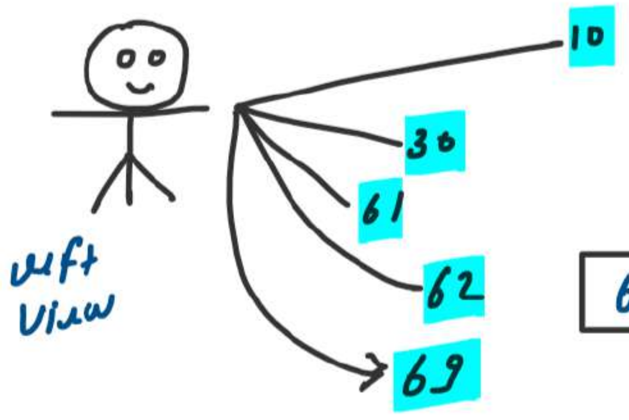
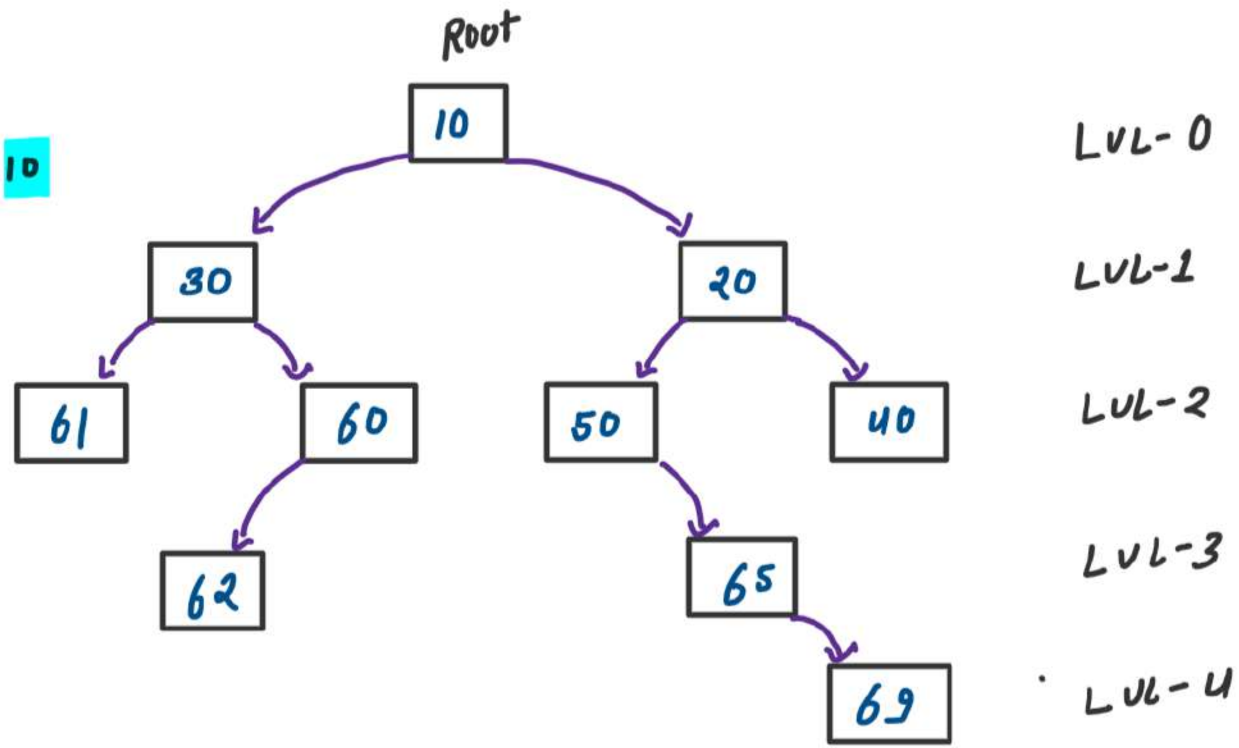
4/12/2023

