagTNode
{
   Data     Key;
   struct tagTNode *pLeft;
   struct tagTNode *pRight;
}TNode;

typedef TNode *TREE;
```



Tree Traversals

- 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.
- Depth First Traversals:
  - (a) Inorder (Left, Root, Right) : 4 2 5 1 3
  - (b) Preorder (Root, Left, Right) : 1 2 4 5 3
  - (c) Postorder (Left, Right, Root) : 4 5 2 3 1
- Breadth First or Level Order Traversal : 1 2 3 4 5



Depth First Traversals

Inorder Traversal (Practice):

```
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)
```

Depth First Traversals

Preorder Traversal (Practice):

```
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)
```

Depth First Traversals

Postorder Traversal (Practice):

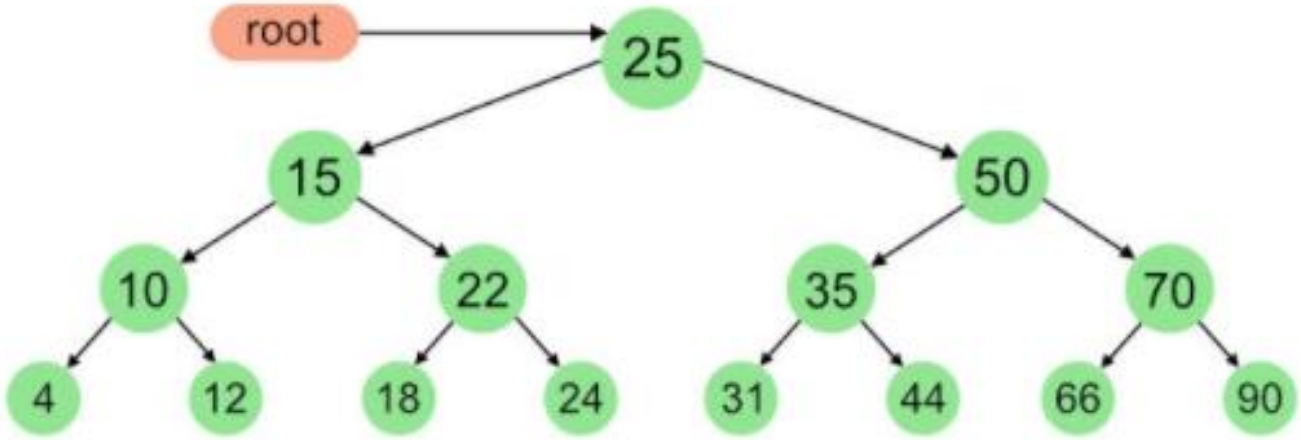
```
Algorithm Postorder(tree)
   1. Traverse the left subtree, i.e., call Postorder(left-subtree)
   2. Traverse the right subtree, i.e., call Postorder(right-subtree)
   3. Visit the root.
```

Depth First Traversals

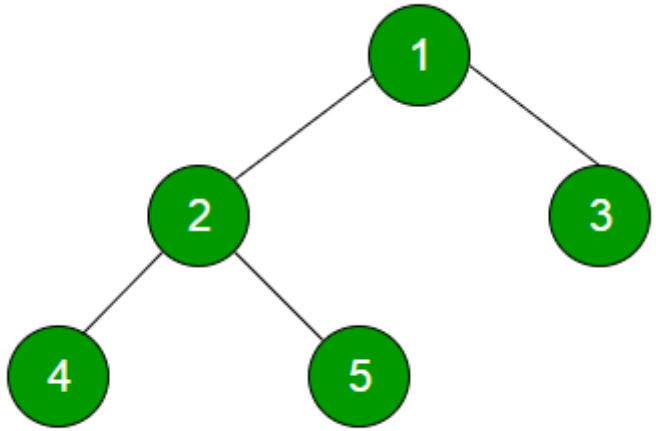
InOrder(root) visits nodes in the following order:
4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

A Pre-order traversal visits nodes in the following order:
25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

A Post-order traversal visits nodes in the following order:
4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25



Breadth First Traversal



- Level order traversal of the above tree is 1 2 3 4 5

Breadth First Traversal

- Method 1 (Using function to print a given level)

