BINARY TREE

1.LEVEL ORDER TRAVERSAL

Method 1 (Use function to print a current level)

Algorithm:

There are basically two functions in this method. One is to print all nodes at a given level (printCurrentLevel), and other is to print level order traversal of the tree (printLevelorder). printLevelorder makes use of printCurrentLevel to print nodes at all levels one by one starting from the root.

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

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

CODE:

```
/* Function to print level
order traversal a tree*/

void printLevelOrder(node* root)
{
    int h = height(root);
    int i;
    for (i = 1; i <= h; i++)
        printCurrentLevel(root, i);
}

/* Print nodes at a current level */
void printCurrentLevel(node* root, int level)
{
    if (root == NULL)
        return;
    if (level == 1)
        cout << root->data << " ";
    else if (level > 1)
    {
        printCurrentLevel(root->left, level-1);
        printCurrentLevel(root->right, level-1);
    }
}

/* Compute the "height" of a tree -- the number of
```

```
    nodes along the longest path from the root node
    down to the farthest leaf node.*/
int height(node* node)
{
    if (node == NULL)
        return 0;
    else
    {
        /* compute the height of each subtree */
        int lheight = height(node->left);
        int rheight = height(node->right);

        /* use the larger one */
        if (lheight > rheight)
        {
            return(lheight + 1);
        }
        else {
          return(rheight + 1);
        }
    }
}
```

Time Complexity: O(n^2) in worst case. For a skewed tree, printGivenLevel() takes O(n) time where n is the number of nodes in the skewed tree. So time complexity of printLevelOrder() is O(n) + O(n-1) + O(n-2) + .. + O(1) which is O(n^2).

Space Complexity: O(n) in worst case. For a skewed tree, printGivenLevel() uses O(n) space for call stack. For a Balanced tree, the call stack uses O(log n) space, (i.e., the height of the balanced tree)

Method 2 (Using queue)

Algorithm:

For each node, first the node is visited and then it's child nodes are put in a FIFO 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 children
      (first left then right children) to q
    c) Dequeue a node from q.
/ Iterative method to find height of Binary Tree
void printLevelOrder(Node *root)
{
    // Base Case
    if (root == NULL)  return;

    // Create an empty queue for level order traversal
    queue<Node *> q;
```

```
    // Enqueue Root and initialize height
    q.push(root);

    while (q.empty() == false)
    {
        // Print front of queue and remove it from queue
        Node *node = q.front();
        cout << node->data << " ";
        q.pop();

        /* Enqueue left child */
        if (node->left != NULL)
            q.push(node->left);

        /*Enqueue right child */
        if (node->right != NULL)
            q.push(node->right);
    }
}
```

Time Complexity: O(n) where n is the number of nodes in the binary tree
Space Complexity: O(n) where n is the number of nodes in the binary tree

---

REVERSE LEVEL ORDER TRAVERSAL

METHOD 1 (Recursive function to print a given level)

We can easily modify the method 1 of the normal level order traversal. In method 1, we have a method printGivenLevel() which prints a given level number. The only thing we need to change is, instead of calling printGivenLevel() from the first level to the last level, we call it from the last level to the first level.

CODE:

```
/* Function to print REVERSE
level order traversal a tree*/
void reverseLevelOrder(node* root)
{
    int h = height(root);
    int i;
    for (i=h; i>=1; i--)        printGivenLevel(root, i);
}

/* Print nodes at a given level */
void printGivenLevel(node* root, int level)
{
    if (root == NULL)
        return;
    if (level == 1)
        cout << root->data << " ";
    else if (level > 1)
    {
```

```
        printGivenLevel(root->left, level - 1);
        printGivenLevel(root->right, level - 1);
    }
}

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

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

Time Complexity: The worst-case time complexity of this method is O(n^2). For a skewed tree, printGivenLevel() takes O(n) time where n is the number of nodes in the skewed tree. So time complexity of printLevelOrder() is O(n) + O(n-1) + O(n-2) + .. + O(1) which is O(n^2)

METHOD 2 (Using Queue and Stack)

The method 2 of normal level order traversal can also be easily modified to print level order traversal in reverse order. The idea is to use a deque(double-ended queue) to get the reverse level order. A deque allows insertion and deletion at both the ends. If we do normal level order traversal and instead of printing a node, push the node to a stack and then print the contents of the deque, we get “5 4 3 2 1” for the above example tree, but the output should be “4 5 2 3 1”. So to get the correct sequence (left to right at every level), we process children of a node in reverse order, we first push the right subtree to the deque, then process the left subtree.

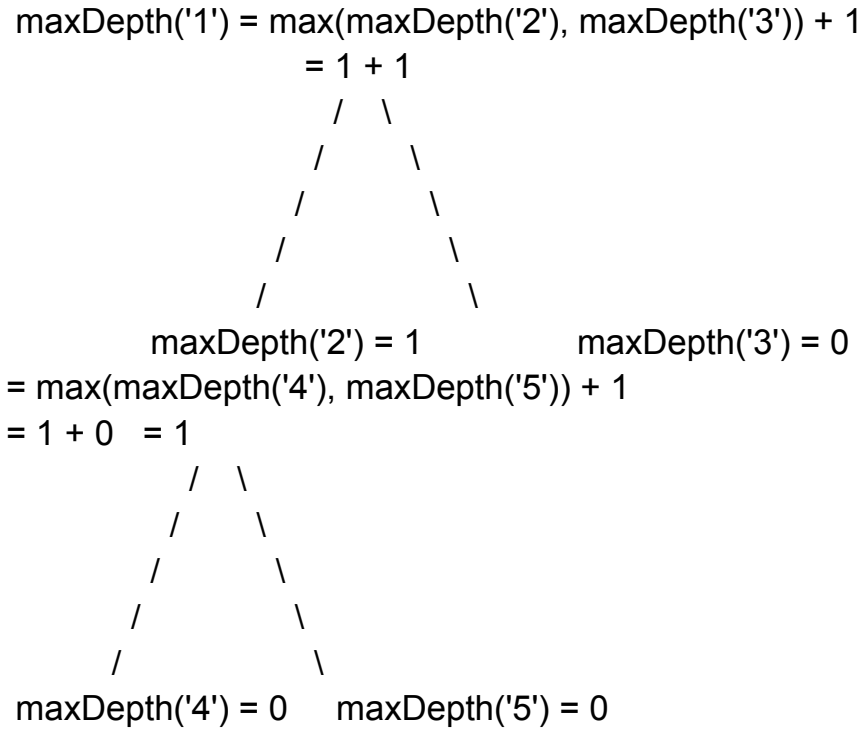
```
* Given a binary tree, print its nodes in reverse level order */
void reverseLevelOrder(node* root)
{
    stack <node *> S;
    queue <node *> Q;
    Q.push(root);

    // Do something like normal level order traversal order. Following are the
    // differences with normal level order traversal
```

```
    // 1) Instead of printing a node, we push the node to stack
    // 2) Right subtree is visited before left subtree
    while (Q.empty() == false)
    {
        /* Dequeue node and make it root */
        root = Q.front();
        Q.pop();
        S.push(root);

        /* Enqueue right child */
        if (root->right)
            Q.push(root->right); // NOTE: RIGHT CHILD IS ENQUEUED BEFORE LEFT

        /* Enqueue left child */
        if (root->left)
            Q.push(root->left);
    }

    // Now pop all items from stack one by one and print them
    while (S.empty() == false)
    {
        root = S.top();
        cout << root->data << " ";
        S.pop();
    }
}
```

