TRAVERSALS

Techniques to traverse through a Binary Tree

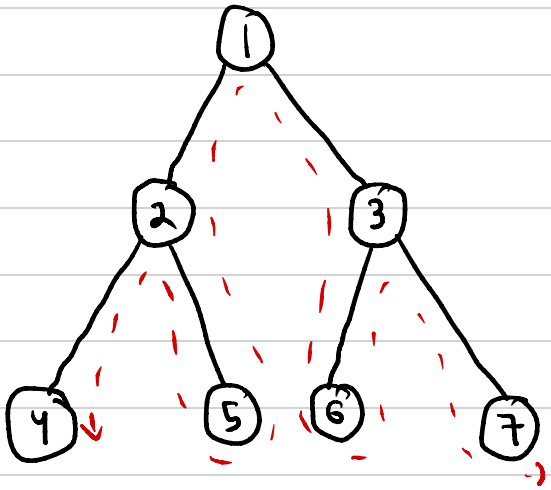
Traversal can either be depth first or breadth first.

Depth first traversal is of three types, inorder preorder and postorder

Inorder Traversal (Left Root Right)

Asks us to go to the extreme left subtree, apply our LRR logic and then move to the right subtree.

Eg :

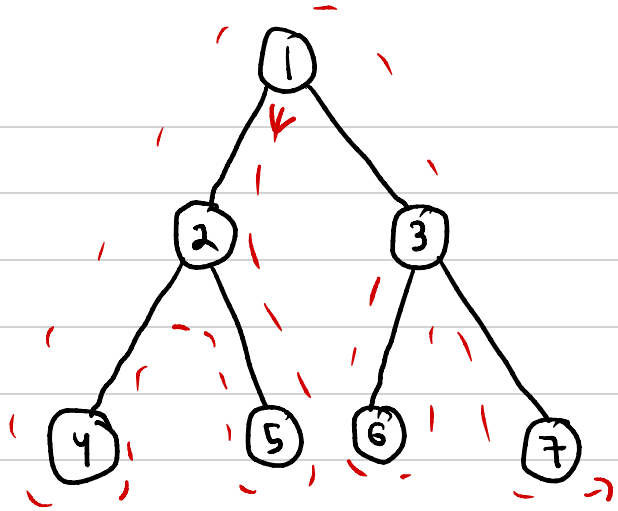


4 2 5 1 6 3 7

Preorder Traversal (Root Left Right)

Asks us to apply our RLR logic, then move to the extreme left subtree and then move to the right subtree

Eg :

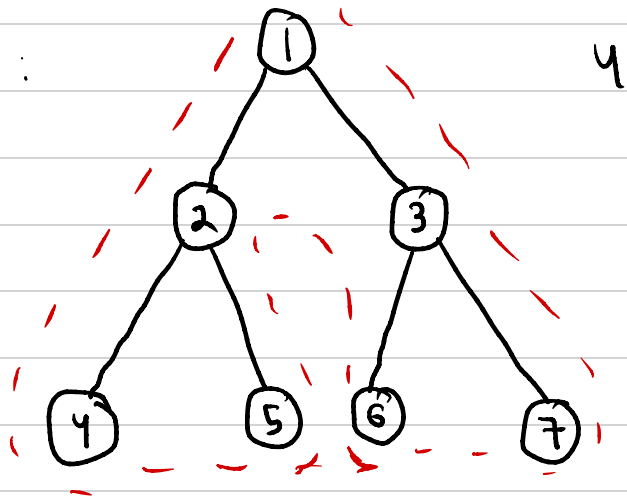


1 2 4 5 3 6 7

Post order traversals (Left Right Root)

Asks us to move to extreme left subtree, then move to extreme right subtree and then apply our LRR logic

Eg:



4 5 2 6 7 3 1

Pseudocode :

```
void preorder(Node * root) {
    if(root != NULL) {
        cout << root->value ; // Root
        preorder(root->left) ; // Left Subtree
        preorder(root->right) ; // Right Subtree
    } else {
        return ;
    }
```

```
}

void inorder(Node * root) {
    if(root != NULL) {
        inorder(root -> left) ; // Left Subtree
        cout << root -> value ; // Root
        inorder (root -> right) ; // Right Subtree
    } else {
        return ;
    }
}

void postorder(Node * root) {
    if(root != NULL) {
        postorder(root -> left); // Left Subtree
        postorder(root -> right); // Right Subtree
        cout << root -> value ; // Root
    } else {
        return ;
    }
}
```

Breadth First Search | Traversal

In Level Order Traversal, we traverse through each level of the binary tree and print each level.

We maintain a queue and a 2D array for this traversal.

At each iteration, we check if the left child and right child exists

for node at hand. If it does, then we keep adding it to a queue. As when all of this is done, we take these values out of the queue push them in a vector as this vector consists all the elements of the level.

Pseudocode :

```
levelOrderTraversal (Node * root) {
    vector <vector <int >> ans ;
    if (root == NULL) {
        return ans ;
    }
    queue <Node * > q ;
    q.push (root) ;
    while (!q.empty ()) {
        int size = q.size() ;
        vector <int > level ;
        for (int i = 0 ; i <size ; i ++) {
            Node * n = q.front () ;
            q.pop () ;
            if (n → left != NULL) {
                q.push (n → left) ;
            }
            if (n → right != NULL) {
                q.push (n → right) ;
            }
            level.push_back (n → val) ;
        }
        ans.push_back (level) ;
    }
    return ans ;
}
```

Iterative v/s Recursive

We can also use iterative algorithms to traverse through binary trees. Involves use of stacks and queues

Iterative Preorder :
We initialize the stack, push the root into it. As long as the stack is not empty, we push the right guy first and then the left guy into the stack.

This way, at each pop, left elements get printed and right ones get stored for final printing.

Pseudocode :

```
iterativePreorder(Node * root) {
    vector<int> preorder;
    if(root == NULL) {
        return preorder;
    }
    stack<Node *> st;
    st.push(root);
    while(!st.empty()) {
        root = st.top();
        st.pop();
        preorder.push_back(root->val);
        if(root->right != NULL) {
            st.push(root->right);
        }
```

```
            if (root -> left != NULL) {
                st.push(root -> left);
            }
        }
        return preorder;
    }
```

Iterative Inorder:

We initialize an empty stack. At each iteration we push the Node to the stack and move to its left. We keep doing this until a left doesnt exist, at which case we print the node and move to its right.

Pseudocode:

```
iterative Inorder (Node * root) {
    vector<int> inorder;
    if (root == NULL) {
        return inorder;
    }
    stack<Node *> st;
    Node * n = root;
    while (true) {
        if (n != NULL) {
            st.push(n);
            n = n -> left;
        } else {
            if (st.empty()) {
                break;
            }
```

```
            n = st.top();
            st.pop();
            inorder.push_back(n->val);
            n = n->right;
        }
    }
    return inorder;
}
```

Iterative Postorder using 2 Stacks:

We initialize two empty stacks. We start with the root and push it to the first stack. Now, at each iteration, we pop the top element from first stack, push it onto the second and push its left and right child to the first stack.

By the time the first stack becomes empty, our second stack already has all elements in postorder style.

Pseudocode:

```
iterativePostorderDoubleStack(Node* root) {
    Node* n = root;
    vector<int> postorder;
    if(n == NULL) {
        return postorder;
    }
    stack<Node*> st1, st2;
    st1.push(n);
    while(!st1.empty()) {
```

```
            n = st1.top();
            st1.pop();
            if(n->left != NULL) {
                st1.push(n->left);
            }
            st2.push(n);
            if(n->right != NULL) {
                st1.push(n->right);
            }
        }
        return postorder;
}
```

Iterative Postorder with 1 Stack:
We initialize a single empty stack. We initialize a mover node at root and start iterating. As long as either our stack is not empty OR our current node is not NULL, we iterate. At each iteration, push mover to the stack and move to the left if mover isn't NULL, otherwise we move to the right. If right is also NULL, we pop the top of the stack and print it.