BINARY TREE
CLASS - 3

1. Left View of Binary Tree

Ex1

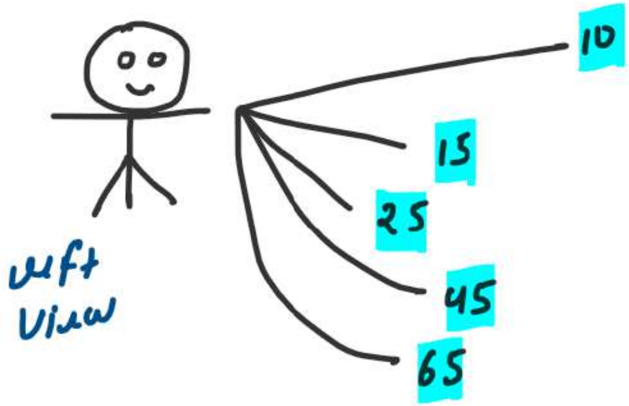


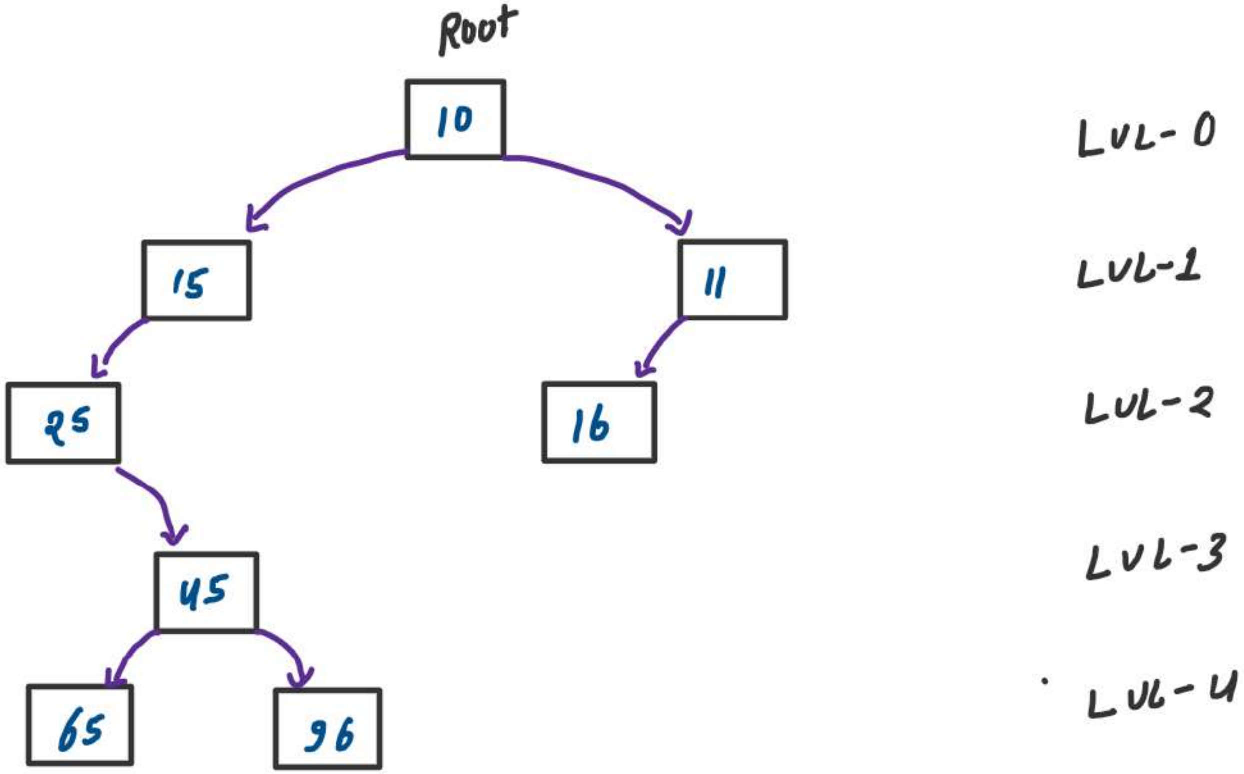


Output

10	30	61	62	69

Ex2





Output

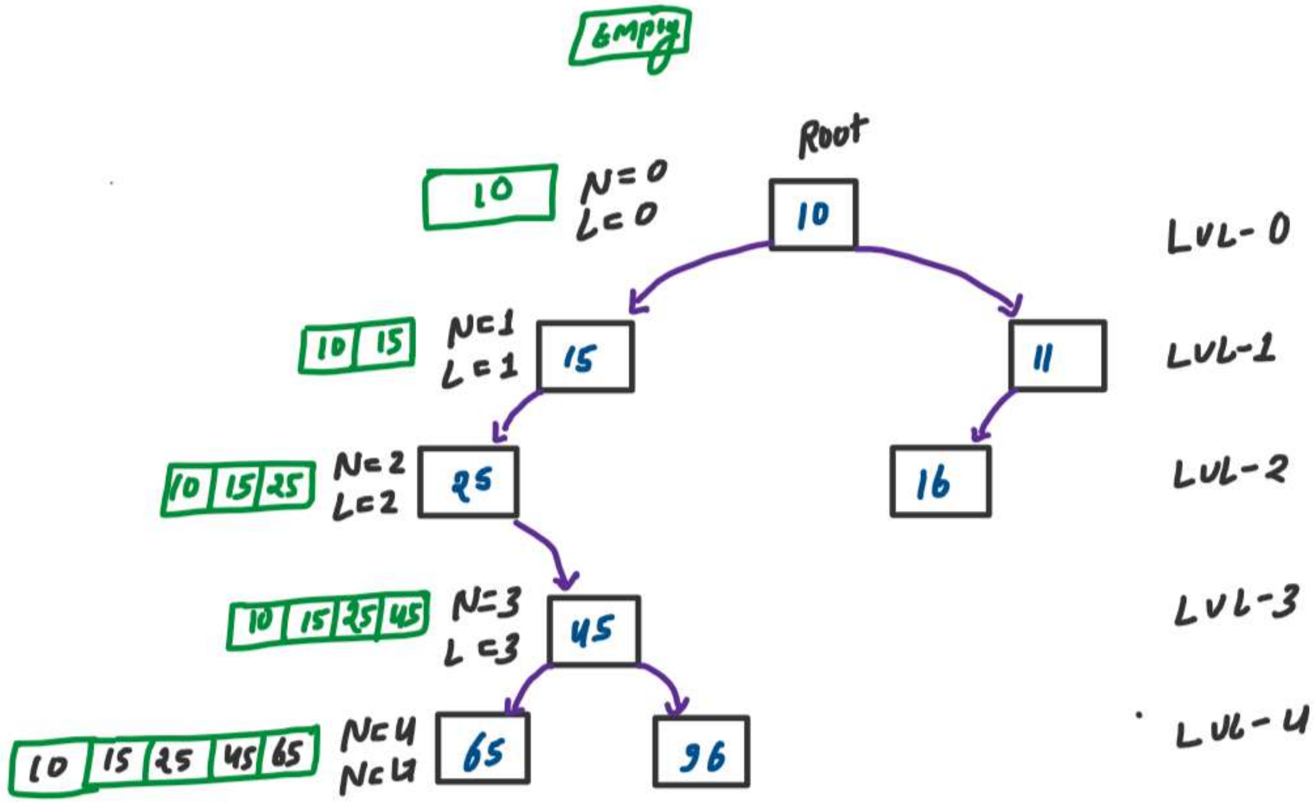
10	15	25	45	65

Logic Building

vector<int> leftView

0	1	2	3	4
10	15	25	45	65

N = leftview.size()