3.Write a Program to Find the Maximum Depth or Height of a Tree

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.

```
maxDepth()
1. If tree is empty then return 0
2. Else
    (a) Get the max depth of left subtree recursively  i.e.,
        call maxDepth( tree->left-subtree)
    (a) Get the max depth of right subtree recursively  i.e.,
        call maxDepth( tree->right-subtree)
    (c) Get the max of max depths of left and right
        subtrees and add 1 to it for the current node.
        max_depth = max(max dept of left subtree,
                        max depth of right subtree)
                        + 1
    (d) Return max_depth
```

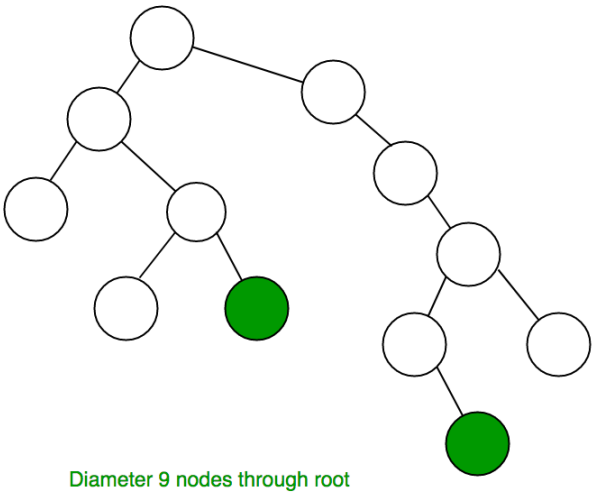



```
/* Compute the "maxDepth" of a tree -- the number of
    nodes along the longest path from the root node
    down to the farthest leaf node.*/

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);
    }
}
```

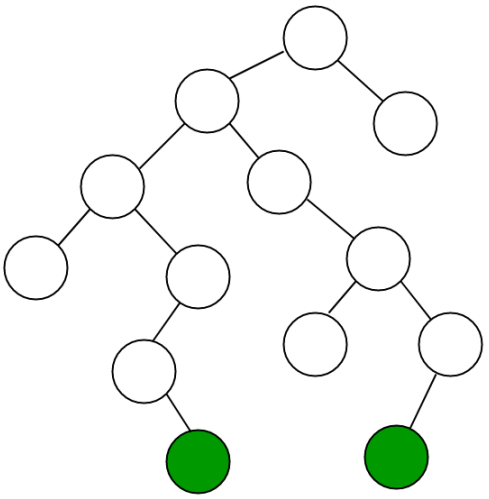
Time Complexity: O(n)

4.DIAMETER OF BINARY TREE

The diameter of a tree (sometimes called the width) is the number of nodes on the longest path between two end nodes. The diagram below shows two trees each with diameter nine, the leaves that form the ends of the longest path are shaded (note that there is more than one path in each tree of length nine, but no path longer than nine nodes).



Diameter 9 nodes through root



Diameter 9 nodes not through root

The diameter of a tree T is the largest of the following quantities:

- the diameter of T's left subtree.
- the diameter of T's right subtree.
- the longest path between leaves that goes through the root of T (this can be computed from the heights of the subtrees of T)

Code :

```
// returns max of two integers
int max(int a, int b) { return (a > b) ? a : b; }

// function to Compute height of a tree.
int height(struct node* node);

// Function to get diameter of a binary tree
int diameter(struct node* tree)
{
    // base case where tree is empty
    if (tree == NULL)
        return 0;

    // get the height of left and right sub-trees
    int lheight = height(tree->left);
    int rheight = height(tree->right);

    // get the diameter of left and right sub-trees
    int ldiameter = diameter(tree->left);
    int rdiameter = diameter(tree->right);

    // Return max of following three
    // 1) Diameter of left subtree
    // 2) Diameter of right subtree
    // 3) Height of left subtree + height of right subtree + 1
    return max(lheight + rheight + 1,
           max(ldiameter, rdiameter));
}
```

```
// The function Compute the "height" of a tree. Height is
// the number f nodes along the longest path from the root
// node down to the farthest leaf node.

int height(struct node* node)
{
    // base case tree is empty
    if (node == NULL)
        return 0;

    // If tree is not empty then height = 1 + max of left
    // height and right heights
    return 1 + max(height(node->left), height(node->right));
}
```

Time Complexity: $O(n^2)$

Optimized implementation: The above implementation can be optimized by calculating the height in the same recursion rather than calling a height() separately.

```
int diameterOpt(struct node* root, int* height)
{
    // lh --> Height of left subtree
    // rh --> Height of right subtree
    int lh = 0, rh = 0;

    // ldiameter  --> diameter of left subtree
    // rdiameter  --> Diameter of right subtree
    int ldiameter = 0, rdiameter = 0;

    if (root == NULL) {
        *height = 0;
        return 0; // diameter is also 0
    }

    // Get the heights of left and right subtrees in lh and
    // rh And store the returned values in ldiameter and
    // ldiameter
    ldiameter = diameterOpt(root->left, &lh);
    rdiameter = diameterOpt(root->right, &rh);

    // Height of current node is max of heights of left and
    // right subtrees plus 1
    *height = max(lh, rh) + 1;

    return max(lh + rh + 1, max(ldiameter, rdiameter));
}
```

Time Complexity: O(n)

5.mirror of tree

Mirror of a Tree: Mirror of a Binary Tree T is another Binary Tree M(T) with left and right children of all non-leaf nodes interchanged.

