Boundary of Binary Tree

Given a binary tree, return the values of its boundary in**anti-clockwise** direction starting from root. Boundary includes left boundary, leaves, and right boundary in order without duplicate nodes.

Left boundary is defined as the path from root to the**left-most** node. **Right boundary** is defined as the path from 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 applies 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 exists. If not, travel to the right subtree. Repeat until you reach a leaf node.

The **right-most** node is also defined by the same way with left and right exchanged.

Example 1

```
Input:
  1
   \
    2
   / \
  3   4

Ouput:
[1, 3, 4, 2]

Explanation:
The root doesn't have left subtree, so the root itself is left boundary.
The leaves are node 3 and 4.
The right boundary are node 1,2,4. Note the anti-clockwise direction means you shou
ld output reversed right boundary.
So order them in anti-clockwise without duplicates and we have [1,3,4,2].
```

Example 2

Input:
```
    ____1_____
   /          \
  2            3
 / \          /
4   5        6
   / \      / \
  7   8    9  10
```

Ouput:
[1,2,4,7,8,9,10,6,3]

Explanation:
The left boundary are node 1,2,4. (4 is the left-most node according to definition)
The leaves are node 4,7,8,9,10.
The right boundary are node 1,3,6,10. (10 is the right-most node).
So order them in anti-clockwise without duplicate nodes we have [1,2,4,7,8,9,10,6,3
].

Solution 1

```
List<Integer> nodes = new ArrayList<>(1000);
public List<Integer> boundaryOfBinaryTree(TreeNode root) {

    if(root == null) return nodes;

    nodes.add(root.val);
    leftBoundary(root.left);
    leaves(root.left);
    leaves(root.right);
    rightBoundary(root.right);

    return nodes;
}
public void leftBoundary(TreeNode root) {
    if(root == null || (root.left == null && root.right == null)) return;
    nodes.add(root.val);
    if(root.left == null) leftBoundary(root.right);
    else leftBoundary(root.left);
}
public void rightBoundary(TreeNode root) {
    if(root == null || (root.right == null && root.left == null)) return;
    if(root.right == null)rightBoundary(root.left);
    else rightBoundary(root.right);
    nodes.add(root.val); // add after child visit(reverse)
}
public void leaves(TreeNode root) {
    if(root == null) return;
    if(root.left == null && root.right == null) {
        nodes.add(root.val);
        return;
    }
    leaves(root.left);
    leaves(root.right);
}
```

written by jaqenhgar original link here

Solution 2

We perform a single preorder traversal of the tree, keeping tracking of the left boundary and middle leaf nodes and the right boundary nodes in the process. A single flag is used to designate the type of node during the preorder traversal. Its values are:
0 - root, 1 - left boundary node, 2 - right boundary node, 3 - middle node.

```java
public List<Integer> boundaryOfBinaryTree(TreeNode root) {
    List<Integer> left = new LinkedList<>(), right = new LinkedList<>();
    preorder(root, left, right, 0);
    left.addAll(right);
    return left;
}

public void preorder(TreeNode cur, List<Integer> left, List<Integer> right, int flag) {
    if (cur == null) return;
    if (flag == 2) right.add(0, cur.val);
    else if (flag <= 1 || cur.left == null && cur.right == null) left.add(cur.val);
    preorder(cur.left, left, right, flag <= 1 ? 1 : (flag == 2 && cur.right == null) ? 2 : 3);
    preorder(cur.right, left, right, flag % 2 == 0 ? 2 : (flag == 1 && cur.left == null) ? 1 : 3);
}
```

written by compton_scatter original link here

Solution 3

```
class Solution {
public:
    vector<int> boundaryOfBinaryTree(TreeNode* root) {
        if (!root) return {};
        vector<int> ret {root->val};
        vector<TreeNode*> l, r, b;
        unordered_set<TreeNode*> s {root};
        lb(root->left, l);
        al(root, b);
        rb(root->right, r);
        for (auto n : l) { if (s.count(n)) continue; ret.push_back(n->val); s.ins
ert(n); }
        for (auto n : b) { if (s.count(n)) continue; ret.push_back(n->val); s.ins
ert(n); }
        for (auto n : r) { if (s.count(n)) continue; ret.push_back(n->val); s.ins
ert(n); }
        return ret;
    }

    void lb(TreeNode* r, vector<TreeNode*>& l) {
        while (r) {
            l.push_back(r);
            if (r->left) r = r->left; else r = r->right;
        }
    }

    void rb(TreeNode* r, vector<TreeNode*>& l) {
        while (r) {
            l.push_back(r);
            if (r->right) r = r->right; else r = r->left;
        }
        reverse(l.begin(), l.end());
    }

    void al(TreeNode* r, vector<TreeNode*>& l) {
        if (!r) return;
        if (!r->left && !r->right) l.push_back(r);
        al(r->left, l);
        al(r->right, l);
    }
};
```

written by 0xFFFFFFFF original link here