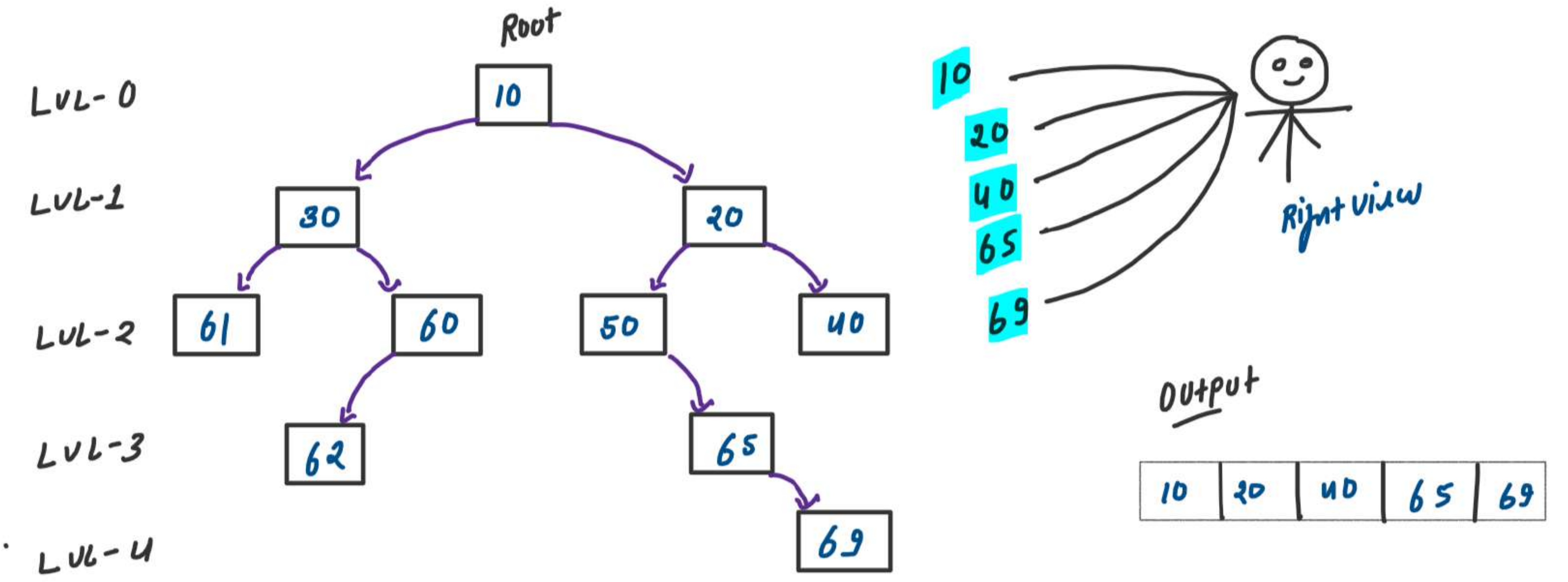


```cpp
// PROBLEM 01: Left view of binary tree
void printLeftView(Node* root, int level, vector<int> &leftView){
    // Base case
    if(root == NULL){
        return;
    }

    // 1 case hum solve kar lenge
    if(level == leftView.size()){
        leftView.push_back(root->data);
    }

    // Ab recursion solve kar lega
    printLeftView(root->left, level+1, leftView);
    printLeftView(root->right, level+1, leftView);
}

/*
Binary Tree Input: 10 15 25 -1 45 65 -1 -1 96 -1 -1 -1 11 16 -1 -1 -1

OUTPUT:
Left view:
10 15 25 45 65
*/
```

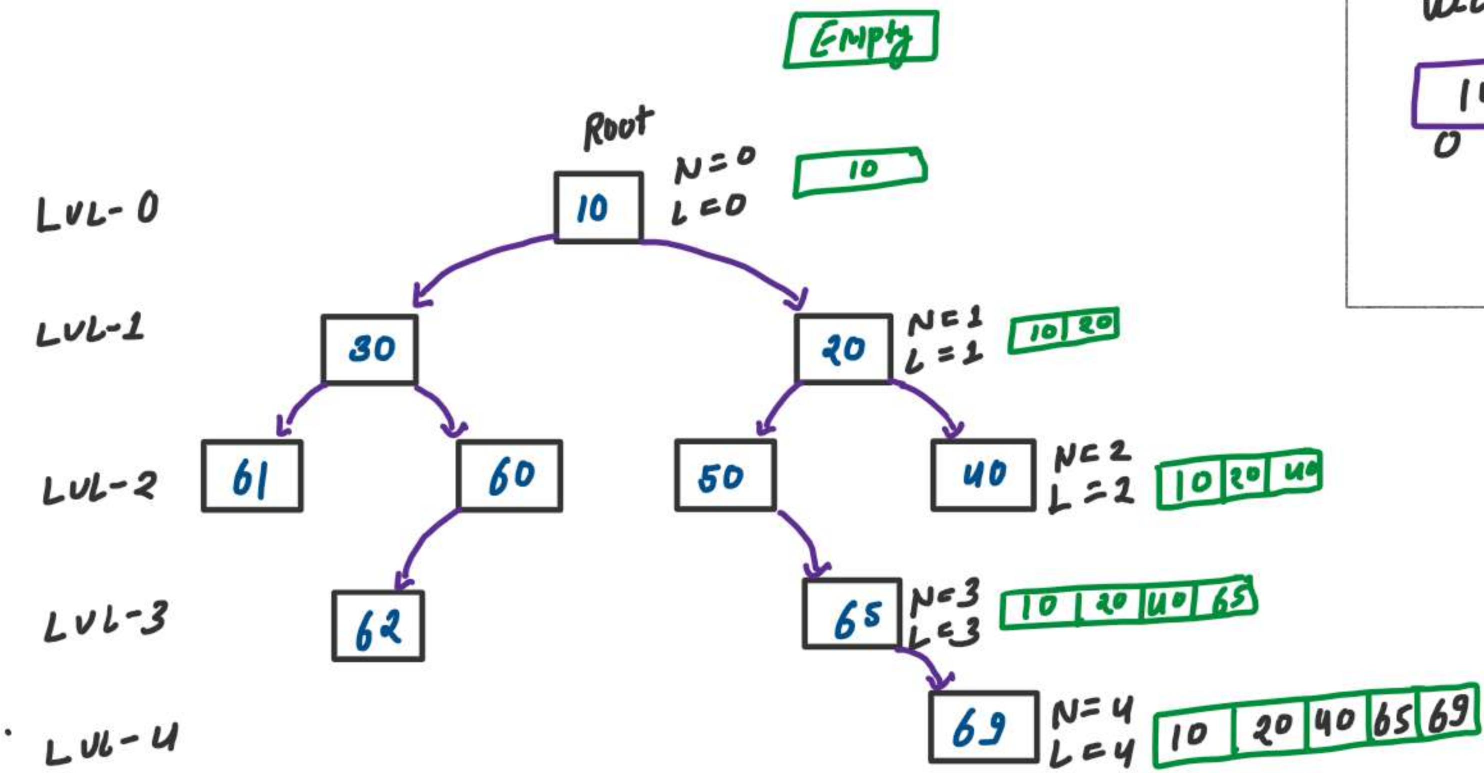
Time Complexity: O(N),
where N is total number of nodes in binary tree