Method 1 (Recursive)

(1) Call Mirror for left-subtree i.e., Mirror(left-subtree)

(2) Call Mirror for right-subtree i.e., Mirror(right-subtree)

(3) Swap left and right subtrees

temp = left-subtree

left-subtree = right-subtree

right-subtree = temp

/* Change a tree so that the roles of the left and right pointers are swapped at every node.

```
void mirror(struct Node* node)
{
   if (node == NULL)
      return;
   else
   {
      struct Node* temp;

      /* do the subtrees */
      mirror(node->left);
      mirror(node->right);

      /* swap the pointers in this node */
      temp       = node->left;
      node->left = node->right;
      node->right = temp;
   }
}
```

Time & Space Complexities: Worst-case Time complexity is O(n) and for space complexity, If we don't consider the size of the recursive stack for function calls then O(1) otherwise O(h) where h is the height of the tree

Method 2 (Iterative)

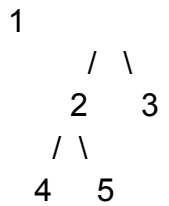
The idea is to do queue based level order traversal. While doing traversal, swap left and right children of every node.

```
void mirror(Node* root)
{
   if (root == NULL)
      return;
```

```
    queue<Node*> q;
    q.push(root);

    // Do BFS. While doing BFS, keep swapping
    // left and right children
    while (!q.empty())
    {
        // pop top node from queue
        Node* curr = q.front();
        q.pop();

        // swap left child with right child
        swap(curr->left, curr->right);

        // push left and right children
        if (curr->left)
            q.push(curr->left);
        if (curr->right)
            q.push(curr->right);
    }
}
```

6.inorder traversal of a tree both using recursion and iteration

Using Stack is the obvious way to traverse tree without recursion
1) Create an empty stack S.
2) Initialize current node as root
3) Push the current node to S and set current = current->left until current is NULL
4) If current is NULL and stack is not empty then
    a) Pop the top item from stack.
    b) Print the popped item, set current = popped_item->right
    c) Go to step 3.
5) If current is NULL and stack is empty then we are done.



Step 1 Creates an empty stack: S = NULL

Step 2 sets current as address of root: current -> 1

Step 3 Pushes the current node and set current = current->left
    until current is NULL
    current -> 1
    push 1: Stack S -> 1
    current -> 2
    push 2: Stack S -> 2, 1
    current -> 4

push 4: Stack S -> 4, 2, 1
current = NULL

Step 4 pops from S
a) Pop 4: Stack S -> 2, 1
b) print "4"
c) current = NULL /*right of 4 */ and go to step 3
Since current is NULL step 3 doesn't do anything.

Step 4 pops again.
a) Pop 2: Stack S -> 1
b) print "2"
c) current -> 5/*right of 2 */ and go to step 3

Step 3 pushes 5 to stack and makes current NULL
Stack S -> 5, 1
current = NULL

Step 4 pops from S
a) Pop 5: Stack S -> 1
b) print "5"
c) current = NULL /*right of 5 */ and go to step 3
Since current is NULL step 3 doesn't do anything

Step 4 pops again.
a) Pop 1: Stack S -> NULL
b) print "1"
c) current -> 3 /*right of 1 */

Step 3 pushes 3 to stack and makes current NULL
Stack S -> 3
current = NULL

Step 4 pops from S
a) Pop 3: Stack S -> NULL
b) print "3"
c) current = NULL /*right of 3 */

Traversal is done now as stack S is empty and current is NULL.

```
/* Iterative function for inorder tree
   traversal */
void inOrder(struct Node *root)
{
    stack<Node *> s;
    Node *curr = root;

    while (curr != NULL || s.empty() == false)
    {
        /* Reach the left most Node of the
           curr Node */
        while (curr !=  NULL)
```

```
        {
            /* place pointer to a tree node on
               the stack before traversing
              the node's left subtree */
            s.push(curr);
            curr = curr->left;
        }

        /* Current must be NULL at this point */
        curr = s.top();
        s.pop();

        cout << curr->data << " ";

        /* we have visited the node and its
           left subtree.  Now, it's right
           subtree's turn */
        curr = curr->right;

    } /* end of while */
}
```

Time Complexity: O(n)

Inorder Tree Traversal without recursion and without stack!

Using Morris Traversal, we can traverse the tree without using stack and recursion. The idea of Morris Traversal is based on Threaded Binary Tree. In this traversal, we first create links to Inorder successor and print the data using these links, and finally revert the changes to restore original tree.

1. Initialize current as root
2. While current is not NULL
   If the current does not have left child
      a) Print current's data
      b) Go to the right, i.e., current = current->right
   Else
      a) Find rightmost node in current left subtree OR
            node whose right child == current.
         If we found right child == current
            a) Update the right child as NULL of that node whose right child is current
            b) Print current's data
            c) Go to the right, i.e. current = current->right
         Else
            a) Make current as the right child of that rightmost
               node we found; and
            b) Go to this left child, i.e., current = current->left

Although the tree is modified through the traversal, it is reverted back to its original shape after the completion. Unlike Stack based traversal, no extra space is required for this traversal

```
/* Function to traverse the binary tree without recursion
   and without stack */
void MorrisTraversal(struct tNode* root)
{
    struct tNode *current, *pre;

    if (root == NULL)
        return;

    current = root;
    while (current != NULL) {

        if (current->left == NULL) {
            printf("%d ", current->data);
            current = current->right;
        }
        else {

            /* Find the inorder predecessor of current */
            pre = current->left;
            while (pre->right != NULL
                   && pre->right != current)
                pre = pre->right;

            /* Make current as the right child of its
               inorder predecessor */
            if (pre->right == NULL) {
                pre->right = current;
                current = current->left;
            }

            /* Revert the changes made in the 'if' part to
               restore the original tree i.e., fix the right
               child of predecessor */
            else {
                pre->right = NULL;
                printf("%d ", current->data);
                current = current->right;
            } /* End of if condition pre->right == NULL */
        } /* End of if condition current->left == NULL*/
    } /* End of while */
}
```

Time Complexity : O(n) If we take a closer look, we can notice that every edge of the tree is traversed at most two times. And in the worst case, the same number of extra edges (as input tree) are created and removed.

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

```
/* Given a binary tree, print its nodes in inorder*/
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);
}
```

---

7. Preorder traversal using iterative and recursion

Given a Binary Tree, write an iterative function to print the Preorder traversal of the given binary tree.
. To convert an inherently recursive procedure to iterative, we need an explicit stack. Following is a simple stack based iterative process to print Preorder traversal.

