Concepts:

1. Binary trees - each node can have at max of 2 childs
2. root node
3. child node
4. leaf node
5. subtree -- node and its decendants
6. ancestors
7. Types of binary trees -
    1. Full binary tree - every node will have 0 or 2 children
    2. Complete binary tree -
        1. all levels are completely filled except the last level (completely filing the last level is optional)
        2. the last level has all the nodes as left as possible - if not completely filled
    3. Perfect binary tree - all leaf nodes are at the same level
    4. balanced binary tree - height can be at max - log(n) - where n = number of nodes
    5. degenerate tree -- skew tree -- all nodes appear to be in a same single line
8. Traversals in trees
    1. Depth first search
        1. Inorder - left - root - right
        2. PreOrder - root - left - right
        3. PostOrder - left right root
    2. Breadth first search - level order traversal

Questions

Question-1: Reverse Level Order Traversal

Statement

You have been given a Binary Tree of integers. You are supposed to return the reverse of the level order traversal.

Solution

```
vector<int> reverseLevelOrder(TreeNode<int> *root){
    // Write your code here.
    vector<int> res;
    if(root == NULL) {
        return res;
    }
    queue<TreeNode<int>*> nodesQueue;
    nodesQueue.push(root);
    while(!nodesQueue.empty()) {
        TreeNode<int>* front = nodesQueue.front();
```

```
            nodesQueue.pop();
            res.push_back(front -> val);
            if(front -> left != NULL) {
                nodesQueue.push(front -> left);
            }
            if(front -> right != NULL) {
                nodesQueue.push(front -> right);
            }
        }
        int n = res.size();
        int start = 0, end = n-1;
        while(start < end) {
            int temp = res[start];
            res[start] = res[end];
            res[end] = temp;
            start++; end--;
        }
        return res;
    }
```

Question 2: Boundary Traversal of binary tree

Statement

Given a Binary Tree, find its Boundary Traversal. The traversal should be in the following order:

Left boundary nodes: defined as the path from the root to the left-most node ie- the leaf node you could reach when you always travel preferring the left subtree over the right subtree. Leaf nodes: All the leaf nodes except for the ones that are part of left or right boundary. Reverse right boundary nodes: defined as the path from the right-most node to the root. The right-most node is the leaf node you could reach when you always travel preferring the right subtree over the left subtree. Exclude the root from this as it was already included in the traversal of left boundary nodes. Note: If the root doesn't have a left subtree or right subtree, then the root itself is the left or right boundary.

Example 1:

Input:

```
        1
      /   \
     2     3
    / \   / \
   4   5 6   7
      / \
     8   9
```

Output: 1 2 4 8 9 6 7 3

Solution

Explanation

- Basically we divide this into three parts (in this order)
    - First, traverse the left side
        - If for a node - left node is NULL, then we need to go into the right child
    - Traverse the leaf nodes (left -> right)
    - Traverse the right side (right most nodes should appear first)
        - If for a node - right node is NULL, then we need to go into the left child

```
void leftTraversal(Node* node, vector<int>& res) {
    if(node == NULL) {
        return;
    }
    if(node -> left == NULL && node -> right == NULL) {
        return;
    }
    res.push_back(node -> data);
    if(node -> left != NULL) {
        leftTraversal(node -> left, res);
    } else {
        leftTraversal(node -> right, res);
    }
}

void leafTraversal(Node* node, vector<int>& res) {
    if(node == NULL) {
        return;
    }
    if(node -> left == NULL && node -> right == NULL) {
        res.push_back(node -> data);
    }
    leafTraversal(node -> left, res);
    leafTraversal(node -> right, res);
}

void rightTraversal(Node* node, vector<int>& res) {
    if(node == NULL) {
        return;
    }
    if(node -> left == NULL && node -> right == NULL) {
        return;
    }

    if(node -> right != NULL) {
        rightTraversal(node -> right, res);
    } else {
        rightTraversal(node -> left, res);
    }
    res.push_back(node -> data);
}

vector <int> boundary(Node *root)
```

```
{
    //Your code here
    vector<int> res;
    res.push_back(root -> data);
    leftTraversal(root -> left, res);
    if(root -> left != NULL || root -> right != NULL) {
        leafTraversal(root, res);
    }
    rightTraversal(root -> right, res);
    return res;
}
```

Question 3: Preorder Traversal

Statement

Given a binary tree, find its preorder traversal. Input:

```
        1
      /
    4
  /    \
4       2
```

Output: 1 4 4 2

Solution

```
void helperRec(Node* root, vector<int>& output) {
    if(root == NULL) {
        return;
    }
    output.push_back(root -> data);
    helper(root -> left, output);
    helper(root -> right, output);
}

vector<int> helperIterate(Node* root) {
    stack<Node*> nodeStk;
    vector<int> res;
    Node* node = root;
    while(node != NULL) {
        res.push_back(node -> data);
        nodeStk.push(node);
        node = node -> left;
    }
    while(!nodeStk.empty()) {
        node = nodeStk.top();
        nodeStk.pop();
```

```
        Node* rightNode = node -> right;
        while(rightNode != NULL) {
            res.push_back(rightNode -> data);
            nodeStk.push(rightNode);
            rightNode = rightNode -> left;
        }
    }
    return res;
}

vector <int> preorder(Node* root) {
    return helperIterate(root);
    vector<int> output;
    helper(root, output);
    return output;
}
```

Question 4: Inorder Traversal

Statement

Given a Binary Tree, find the In-Order Traversal of it.

Example 1:

Input:

```
      1
    /   \
   3     2
```

Output: 3 1 2

Solution

```
void helper(Node* root, vector<int>& output) {
    if(root == NULL) {
        return;
    }
    helper(root -> left, output);
    output.push_back(root -> data);
    helper(root -> right, output);
}

vector<int> helperIterate(Node* root) {
    vector<int> res;
    stack<Node*> nodeStack;
    Node* node = root;
    while(node != NULL) {
```

```
            nodeStack.push(node);
            node = node -> left;
        }
        while(!nodeStack.empty()) {
            node = nodeStack.top();
            nodeStack.pop();
            res.push_back(node -> data);
            node = node -> right;
            while(node != NULL) {
                nodeStack.push(node);
                node = node -> left;
            }
        }
        return res;
    }

    vector<int> inOrder(Node* root) {
        // Your code here
        return helperIterate(root);
        vector<int> output;
        helper(root, output);
        return output;
    }
```