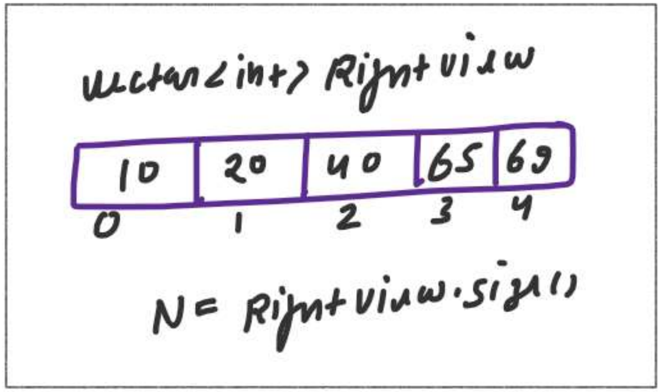
Space Complexity: O(L),
where L is maximum number of nodes in the level of binary tree

2. Right View of Binary Tree



Logic Building



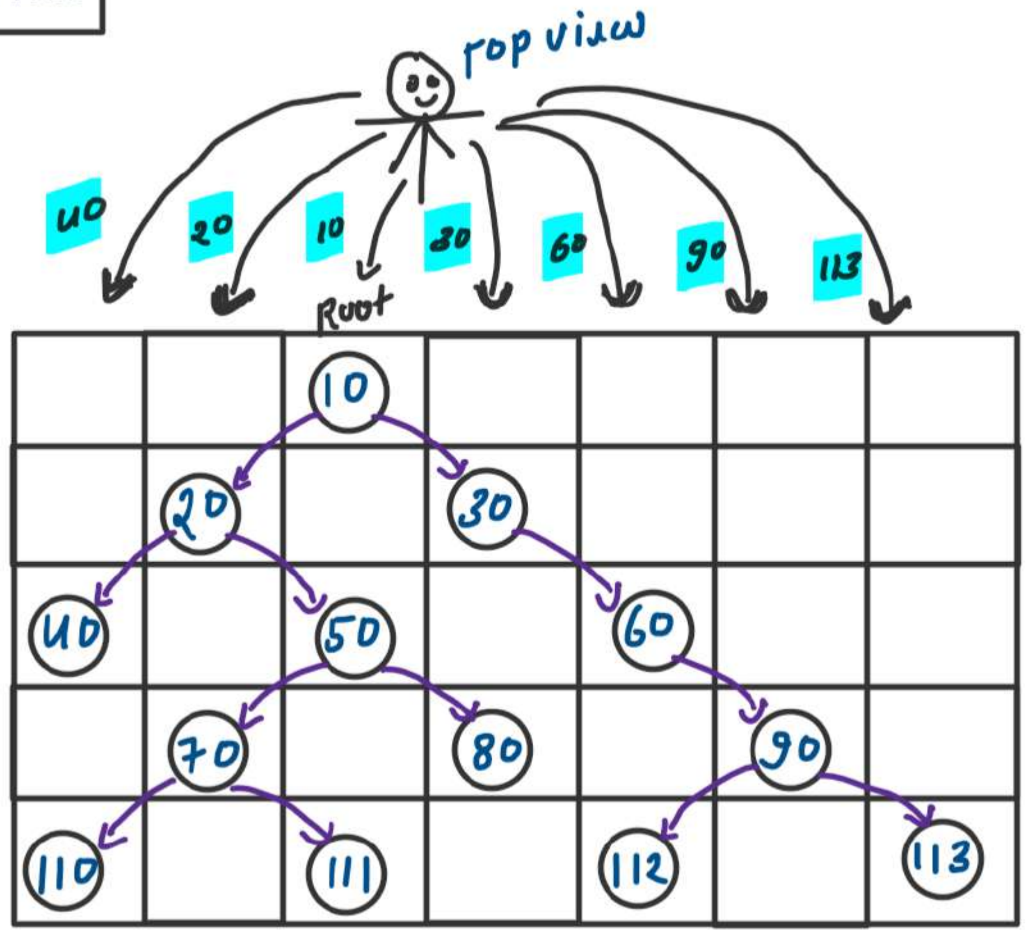


```cpp
// PROBLEM 02: Right view of binary tree
void printRightView(Node* root, int level, vector<int> &rightView){
    // Base case
    if(root == NULL){
        return;
    }

    // 1 case hum solve kar lenge
    if(level == rightView.size()){
        rightView.push_back(root->data);
    }

    // Ab recursion solve kar lega
    printRightView(root->right, level+1, rightView);
    printRightView(root->left, level+1, rightView);
}

/*
Binary Tree Input:
10 30 61 -1 -1 60 62 -1 -1 -1 20 50 -1 65 -1 69 -1 -1 40 -1 -1

OUTPUT:
Right view:
10 20 40 65 69
*/
```

Time Complexity: O(N),
where N is total number of nodes in binary tree

Space Complexity: O(L),
where L is maximum number of nodes in the level of binary tree