1) Create an empty stack *nodeStack* and push root node to stack.
2) Do the following while *nodeStack* is not empty.
….**a)** Pop an item from the stack and print it.
….**b)** Push right child of a popped item to stack
….**c)** Push left child of a popped item to stack
The right child is pushed before the left child to make sure that the left subtree is processed first.

```
/ An iterative process to print preorder traversal of Binary tree
void iterativePreorder(node* root)
{
    // Base Case
    if (root == NULL)
        return;

    // Create an empty stack and push root to it
    stack<node*> nodeStack;
    nodeStack.push(root);

    /* Pop all items one by one. Do following for every popped item
       a) print it
       b) push its right child
       c) push its left child
    Note that right child is pushed first so that left is processed first */
    while (nodeStack.empty() == false)
     {
        // Pop the top item from stack and print it
        struct node* node = nodeStack.top();
        printf("%d ", node->data);
        nodeStack.pop();
```

```
        // Push right and left children of the popped node to stack
        if (node->right)
            nodeStack.push(node->right);
        if (node->left)
            nodeStack.push(node->left);
    }
}
```

Time Complexity: O(N)

Auxiliary Space: O(N), where N is the total number of nodes in the tree.

Space Optimized Solution: The idea is to start traversing the tree from the root node, and keep printing the left child while exists and simultaneously, push the right child of every node in an auxiliary stack. Once we reach a null node, pop a right child from the auxiliary stack and repeat the process while the auxiliary stack is not-empty.

```
/ Iterative function to do Preorder traversal of the tree
void preorderIterative(Node* root)
{
    if (root == NULL)
        return;

    stack<Node*> st;
     // start from root node (set current node to root node)
    Node* curr = root;
      // run till stack is not empty or current is
    // not NULL
    while (!st.empty() || curr != NULL) {
        // Print left children while exist
        // and keep pushing right into the
        // stack.
        while (curr != NULL) {
            cout << curr->data << " ";

            if (curr->right)
                st.push(curr->right);

            curr = curr->left;
        }

        // We reach when curr is NULL, so We
        // take out a right child from stack
        if (st.empty() == false) {
            curr = st.top();
            st.pop();
        }
    }
}
```

Time Complexity: O(N)
Auxiliary Space: O(H), where H is the height of the tree.

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)

```
* Given a binary tree, print its nodes in preorder*/
void printPreorder(struct Node* node)
{
    if (node == NULL)
        return;

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

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

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

8.postorder traversal of a tree

Using one stack

The idea is to move down to leftmost node using left pointer. While moving down, push root and root's right child to stack. Once we reach leftmost node, print it if it doesn't have a right child. If it has a right child, then change root so that the right child is processed before.

1.1 Create an empty stack

2.1 Do following while root is not NULL

a) Push root's right child and then root to stack.

b) Set root as root's left child.

2.2 Pop an item from stack and set it as root.

a) If the popped item has a right child and the right child

is at top of stack, then remove the right child from stack,

push the root back and set root as root's right child.

b) Else print root's data and set root as NULL.

2.3 Repeat steps 2.1 and 2.2 while stack is not empty.

1. Right child of 1 exists.
Push 3 to stack. Push 1 to stack. Move to left child.
Stack: 3, 1

2. Right child of 2 exists.
Push 5 to stack. Push 2 to stack. Move to left child.
Stack: 3, 1, 5, 2

3. Right child of 4 doesn't exist. '
Push 4 to stack. Move to left child.
Stack: 3, 1, 5, 2, 4

4. Current node is NULL.
Pop 4 from stack. Right child of 4 doesn't exist.
Print 4. Set current node to NULL.
Stack: 3, 1, 5, 2

5. Current node is NULL.
Pop 2 from stack. Since right child of 2 equals stack top element,
pop 5 from stack. Now push 2 to stack.
Move current node to right child of 2 i.e. 5
Stack: 3, 1, 2

6. Right child of 5 doesn't exist. Push 5 to stack. Move to left child.
Stack: 3, 1, 2, 5

7. Current node is NULL. Pop 5 from stack. Right child of 5 doesn't exist.
Print 5. Set current node to NULL.
Stack: 3, 1, 2

8. Current node is NULL. Pop 2 from stack.
Right child of 2 is not equal to stack top element.
Print 2. Set current node to NULL.
Stack: 3, 1

9. Current node is NULL. Pop 1 from stack.
Since right child of 1 equals stack top element, pop 3 from stack.
Now push 1 to stack. Move current node to right child of 1 i.e. 3
Stack: 1

10. Repeat the same as above steps and Print 6, 7 and 3.
Pop 1 and Print 1.

```
// An iterative function to do postorder traversal of a given binary tree
void postOrderIterative(struct Node* root)
{
    // Check for empty tree
    if (root == NULL)
        return;

    struct Stack* stack = createStack(MAX_SIZE);
    do
    {
        // Move to leftmost node
        while (root)
        {
            // Push root's right child and then root to stack.
            if (root->right)
                push(stack, root->right);
            push(stack, root);

            // Set root as root's left child
            root = root->left;
        }

        // Pop an item from stack and set it as root
        root = pop(stack);

        // If the popped item has a right child and the right child is not
        // processed yet, then make sure right child is processed before root
        if (root->right && peek(stack) == root->right)
        {
            pop(stack); // remove right child from stack
            push(stack, root); // push root back to stack
            root = root->right; // change root so that the right
                                // child is processed next
        }
        else // Else print root's data and set root as NULL
        {
            printf("%d ", root->data);
            root = NULL;
        }
    } while (!isEmpty(stack));
}
```

in this post, iterative postorder traversal is discussed, which is more complex than the other two traversals (due to its nature of non-tail recursion, there is an extra statement after the final recursive call to itself). Postorder traversal can easily be done using two stacks, though. The idea is to push reverse postorder traversal to a stack. Once we have the reversed postorder traversal in a stack, we can just pop all items one by one from the stack and print them; this order of printing will be in postorder because of the LIFO property of stacks. Now the question is, how to get reversed postorder elements in a stack – the second stack is used for this purpose. For example, in the following tree, we need to get 1, 3, 7, 6, 2, 5, 4 in a stack. If take a closer look at this sequence, we can observe that this sequence is very similar to the preorder traversal. The only difference is that the right child is visited before left child, and therefore the

sequence is “root right left” instead of “root left right”. So, we can do something like iterative preorder traversal with the following differences:
a) Instead of printing an item, we push it to a stack.
b) We push the left subtree before the right subtree.
Following is the complete algorithm. After step 2, we get the reverse of a postorder traversal in the second stack. We use the first stack to get the correct order.

1. Push root to first stack.
2. Loop while first stack is not empty
   2.1 Pop a node from first stack and push it to second stack
   2.2 Push left and right children of the popped node to first stack
3. Print contents of second stack

1. Push 1 to first stack.
   First stack: 1
   Second stack: Empty

2. Pop 1 from first stack and push it to second stack.
   Push left and right children of 1 to first stack
   First stack: 2, 3
   Second stack: 1

3. Pop 3 from first stack and push it to second stack.
   Push left and right children of 3 to first stack
   First stack: 2, 6, 7
   Second stack: 1, 3

4. Pop 7 from first stack and push it to second stack.
   First stack: 2, 6
   Second stack: 1, 3, 7

5. Pop 6 from first stack and push it to second stack.
   First stack: 2
   Second stack: 1, 3, 7, 6

6. Pop 2 from first stack and push it to second stack.
   Push left and right children of 2 to first stack
   First stack: 4, 5
   Second stack: 1, 3, 7, 6, 2

7. Pop 5 from first stack and push it to second stack.
   First stack: 4
   Second stack: 1, 3, 7, 6, 2, 5

8. Pop 4 from first stack and push it to second stack.
   First stack: Empty
   Second stack: 1, 3, 7, 6, 2, 5, 4

The algorithm stops here since there are no more items in the first stack.
Observe that the contents of second stack are in postorder fashion