Then we keep on printing elements until my popped node is the same as right child of the top node.

Pseudocode:

```
iterativePostorderSingleStack(Node * root) {
    vector<int> postorder;
```

```
    if (root == NULL) {
        return postorder;
    }
    stack<Node*> st;
    Node* mover = root;
    while(mover != NULL || !st.empty()) {
        if(mover != NULL) {
            st.push(mover);
            mover = mover->left;
        } else {
            Node* temp = st.top()->right;
            if(temp != NULL) {
                mover = temp;
            } else {
                temp = st.top();
                st.pop();
                postorder.push_back(temp->val);
                while(!st.empty && temp==st.top()
                                        ->right){
                    temp = st.top();
                    st.pop();
                    postorder.push_back(temp->val);
                }
            }
        }
    }
    return postorder;
}
```

All in one traversal

We construct the lists of all 3 possible depth first traversals in a

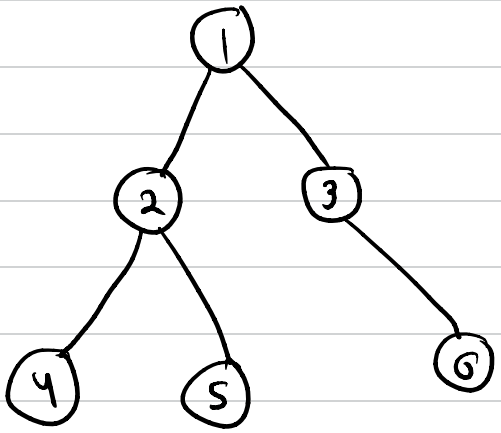
single iteration using a stack which stores node and a number

Pseudo code :

```
pre In Post Traversal (Node * root) {
    stack < pair < Node * , int >> st ;
    st.push ({root , 1}) ;
    vector <int > pre , in , post ;
    while (!st.empty ()) {
        auto it = st.top() ;
        st.pop () ;
        if (it.second == =1) {
            pre .push - back (it.first → val) ;
            it.second + =1 ;
            st.push (it) ;
            if(it.first → left ! = NULL) {
                st.push ({it.first → left , 1}) ;
            }
        } else if (it.second ==2) {
            in .push - back (it.first → val) ;
            it.second + =1 ;
            st.push (it) ;
            if(it.first → right ! = NULL) {
                st.push ({it.first → right , 1}) ;
            }
        } else {
            post.push_back (it.first → val) ;
        }
    }
}
```

Zig - Zag Traversal

Similar to level order traversal but every even level is reversed. Eg:



1 3 2 4 5 6

We maintain a flag to realize if we are at an even level or not.

Pseudocode :

```
Zig Zag Traversal (Node * root) {
    vector <vector <int>> ans ;
    if (root == NULL) {
        return ans ;
    }
    queue <Node *> q ;
    q.push (root) ;
    bool LeftToRight = true ;
    while (!q.empty ()) {
        int size = q.size() ;
        vector <int> level(size) ;
        for (int i=0 ; i <size ; i++) {
            Node * n = q.front () ;
            q.pop () ;
            int index = leftToRight ? i : (size-i-1) ;
            level [index] = n → val ;
            if (n → left != NULL) {
                q.push (n → left) ;
            }
```

```
            if (n -> right != NULL) {
                q.push(n -> right);
            }
        }
        leftToRight = !leftToRight;
        ans.push_back(level);
    }
    return ans;
}
```

Boundary Traversal

Problem asks us to the traverse across the boundary of a binary tree in either clockwise or anti clockwise order.