```
/*Function to print level order traversal of tree*/
printLevelorder(tree)
for d = 1 to height(tree)
   printGivenLevel(tree, d);

/*Function to print all nodes at a given level*/
printGivenLevel(tree, level)
if tree is NULL then return;
if level is 1, then
    print(tree->data);
else if level greater than 1, then
    printGivenLevel(tree->left, level-1);
    printGivenLevel(tree->right, level-1);
```

Breadth First Traversal

- Method 2 (Using queue)

```
printLevelorder(tree)
1) Create an empty queue q
2) temp_node = root /*start from root*/
3) Loop while temp_node is not NULL
    a) print temp_node->data.
    b) Enqueue temp_node's left and right children to q
    c) temp_node = Dequeue a node from q.
```

BFS vs DFS for Binary Tree

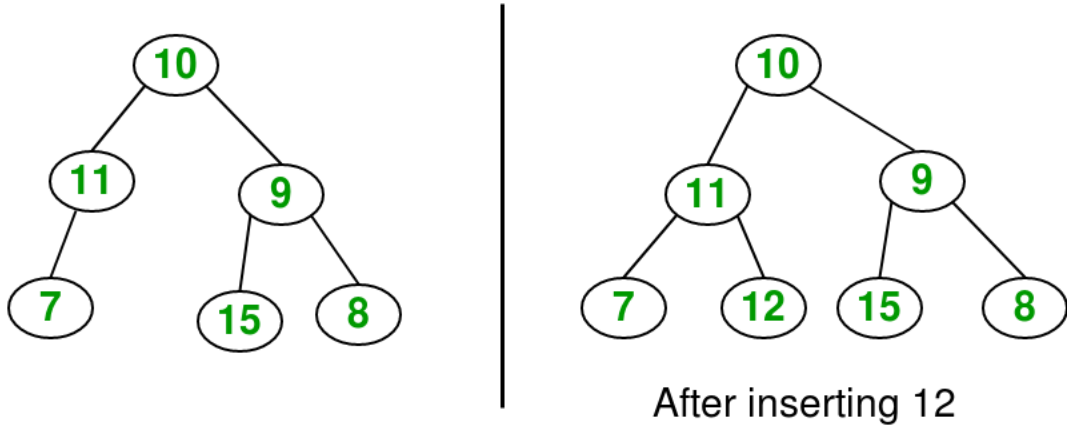
- Is there any difference in terms of Time Complexity?
  - All four traversals require O(n) time as they visit every node exactly once.

- Is there any difference in terms of Extra Space?
  - Extra Space required for Level Order Traversal is O(w) where w is maximum width of Binary Tree. In level order traversal, queue one by one stores nodes of different level.
  - Extra Space required for Depth First Traversals is O(h) where h is maximum height of Binary Tree. In Depth First Traversals, stack (or function call stack) stores all ancestors of a node.

- How to Pick One?
  - Extra Space can be one factor (Explained above)
  - Depth First Traversals are typically recursive and recursive code requires function call overheads.
  - The most important points is, BFS starts visiting nodes from root while DFS starts visiting nodes from leaves. So if our problem is to search something that is more likely to closer to root, we would prefer BFS. And if the target node is close to a leaf, we would prefer DFS.

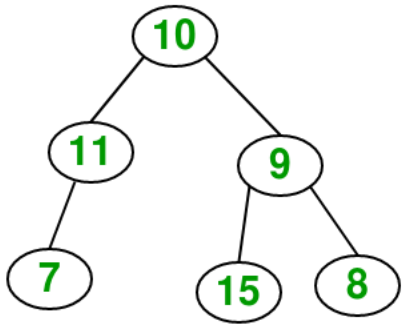
Insertion in a Binary Tree in level order

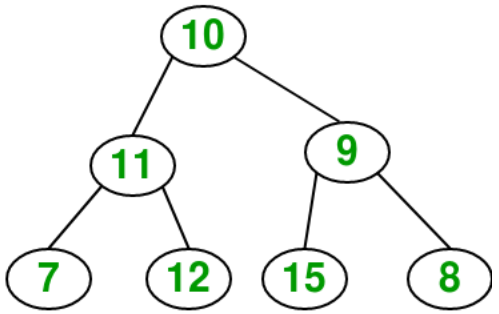
- Given a binary tree and a key, insert the key into the binary tree at the first position available in level order.



- Idea:
  - iterative level order traversal of the given tree using queue.
  - If we find a node whose left child is empty, we make new key as left child of the node.
  - Else if we find a node whose right child is empty, we make the new key as right child.
  - Keep traversing the tree until we find a node whose either left or right is empty.