```
// An iterative function to do post order
// traversal of a given binary tree
void postOrderIterative(Node* root)
{
    if (root == NULL)
        return;

    // Create two stacks
    stack<Node *> s1, s2;

    // push root to first stack
    s1.push(root);
    Node* node;

    // Run while first stack is not empty
    while (!s1.empty()) {
        // Pop an item from s1 and push it to s2
        node = s1.top();
        s1.pop();
        s2.push(node);

        // Push left and right children
        // of removed item to s1
        if (node->left)
            s1.push(node->left);
        if (node->right)
            s1.push(node->right);
    }

    // Print all elements of second stack
    while (!s2.empty()) {
        node = s2.top();
        s2.pop();
        cout << node->data << " ";
    }
}
```

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.

```
/* Given a binary tree, print its nodes according to the
"bottom-up" postorder traversal. */
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 << " ";
}
```

Left view of binary tree

Method-1 (Using Recursion)

The left view contains all nodes that are first nodes in their levels. A simple solution is to **do** **level order traversal** and print the first node in every level.

The problem can also be solved **using simple recursive traversal**. We can keep track of the level of a node by passing a parameter to all recursive calls. The idea is to keep track of the maximum level also. Whenever we see a node whose level is more than maximum level so far, we print the node because this is the first node in its level (Note that we traverse the left subtree before right subtree).

```
// Recursive function to print
// left view of a binary tree.
void leftViewUtil(struct Node *root,
                  int level, int *max_level)
{
    // Base Case
    if (root == NULL) return;

    // If this is the first Node of its level
    if (*max_level < level)
    {
        cout << root->data << " ";
        *max_level = level;
    }

    // Recur for left subtree first,
    // then right subtree
      leftViewUtil(root->left, level + 1, max_level);
    leftViewUtil(root->right, level + 1, max_level);

}

// A wrapper over leftViewUtil()
void leftView(struct Node *root)
{
    int max_level = 0;
    leftViewUtil(root, 1, &max_level);
}
```

Time Complexity: The function does a simple traversal of the tree, so the complexity is O(n).
Auxiliary Space: O(n), due to the stack space during recursive call.

Method-2 (Using Queue):

In this method, level order traversal based solution is discussed. If we observe carefully, we will see that our main task is to print the left most node of every level. So, we will do a level order traversal on the tree and print the leftmost node at every level.

```
// function to print left view of
// binary tree
void printLeftView(Node* root)
{
    if (!root)
        return;

    queue<Node*> q;
    q.push(root);

    while (!q.empty())
    {
        // number of nodes at current level
        int n = q.size();

        // Traverse all nodes of current level
        for(int i = 1; i <= n; i++)
        {
            Node* temp = q.front();
            q.pop();

            // Print the left most element
            // at the level
            if (i == 1)
                cout<<temp->data<<" ";

            // Add left node to queue
            if (temp->left != NULL)
                q.push(temp->left);

            // Add right node to queue
            if (temp->right != NULL)
                q.push(temp->right);
        }
    }
}
```

Time Complexity: O(n), where n is the number of nodes in the binary tree.

RIGHT VIEW OF BINARY TREE

The problem can be solved using simple recursive traversal. We can keep track of level of a node by passing a parameter to all recursive calls. The idea is to keep track of maximum level also. And traverse the tree in a manner that right subtree is visited before left subtree. Whenever we see a node whose level is more than maximum level so far, we print the node because this is the last node in its level (Note that we traverse the right subtree before left subtree).

```
// Recursive function to print
// right view of a binary tree.
void rightViewUtil(struct Node *root,
                   int level, int *max_level)
{
    // Base Case
    if (root == NULL) return;

    // If this is the last Node of its level
    if (*max_level < level)
    {
        cout << root->data << "\t";
        *max_level = level;
    }

    // Recur for right subtree first,
    // then left subtree
    rightViewUtil(root->right, level + 1, max_level);
    rightViewUtil(root->left, level + 1, max_level);
}

// A wrapper over rightViewUtil()
void rightView(struct Node *root)
{
    int max_level = 0;
    rightViewUtil(root, 1, &max_level);
}
```

Time Complexity: The function does a simple traversal of the tree, so the complexity is O(n)

Method 2: In this method, level order traversal based solution is discussed. If we observe carefully, we will see that our main task is to print the right most node of every level. So, we will do a level order traversal on the tree and print the last node at every level. Below is the implementation of above approach:

```
// function to print Right view of
// binary tree
void printRightView(Node* root)
{
    if (root == NULL)
        return;

    queue<Node*> q;
    q.push(root);

    while (!q.empty()) {
        // get number of nodes for each level
        int n = q.size();

        // traverse all the nodes of the current level
        while (n--) {
```

```
            Node* x = q.front();
            q.pop();

            // print the last node of each level
            if (n == 0) {
                cout << x->data << " ";
            }
            // if left child is not null push it into the
            // queue
            if (x->left)
                q.push(x->left);
            // if right child is not null push it into the
            // queue
            if (x->right)
                q.push(x->right);
        }
    }
}
```

Time Complexity: O(n), where n is the number of nodes in the binary tree.