3. Top View of Binary Tree

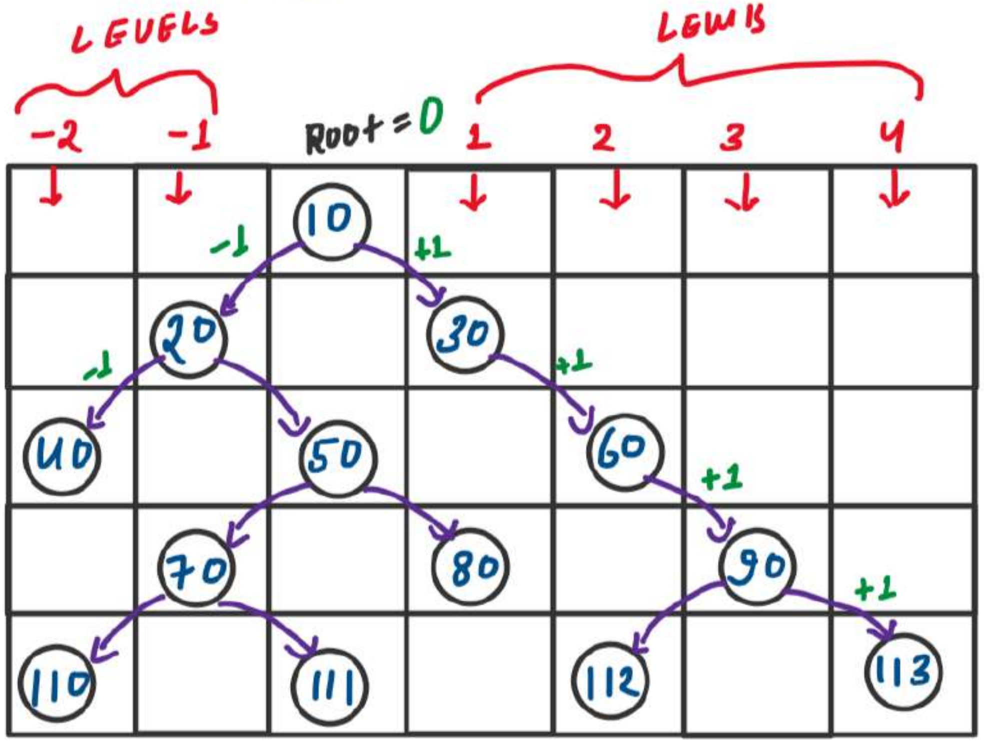


Output

40	20	10	30	60	90	113

Logic Building

VERTICAL LEVEL



Why use map
↳ Ans: To Print The Top view Node data According to Horizontal distance wise of Node in order

List

-2	-1	0	1	2	3	4	→ LEVEL
40	20	10	30	60	90	113	→ Node Data

map

MAP

Node → Data	Horizontal Distance To Node

LEVEL

QUEUE

pair

Node*	LEVEL

Map<int, int> hdToNodeMp;

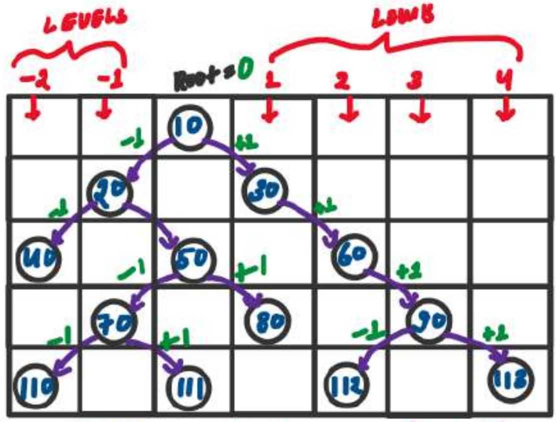
Queue< pair< Node*, int> > q;

Initially
↳ q.push(make_pair(root, 0));

DRY RUN

map

Data	LEVEL
40	-2
20	-1
10	0
30	+1
60	2
90	3
113	4



	Front	FRONT	FRONT	FRONT	FRONT	FRONT	FRONT	FRONT	FRONT	FRONT	FRONT	FRONT	FRONT
Node / queue	10	20	30	40	50	60	70	80	90	110	111	112	113
Level	0	-1	1	-2	0	2	-1	1	3	-2	0	2	4

0-1 | 0+1 | -1-1 | -1+1 | 1+1 | 0-1 | 0+1 | 2+1 | -1-1 | -1+1 | 3-1 | 3+1

Now queue is Empty — STOP

STEP1 Store Node and level into queue and pop

STEP2 Fetch level from queue and store data and level into map when when unique level will occur.

```cpp
// PROBLEM 03: Top view of binary tree

void printTopView(Node* root){
    map<int, int> hdToNodeMap; // < level, data >
    queue<pair<Node*, int>> q;
    // Initialy store the root node and level 0 into queue
    q.push(make_pair(root,0));

    while(!q.empty()){
        // Fetch front from queue and pop
        pair<Node*, int> front = q.front();
        q.pop();

        Node* frontNode = front.first;
        int level = front.second;

        // Store frontNode->data and level into the map when unique level will occur
        if(hdToNodeMap.find(level) == hdToNodeMap.end()){
            hdToNodeMap[level] = frontNode->data;
        }

        // Agar root ka left node exist krta hai to queue me push krdo with level-1
        if(frontNode->left != NULL){
            q.push(make_pair(frontNode->left, level-1));
        }
        // Agar root ka right node exist krta hai to queue me push krdo with level+1
        if(frontNode->right != NULL){
            q.push(make_pair(frontNode->right, level+1));
        }
    }

    cout<< "Printing Top View: " << endl;
    for(auto data: hdToNodeMap){
        cout<< data.second << " ";
    }
}

/*
Binary Tree Input:
10 20 40 -1 -1 50 70 110 -1 -1 111 -1 -1 80 -1 -1 30 -1 60 -1 9 112 -1 -1 113 -1 -1

OUTPUT:
10
20 30
40 50 60
70 80 9
110 111 112 113
Printing Top View:
40 20 10 30 60 9 113
*/
```