```
Node* InsertNode(Node* root, int data){
    if (root == NULL) {
        root = CreateNode(data); return root;
    }
    queue<Node*> q; q.push(root);
    while (!q.empty()) {
        Node* temp = q.front(); q.pop();
        if (temp->left != NULL)
            q.push(temp->left);
        else {
            temp->left = CreateNode(data); return root;
        }
        if (temp->right != NULL)
            q.push(temp->right);
        else {
            temp->right = CreateNode(data); return root;
        }
    }
}
```

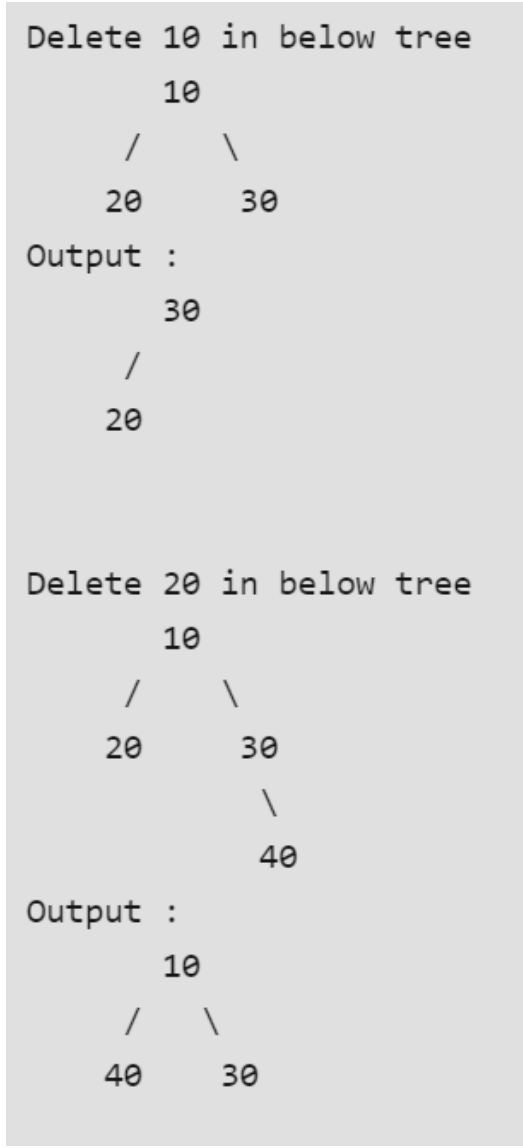




After inserting 12

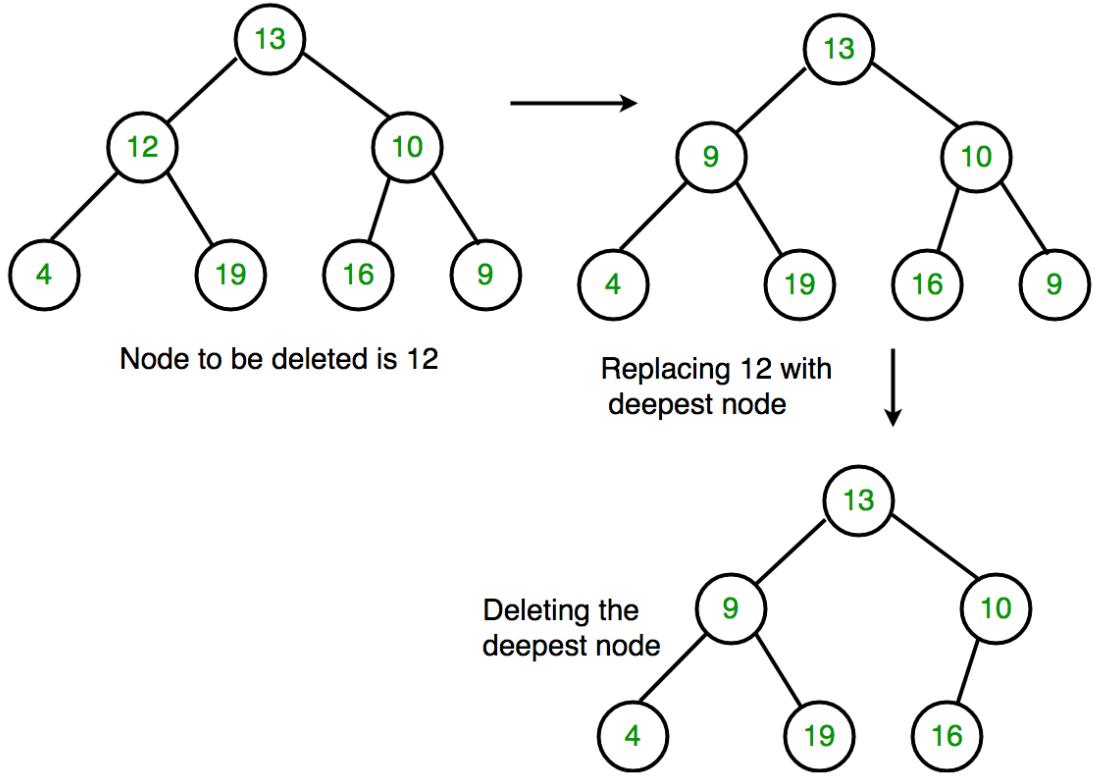
Deletion in a Binary Tree

- Given a binary tree, delete a node from it by making sure that tree shrinks from the bottom (i.e. the deleted node is replaced by bottom most and rightmost node).
- Algorithm
  1. Starting at root, find the deepest and rightmost node in binary tree and node which we want to delete.
  2. Replace the deepest rightmost node's data with node to be deleted.
  3. Then delete the deepest rightmost node.



Deletion in a Binary Tree

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