```
// function to print right view of
// binary tree
void printRightView(Node* root)
{
    if (!root)
        return;

    queue<Node*> q;
    q.push(root);

    while (!q.empty())
    {
        // number of nodes at current level
        int n = q.size();

        // Traverse all nodes of current level
        for(int i = 1; i <= n; i++)
        {
            Node* temp = q.front();
            q.pop();

            // Print the right most element
            // at the level
            if (i == n)
                cout<<temp->data<<" ";

            // Add left node to queue
            if (temp->left != NULL)
                q.push(temp->left);

            // Add right node to queue
```

```
            if (temp->right != NULL)
                q.push(temp->right);
        }
    }
}
```

Time Complexity: O(n), where n is the number of nodes in the binary tree.

Top view of binary tree

The idea is to do something similar to vertical Order Traversal. Like vertical Order Traversal, we need to put nodes of same horizontal distance together. We do a level order traversal so that the topmost node at a horizontal node is visited before any other node of same horizontal distance below it. Hashing is used to check if a node at given horizontal distance is seen or not.

```
// function should print the topView of
// the binary tree
void topview(Node* root)
{
    if (root == NULL)
        return;
    queue<Node*> q;
    map<int, int> m;
    int hd = 0;
    root->hd = hd;

    // push node and horizontal distance to queue
    q.push(root);

    cout << "The top view of the tree is : \n";

    while (q.size()) {
        hd = root->hd;

        // count function returns 1 if the container
        // contains an element whose key is equivalent
        // to hd, or returns zero otherwise.
        if (m.count(hd) == 0)
            m[hd] = root->data;
        if (root->left) {
            root->left->hd = hd - 1;
            q.push(root->left);
        }
        if (root->right) {
            root->right->hd = hd + 1;
            q.push(root->right);
        }
        q.pop();
        root = q.front();
    }

    for (auto i = m.begin(); i != m.end(); i++) {
        cout << i->second << " ";
    }
}
```

Bottom view of binary tree

Method 1 – Using Queue

The following are steps to print Bottom View of Binary Tree.
1. We put tree nodes in a queue for the level order traversal.
2. Start with the horizontal distance(hd) 0 of the root node, keep on adding left child to queue along with the horizontal distance as hd-1 and right child as hd+1.
3. Also, use a TreeMap which stores key value pair sorted on key.
4. Every time, we encounter a new horizontal distance or an existing horizontal distance put the node data for the horizontal distance as key. For the first time it will add to the map, next time it will replace the value. This will make sure that the bottom most element for that horizontal distance is present in the map and if you see the tree from beneath that you will see that element.

```
/ Method that prints the bottom view.
void bottomView(Node *root)
{
    if (root == NULL)
        return;

    // Initialize a variable 'hd' with 0
    // for the root element.
    int hd = 0;

    // TreeMap which stores key value pair
    // sorted on key value
    map<int, int> m;

    // Queue to store tree nodes in level
    // order traversal
    queue<Node *> q;

    // Assign initialized horizontal distance
    // value to root node and add it to the queue.
    root->hd = hd;
    q.push(root);  // In STL, push() is used enqueue an item

    // Loop until the queue is empty (standard
    // level order loop)
    while (!q.empty())
    {
        Node *temp = q.front();
        q.pop();   // In STL, pop() is used dequeue an item

        // Extract the horizontal distance value
        // from the dequeued tree node.
        hd = temp->hd;

        // Put the dequeued tree node to TreeMap
        // having key as horizontal distance. Every
        // time we find a node having same horizontal
        // distance we need to replace the data in
        // the map.
```

```
        m[hd] = temp->data;

        // If the dequeued node has a left child, add
        // it to the queue with a horizontal distance hd-1.
        if (temp->left != NULL)
        {
            temp->left->hd = hd-1;
            q.push(temp->left);
        }

        // If the dequeued node has a right child, add
        // it to the queue with a horizontal distance
        // hd+1.
        if (temp->right != NULL)
        {
            temp->right->hd = hd+1;
            q.push(temp->right);
        }
    }

    // Traverse the map elements using the iterator.
    for (auto i = m.begin(); i != m.end(); ++i)
        cout << i->second << " ";
}
```

Approach:
Create a map like, map where key is the horizontal distance and value is a pair(a, b) where a is the value of the node and b is the height of the node. Perform a pre-order traversal of the tree. If the current node at a horizontal distance of h is the first we've seen, insert it in the map. Otherwise, compare the node with the existing one in map and if the height of the new node is greater, update in the Map.

```
void printBottomViewUtil(Node * root, int curr, int hd, map <int, pair <int, int>> & m)
{
    // Base case
    if (root == NULL)
        return;

    // If node for a particular
    // horizontal distance is not
    // present, add to the map.
    if (m.find(hd) == m.end())
    {
        m[hd] = make_pair(root -> data, curr);
    }
    // Compare height for already
    // present node at similar horizontal
    // distance
    else
    {
```

```
        pair < int, int > p = m[hd];
        if (p.second <= curr)
        {
            m[hd].second = curr;
            m[hd].first = root -> data;
        }
    }

    // Recur for left subtree
    printBottomViewUtil(root -> left, curr + 1, hd - 1, m);

    // Recur for right subtree
    printBottomViewUtil(root -> right, curr + 1, hd + 1, m);
}

void printBottomView(Node * root)
{

    // Map to store Horizontal Distance,
    // Height and Data.
    map < int, pair < int, int > > m;

    printBottomViewUtil(root, 0, 0, m);
         // Prints the values stored by printBottomViewUtil()

    map < int, pair < int, int > > ::iterator it;
    for (it = m.begin(); it != m.end(); ++it)
    {
        pair < int, int > p = it -> second;
        cout << p.first << " ";
    }
}
```

Zigzag traversal

This problem can be solved using two stacks. Assume the two stacks are current: currentlevel and nextlevel. We would also need a variable to keep track of the current level order(whether it is left to right or right to left). We pop from the currentlevel stack and print the nodes value. Whenever the current level order is from left to right, push the nodes left child, then its right child to the stack nextlevel. Since a stack is a LIFO(Last-In-First_out) structure, next time when nodes are popped off nextlevel, it will be in the reverse order. On the other hand, when the current level order is from right to left, we would push the nodes right child first, then its left child. Finally, do-not forget to swap those two stacks at the end of each level(i.e., when current level is empty)

```
// function to print the zigzag traversal
void zizagtraversal(struct Node* root)
{
    // if null then return
    if (!root)
```

```
        return;

    // declare two stacks
    stack<struct Node*> currentlevel;
    stack<struct Node*> nextlevel;

    // push the root
    currentlevel.push(root);

    // check if stack is empty
    bool lefttoright = true;
    while (!currentlevel.empty()) {

        // pop out of stack
        struct Node* temp = currentlevel.top();
        currentlevel.pop();

        // if not null
        if (temp) {

            // print the data in it
            cout << temp->data << " ";

            // store data according to current
            // order.
            if (lefttoright) {
                if (temp->left)
                    nextlevel.push(temp->left);
                if (temp->right)
                    nextlevel.push(temp->right);
            }
            else {
                if (temp->right)
                    nextlevel.push(temp->right);
                if (temp->left)
                    nextlevel.push(temp->left);
            }
        }

        if (currentlevel.empty()) {
            lefttoright = !lefttoright;
            swap(currentlevel, nextlevel);
        }
    }
}
```

Time Complexity: O(n)
Space Complexity: O(n)+(n)=O(n)

Recursive Approach:The approach used here is the observable similarity to the level order traversal. Here we need to include an extra parameter to keep a track of printing each level in left-right or right-left way.