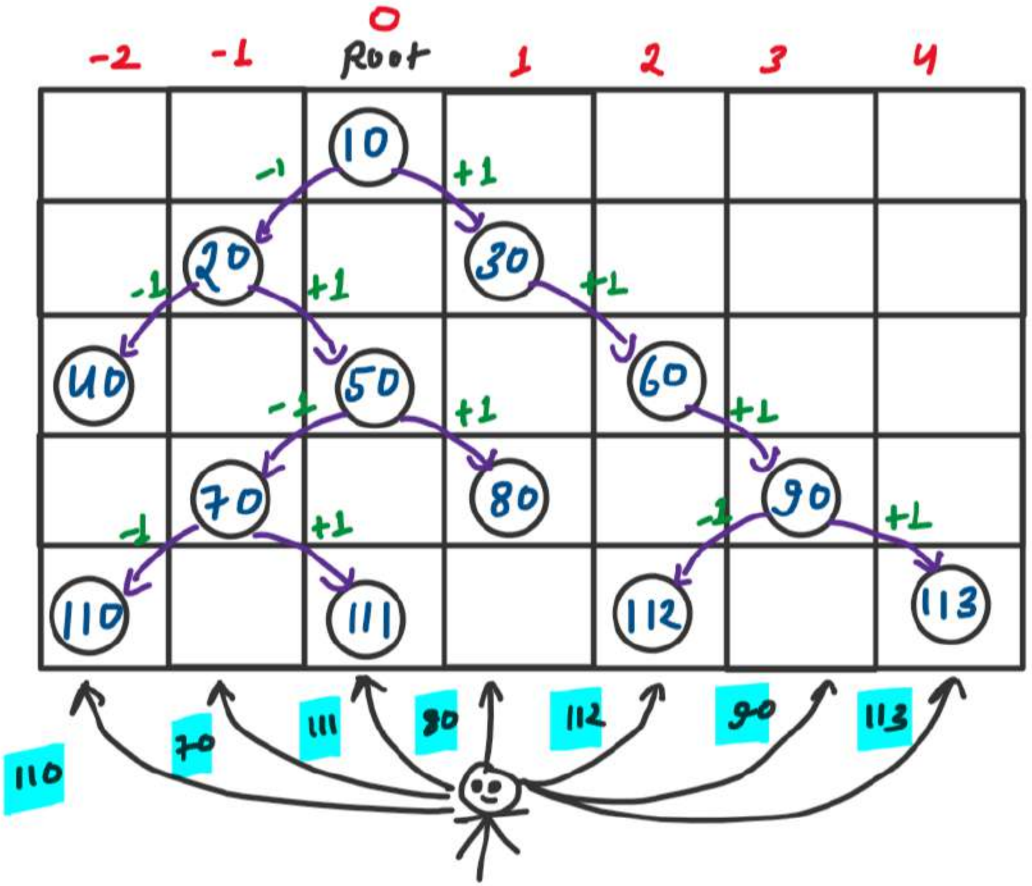
Time Complexity: O(N), *where N is total number of nodes in binary tree*

Space Complexity: O(N), *where*

Case I - *considering a skewed tree:*
space complexity is O(N)

Case II - *considering now skewed tree:*
space complexity is O(W), Where W is maximum width of the tree

4. Bottom View of Binary Tree



Output

110	70	111	80	112	90	113

STEP1 Start Node and level into queue and pop

STEP2 Fetch level from queue and store data and level into map

Overwrite the Exist Entry of map jab same Entry occurs ho rahi ho.

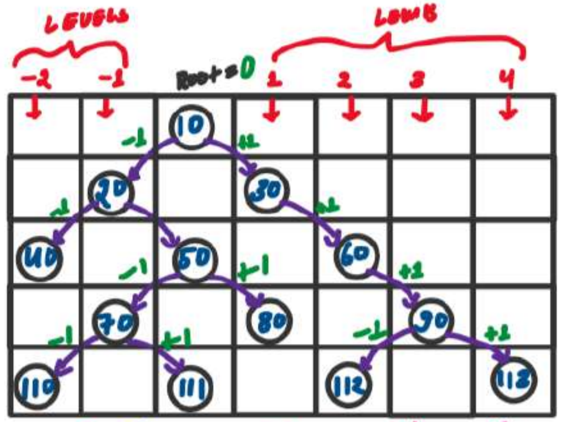
DRY RUN

map

Data	LEVEL
110	-2
70	-1
111	0
80	+1
112	2
90	3
113	4

second

First



Node

Queue

Level

Front	FRONT	FRONT	FRONT	FRONT	FRONT	FRONT	FRONT	FRONT	FRONT	Front	Front	Front
10	20	30	40	50	60	70	80	90	110	111	112	113
0	-1	1	-2	0	2	-1	1	3	-2	0	2	4

0-1   0+1   -1-1   -1+1   1+1   0-1   0+1   2+1   -1-1   -1+1   3-1   3+1

Now queue is Empty—STOP

output [110 70 111 80 112 90 113]

```cpp
// PROBLEM 04: Bottom view of binary tree

void printBottomView(Node* root){
    map<int, int> hdToNodeMap; // < level, data >
    queue<pair<Node*, int>> q;
    // Initialy store the root node and level 0 into queue
    q.push(make_pair(root,0));

    while(!q.empty()){
        // Fetch front from queue and pop
        pair<Node*, int> front = q.front();
        q.pop();

        Node* frontNode = front.first;
        int level = front.second;

        // OVERWRITE: Store frontNode->data and level into the map
        hdToNodeMap[level] = frontNode->data;

        // Agar root ka left node exist krta hai to queue me push krdo with level-1
        if(frontNode->left != NULL){
            q.push(make_pair(frontNode->left, level-1));
        }
        // Agar root ka right node exist krta hai to queue me push krdo with level+1
        if(frontNode->right != NULL){
            q.push(make_pair(frontNode->right, level+1));
        }
    }

    cout<< "Printing Bottom View: " << endl;
    for(auto data: hdToNodeMap){
        cout<< data.second << " ";
    }
}

/*
Binary Tree Input:
10 20 40 -1 -1 50 70 110 -1 -1 111 -1 -1 80 -1 -1 30 -1 60 -1 9 112 -1 -1 113 -1 -1

OUTPUT:
10
20 30
40 50 60
70 80 9
110 111 112 113
Printing Bottom View:
110 70 111 80 112 9 113
*/
```