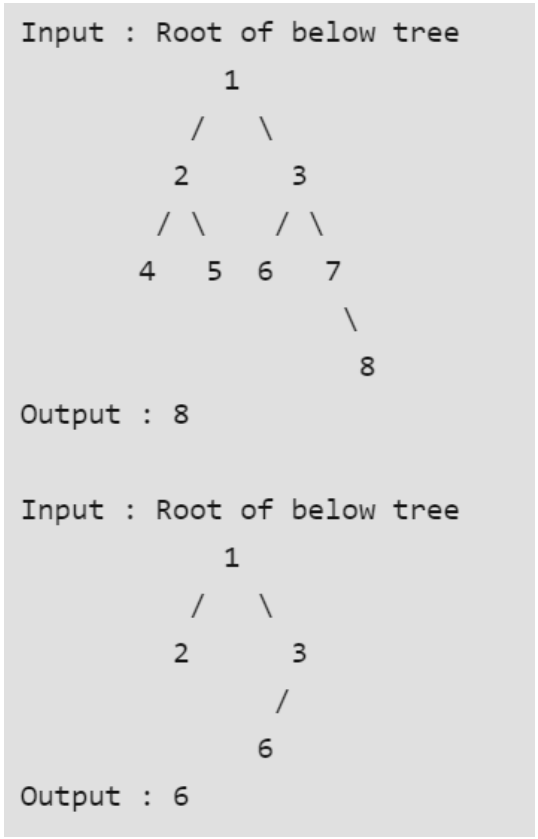
Find the Deepest Node in a Binary Tree

- **Method 1** :

The idea is to do Inorder traversal of given binary tree. While doing Inorder traversal, we pass level of current node also. We keep track of maximum level seen so far and value of deepest node seen so far.

- **Method 2 :**

The idea here is to find the height of the given tree and then print the node at the bottom-most level.



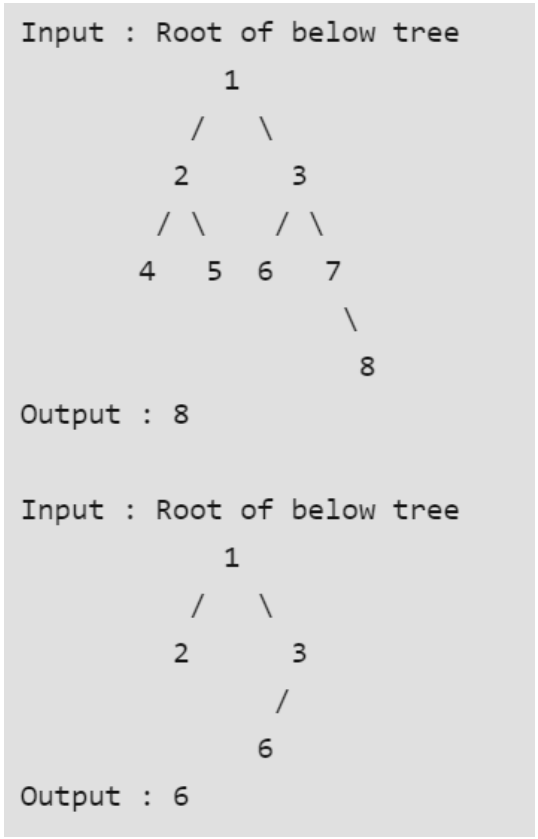
Find the Deepest Node in a Binary Tree

- **Method 1** :

The idea is to do Inorder traversal of given binary tree. While doing Inorder traversal, we pass level of current node also. We keep track of maximum level seen so far and value of deepest node seen so far.

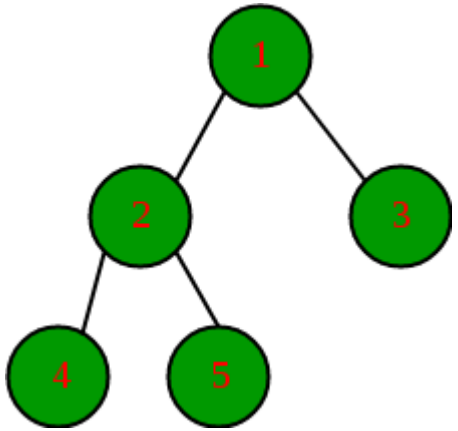
- **Method 2 :**

The idea here is to find the height of the given tree and then print the node at the bottom-most level.



Check whether a binary tree is a full binary tree or not

- A full binary tree is defined as a binary tree in which all nodes have either zero or two child nodes. Conversely, there is no node in a full binary tree, which has one child node.
- To check whether a binary tree is a full binary tree we need to test the following cases:-



1) If a binary tree node is NULL then it is a full binary tree.

2) If a binary tree node does have empty left and right sub-trees, then it is a full binary tree by definition.

3) If a binary tree node has left and right sub-trees, then it is a part of a full binary tree by definition. In this case recursively check if the left and right sub-trees are also binary trees themselves.

4) In all other combinations of right and left sub-trees, the binary tree is not a full binary tree.

Check whether a binary tree is a full binary tree or not

```
/* This function tests if a binary tree is a full binary tree. */
bool isFullTree (struct Node* root)
{
    // If empty tree
    if (root == NULL)
        return true;

    // If leaf node
    if (root->left == NULL && root->right == NULL)
        return true;

    // If both left and right are not NULL, and left & right subtrees  are full
    if ((root->left) && (root->right))
        return (isFullTree(root->left) && isFullTree(root->right));

    // We reach here when none of the above if conditions work
    return false;
}
```