```
 // Function to calculate height of tree
int treeHeight(Node *root){
    if(!root) return 0;
    int lHeight = treeHeight(root->left);
    int rHeight = treeHeight(root->right);
    return max(lHeight, rHeight) + 1;
}

// Helper Function to store the zig zag order traversal
// of tree in a list recursively
void zigZagTraversalRecursion(Node* root, int height, bool lor, vector<int> &ans){
    // Height = 1 means the tree now has only one node
    if(height <= 1){
        if(root) ans.push_back(root->data);
    }
    // When Height > 1
    else{
        if(lor){
            if(root->left) zigZagTraversalRecursion(root->left, height - 1, lor, ans);
            if(root->right) zigZagTraversalRecursion(root->right, height - 1, lor, ans);
        }
        else{
            if(root->right) zigZagTraversalRecursion(root->right, height - 1, lor, ans);
            if(root->left) zigZagTraversalRecursion(root->left, height - 1, lor, ans);
        }
    }
}

// Function to traverse tree in zig zag order
vector <int> zigZagTraversal(Node* root)
{
    vector<int> ans;
    bool leftOrRight = true;
    int height = treeHeight(root);
    for(int i = 1; i <= height; i++){
        zigZagTraversalRecursion(root, i, leftOrRight, ans);
        leftOrRight = !leftOrRight;
    }
    return ans;
}
```

In this approach, use a deque to solve the problem. Push and pop in different ways depending on if the level is odd or level is even.

```
/ Function to print the zigzag traversal

vector<int> zigZagTraversal(Node* root)
{
    deque<Node*> q;
    vector<int> v;
    q.push_back(root);
    v.push_back(root->data);
    Node* temp;
```

```
    // set initial level as 1, because root is
    // already been taken care of.
    int l = 1;

    while (!q.empty()) {
        int n = q.size();

        for (int i = 0; i < n; i++) {

            // popping mechanism
            if (l % 2 == 0) {
                temp = q.back();
                q.pop_back();
            }
            else {
                temp = q.front();
                q.pop_front();
            }

            // pushing mechanism
            if (l % 2 != 0) {

                if (temp->right) {
                    q.push_back(temp->right);
                    v.push_back(temp->right->data);
                }
                if (temp->left) {
                    q.push_back(temp->left);
                    v.push_back(temp->left->data);
                }
            }
            else if (l % 2 == 0) {

                if (temp->left) {
                    q.push_front(temp->left);
                    v.push_back(temp->left->data);
                }
                if (temp->right) {
                    q.push_front(temp->right);
                    v.push_back(temp->right->data);
                }
            }
        }
        l++; // level plus one
    }
    return v;
}
```

Diagonal of binary tree

The idea is to use a map. We use different slope distances and use them as key in the map. Value in the map is a vector (or dynamic array) of nodes. We traverse the tree to store values in the map. Once map is built, we print the contents of it.

```
/* root - root of the binary tree
   d -  distance of current line from rightmost
        -topmost slope.
   diagonalPrint - multimap to store Diagonal
                elements (Passed by Reference) */
void diagonalPrintUtil(Node* root, int d,
             map<int, vector<int>> &diagonalPrint)
{
    // Base case
    if (!root)
        return;

    // Store all nodes of same
    // line together as a vector
    diagonalPrint[d].push_back(root->data);

    // Increase the vertical
    // distance if left child
    diagonalPrintUtil(root->left,
                  d + 1, diagonalPrint);

    // Vertical distance remains
    // same for right child
    diagonalPrintUtil(root->right,
                    d, diagonalPrint);
}

// Print diagonal traversal
// of given binary tree
void diagonalPrint(Node* root)
{

    // create a map of vectors
    // to store Diagonal elements
    map<int, vector<int> > diagonalPrint;
    diagonalPrintUtil(root, 0, diagonalPrint);

    cout << "Diagonal Traversal of binary tree : \n";
    for (auto it :diagonalPrint)
    {
        vector<int> v=it.second;
        for(auto it:v)
          cout<<it<<" ";
        cout<<endl;
    }
}
```

The **time complexity** of this solution is **O(N logN)** and the space complexity is **O(N)**

Approach 2 : Using Queue.

Every node will help to generate the next diagonal. We will push the element in the queue only when its left is available. We will process the node and move to its right.

```
vector <vector <int>> result;
void diagonalPrint(Node* root)
{
   if(root == NULL)
      return;

   queue <Node*> q;
   q.push(root);

   while(!q.empty())
   {
      int size = q.size();
      vector <int> answer;

      while(size--)
      {
         Node* temp = q.front();
         q.pop();

         // traversing each component;
         while(temp)
         {
            answer.push_back(temp->data);

            if(temp->left)
               q.push(temp->left);

            temp = temp->right;
         }
      }
      result.push_back(answer);
   }
}
```

Time Complexity: O(N), because at a node will be traversed 2 times. hence O(2N) == O(N).

Space Complexity: O(N), because we are using queue data structure.

The idea is to use a queue to store only the left child of current node. After printing the data of current node make the current node to its right child, if present.
A delimiter NULL is used to mark the starting of next diagonal.