Time Complexity: O(N), *where N is total number of nodes in binary tree*

Space Complexity: O(N), *where*

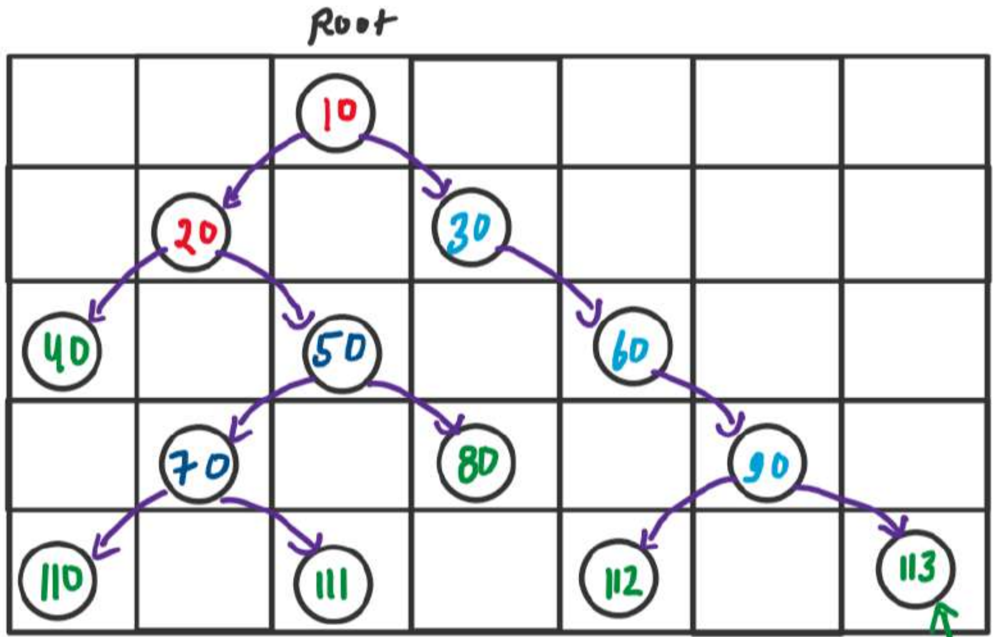
Case I - *considering a skewed tree:*
space complexity is O(N)

Case II - *considering now skewed tree:*
space complexity is O(W), Where W is maximum width of the tree

📂 5. Boundary Traversal of Binary Tree

Example:01

First left Leaf Node



First Right Leaf Node

Output

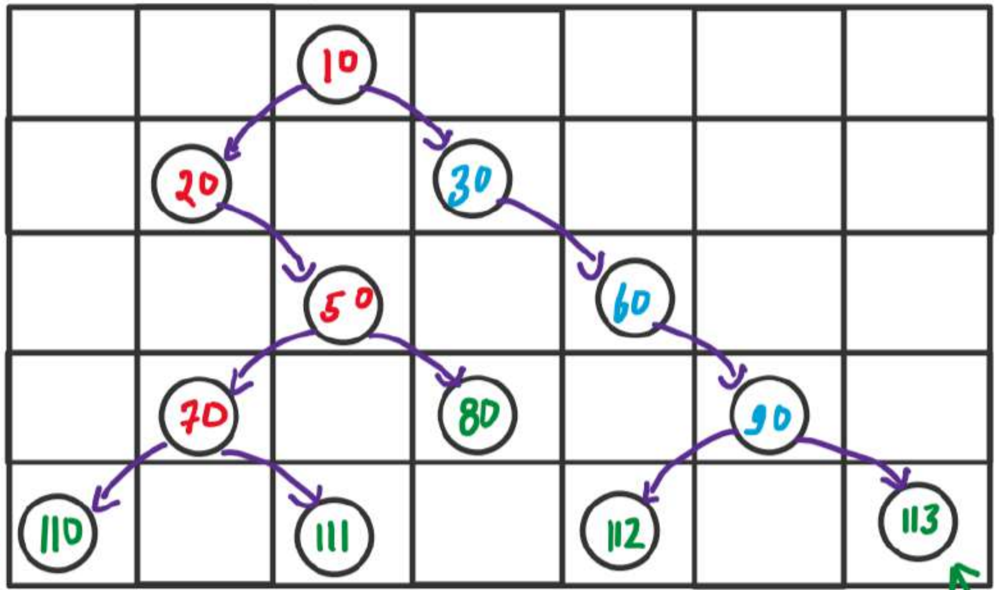
10 20 40 110 111 80
112 113 90 60 30

Print A ⇒ Left Node Boundary
Print B ⇒ Leaf Node Boundary
Print C ⇒ Right Node Boundary

Print (A) Print All Node before left Node
Jo Ek Leaf Node Hoga

Example:02

Root



First Left Leaf Node

First Right Leaf Node

Output

10	20	50	70	110	111
80	112	113	90	60	30

Point A ⇒ Left Node Boundary

Point B ⇒ Leaf Node Boundary

Point C ⇒ Right Node Boundary

Point A Print All Node before left Node
Jo Ek Leaf Node Hoga

FUNCTION (A)

```
void PrintLeftBoundary(Node* Root){
    if(root == Null) return;
    if(root->left == Null &&
       root->right == Null)
        return;
    cout << root->data << " ";
    if(root->left != Null)
        PrintLeftBoundary(root->left);
    Elseif(Root->right != Null)
        PrintLeftBoundary(root->Right);
}
```

First left leaf Node AANE par Function se Bahar Ho jaao---

Ex1 par DRY RUN Karo

Ex2 par DRY RUN Karo

o/p $\boxed{10 \quad 20}$ → EX1

o/p $\boxed{10 \quad 20 \quad 50 \quad 70}$ → EX2

FUNCTION B

```
void printLeafBoundary(Node* root){

    if(root == NULL) return;
    if(root->left == NULL &&
        root->right == NULL) {

        cout << root->data << " ";
    }
    printLeafBoundary(root->left);
    printLeafBoundary(root->right);
}
```

Jab Node leaf Hai Tabhi print Karna Hai

O/P | 40 110 111 80 112 113 | → EX 1

O/P | 110 111 80 112 113 | → EX 2

FUNCTION (C)

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

    if( root->left == NULL &&
        root->right == NULL ) {
            return;
    }

    if( root->right != NULL)
        printRightBoundary( root->right);
    Else if( root->left != NULL)
        print RightBoundary( root->left);

    cout << Root->data << " ";
}
```

First Right leaf Node AANE par Function se Bahar Ho jaao....

O/P 90 60 30 → EX1

O/P 90 60 30 → EX2

Function Boundary Traversal

```
void boundaryTraversal(Node* Root){
    if(root == NULL) return;
    PrintLeftBoundary(Root);
    PrintLeafBoundary(Root);
    PrintRightBoundary(Root);
}
```

WRONG → Output **Ex1**

10 20 40 110 111 80
112 113 90 60 30 10 X

WRONG → Output **Ex2**

10 20 50 70 110 111
80 112 113 90 60 30
10 X

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

    PrintLeftBoundary(Root);
    PrintLeafBoundary(Root);
    if(root->right != NULL){
        PrintRightBoundary(Root->right);
    }
    Else{
        PrintRightBoundary(Root->left);
    }
}
```

Correct → Output

Ex1

10 20 40 110 111 80

112 113 90 60 30

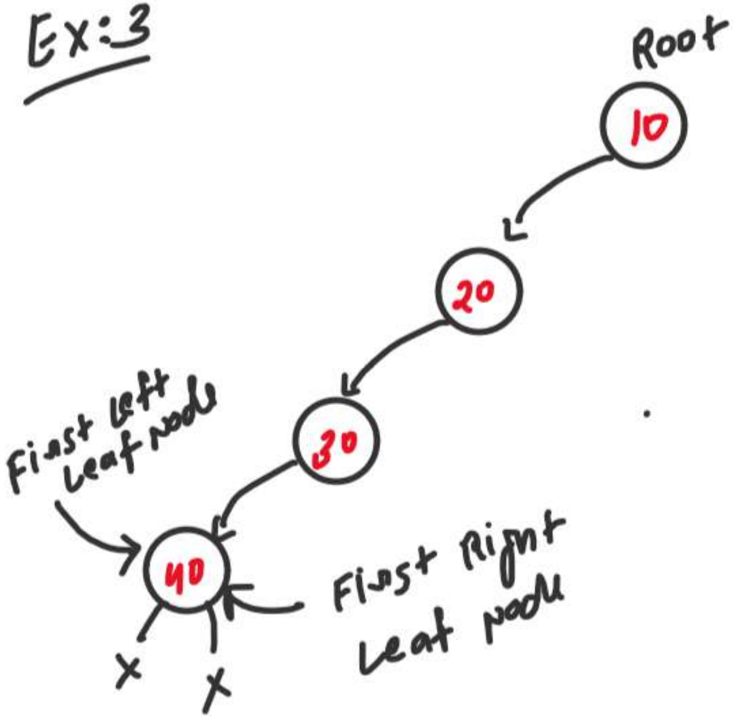
Correct → Output

Ex2

10 20 50 70 110 111

80 112 113 90 60 30

Ex:3



Left ⇒ 10 20 30

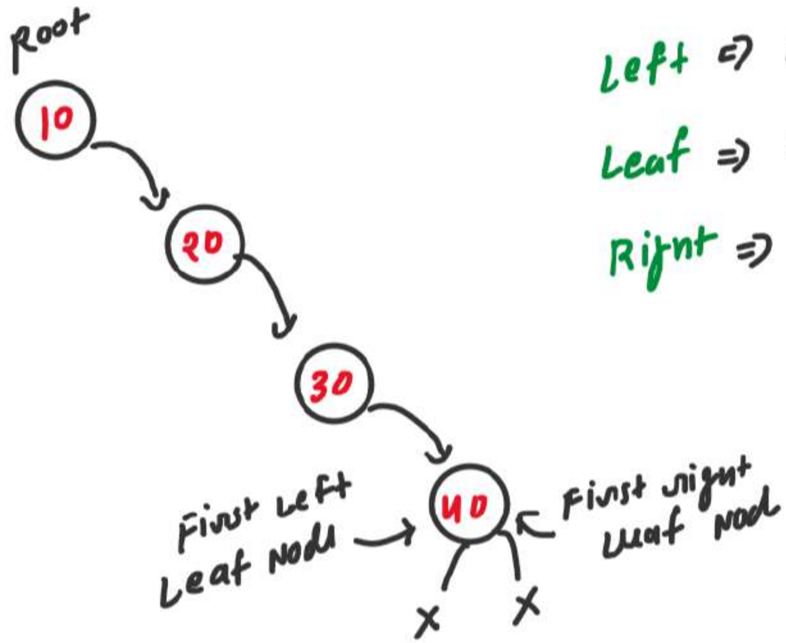
Leaf ⇒ 40

Right ⇒ 30 20

Output

10 20 30 40 30 20

Ex = 4



Left ⇒ 10 20 30

Leaf ⇒ 40

Right ⇒ 30 20

Output

10 20 30 40 30 20

A

```cpp
void printLeftBoundary(Node* root){
    // Base Case
    if(root == NULL){
        return;
    }

    // First Left Leaf Node aane par function se bahar ho jaao
    if(root->left == NULL && root->right == NULL){
        return;
    }

    cout<< root->data << " ";

    if(root->left != NULL){
        printLeftBoundary(root->left);
    }
    else if(root->right != NULL){
        printLeftBoundary(root->right);
    }
}
```

B

```cpp
void printLeafBoundary(Node* root){
    // Base case
    if(root == NULL){
        return;
    }

    // Jab-2 leaf node ayega tabhi print karna hai
    if(root->left == NULL && root->right == NULL){
        cout<< root->data << " ";
    }

    printLeafBoundary(root->left);
    printLeafBoundary(root->right);
}
```

C

```cpp
void printRightBoundary(Node* root){
    // Base case
    if(root == NULL){
        return;
    }

    // Jab first right leaf node aa jaye to function se bahar ho jaao
    if(root->left == NULL && root->right == NULL){
        return;
    }

    if(root->right != NULL){
        printRightBoundary(root->right);
    }
    else if(root->left != NULL){
        printRightBoundary(root->left);
    }

    cout<< root->data << " ";
}
```

```cpp
void boundaryTraversal(Node* root){
    if(root == NULL){
        return;
    }

    printLeftBoundary(root);
    printLeafBoundary(root);
    if(root->right != NULL){
        printRightBoundary(root->right);
    }
    else if(root->left != NULL){
        printRightBoundary(root->left);
    }
}
```

A

B

C

Time Complexity *and* ***Space Complexity: O(N),***

Where N is number of nodes in binary tree