```
// Iterative function to print diagonal view
void diagonalPrint(Node* root)
{
   // base case
   if (root == NULL)
      return;

   // inbuilt queue of Treenode
   queue<Node*> q;

   // push root
   q.push(root);
```

```
    // push delimiter
    q.push(NULL);

    while (!q.empty()) {
        Node* temp = q.front();
        q.pop();

        // if current is delimiter then insert another
        // for next diagonal and cout nextline
        if (temp == NULL) {

            // if queue is empty return
            if (q.empty())
                return;

            // output nextline
            cout << endl;

            // push delimiter again
            q.push(NULL);
        }
        else {
            while (temp) {
                cout << temp->data << " ";

                // if left child is present
                // push into queue
                if (temp->left)
                    q.push(temp->left);

                // current equals to right child
                temp = temp->right;
            }
        }
    }
}
```

Boundary traversal

Given a binary tree, print boundary nodes of the binary tree Anti-Clockwise starting from the root. The boundary includes left boundary, leaves, and right boundary in order without duplicate nodes. (The values of the nodes may still be duplicates.)
The left boundary is defined as the path from the root to the left-most node. The right boundary is defined as the path from the root to the right-most node. If the root doesn't have left subtree or right subtree, then the root itself is left boundary or right boundary. Note this definition only applies to the input binary tree, and not apply to any subtrees.
The left-most node is defined as a leaf node you could reach when you always firstly travel to the left subtree if it exists. If not, travel to the right subtree. Repeat until you reach a leaf node. The right-most node is also defined in the same way with left and right exchanged.

We break the problem in 3 parts:
1. Print the left boundary in top-down manner.
2. Print all leaf nodes from left to right, which can again be sub-divided into two sub-parts:
…..2.1 Print all leaf nodes of left sub-tree from left to right.
…..2.2 Print all leaf nodes of right subtree from left to right.
3. Print the right boundary in bottom-up manner.
We need to take care of one thing that nodes are not printed again. e.g. The left most node is also the leaf node of the tree.

```
/ A simple function to print leaf nodes of a binary tree
void printLeaves(struct node* root)
{
    if (root == NULL)
        return;

    printLeaves(root->left);

    // Print it if it is a leaf node
    if (!(root->left) && !(root->right))
        printf("%d ", root->data);

    printLeaves(root->right);
}

// A function to print all left boundary nodes, except a leaf node.
// Print the nodes in TOP DOWN manner
void printBoundaryLeft(struct node* root)
{
    if (root == NULL)
        return;

    if (root->left) {

        // to ensure top down order, print the node
        // before calling itself for left subtree
        printf("%d ", root->data);
        printBoundaryLeft(root->left);
    }
    else if (root->right) {
        printf("%d ", root->data);
        printBoundaryLeft(root->right);
    }
    // do nothing if it is a leaf node, this way we avoid
    // duplicates in output
}

// A function to print all right boundary nodes, except a leaf node
// Print the nodes in BOTTOM UP manner
void printBoundaryRight(struct node* root)
{
    if (root == NULL)
        return;
```

```
    if (root->right) {
        // to ensure bottom up order, first call for right
        // subtree, then print this node
        printBoundaryRight(root->right);
        printf("%d ", root->data);
    }
    else if (root->left) {
        printBoundaryRight(root->left);
        printf("%d ", root->data);
    }
    // do nothing if it is a leaf node, this way we avoid
    // duplicates in output
}

// A function to do boundary traversal of a given  binary tree
void printBoundary(struct node* root)
{
    if (root == NULL)
        return;

    printf("%d ", root->data);

    // Print the left boundary in top-down manner.
    printBoundaryLeft(root->left);

    // Print all leaf nodes
    printLeaves(root->left);
    printLeaves(root->right);

    // Print the right boundary in bottom-up manner
    printBoundaryRight(root->right);
}
```

Time Complexity: O(n) where n is the number of nodes in binary tree.

Check if tree is balanced or not

A tree where no leaf is much farther away from the root than any other leaf. Different balancing schemes allow different definitions of “much farther” and different amounts of work to keep them balanced.

Consider a height-balancing scheme where following conditions should be checked to determine if a binary tree is balanced.

An empty tree is height-balanced. A non-empty binary tree T is balanced if:

1) Left subtree of T is balanced

2) Right subtree of T is balanced

3) The difference between heights of left subtree and right subtree is not more than 1.

The above height-balancing scheme is used in AVL trees. The diagram below shows two trees, one of them is height-balanced and other is not. The second tree is not height-balanced because height of left subtree is 2 more than height of right subtree.

To check if a tree is height-balanced, get the height of left and right subtrees. Return true if difference between heights is not more than 1 and left and right subtrees are balanced, otherwise return false.

```
 /* Returns true if binary tree
with root as root is height-balanced */

bool isBalanced(node* root)
{
   int lh; /* for height of left subtree */
   int rh; /* for height of right subtree */

   /* If tree is empty then return true */
   if (root == NULL)
       return 1;

   /* Get the height of left and right sub trees */
   lh = height(root->left);
   rh = height(root->right);

   if (abs(lh - rh) <= 1 && isBalanced(root->left) && isBalanced(root->right))
       return 1;

   /* If we reach here then
   tree is not height-balanced */
   return 0;
}

/* UTILITY FUNCTIONS TO TEST isBalanced() FUNCTION */

/* returns maximum of two integers */
int max(int a, int b)
{
   return (a >= b) ? a : b;
}

/* The function Compute the "height"
of a tree. Height is the number of
nodes along the longest path from
the root node down to the farthest leaf node.*/

int height(node* node)
{
   /* base case tree is empty */
   if (node == NULL)
```

```
    return 0;

  /* If tree is not empty then
  height = 1 + max of left
     height and right heights */
  return 1 + max(height(node->left),
                 height(node->right));
}
```

Time Complexity: O(n^2) in case of full binary tree.
Optimized implementation: Above implementation can be optimized by calculating the height in the same recursion rather than calling a height() function separately.

```
/* The function returns true if root is
balanced else false The second parameter
is to store the height of tree. Initially,
we need to pass a pointer to a location with
value as 0. We can also write a wrapper
over this function */

bool isBalanced(node* root, int* height)
{

  /* lh --> Height of left subtree
  rh --> Height of right subtree */
  int lh = 0, rh = 0;

  /* l will be true if left subtree is balanced
  and r will be true if right subtree is balanced */
  int l = 0, r = 0;

  if (root == NULL) {
     *height = 0;
     return 1;
  }

  /* Get the heights of left and right subtrees in lh and rh
  And store the returned values in l and r */
  l = isBalanced(root->left, &lh);
  r = isBalanced(root->right, &rh);

  /* Height of current node is max of heights of left and
  right subtrees plus 1*/
  *height = (lh > rh ? lh : rh) + 1;

  /* If difference between heights of left and right
  subtrees is more than 2 then this node is not balanced
  so return 0 */
  if (abs(lh - rh) >= 2)
     return 0;
```

```
    /* If this node is balanced and left and right subtrees
    are balanced then return true */
    else
        return l && r;
}
```

Time Complexity: O(n)