For anticlockwise:
Step 1: Left Boundary (excluding leafs)
Step 2: Leaf Nodes
Step 3: Right Boundary in Reverse

We accomplish Step 1 using standard traversal by sticking to the left.
Step 2 is accomplished using inorder traversal
Step 3 is accomplished using standard traversal by sticking to the right

Pseudocode:

```
leftBoundary(Node * root, vector<int> &res) {
    Node * mover = root -> left;
    while (mover != NULL) {
```

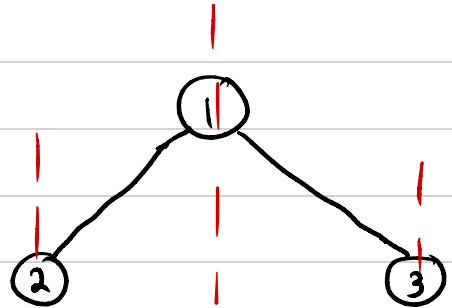
```
        if(!isLeaf(mover)) {
            res.push_back(mover->val);
        }
        if(mover->left != NULL) {
            mover = mover->left;
        } else {
            mover = mover->right;
        }
    }
}
rightBoundary(Node * root, vector<int>& res) {
    Node * mover = root;
    vector<int> t;
    while(mover != NULL) {
        if(!isLeaf(mover)) {
            t.push_back(mover->val);
        }
        if(mover->right != NULL) {
            mover = mover->right;
        } else {
            mover = mover->left;
        }
    }
    for(int i = t.size()-1; i >= 0; i--) {
        res.push_back(t[i]);
    }
}
leaves(Node * root, vector<int>& res) {
    if(isLeaf(root)) {
        res.push_back(root->data);
        return;
    }
    if(root->left != NULL) {
```

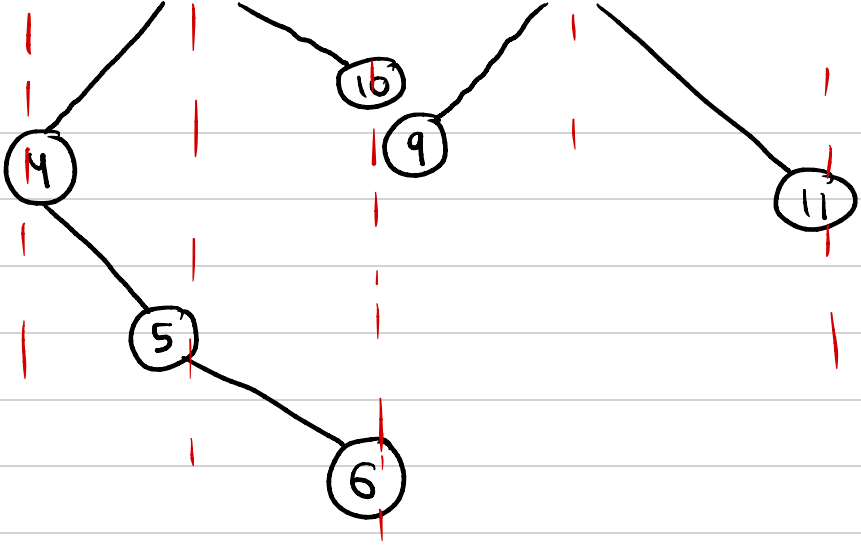
```
        leaves(root->left, res);
    }
    if(root->right != NULL) {
        leaves(root->right, res);
    }
}
isLeaf(Node * n) {
    return (n->left == n->right == NULL);
}

boundaryTraversal(Node * root) {
    vector<int> ans;
    if(root == NULL) {
        return ans;
    }
    if(!isLeaf(root)) {
        ans.push_back(root->val);
    }
    leftBoundary(root, ans);
    leaves(root, ans);
    rightBoundary(root, ans);
    return ans;
}
```

Vertical Order Traversal

Problem asks us to traverse through a binary tree from left to right by verticals.

Eg:





In case of overlapping nodes, we display them in a sorted order.
To facilitate this traversal, we format the tree in a cartesian format where each level is y and each vertical is x.

We use a queue which stores the node along with its x and y
We also use a map which maps each vertical to an integer.

We start iterations with the root at 0, 0 already in our queue.
Move to left, add it to our queue,
move to right, add it to our queue.
As we are done adding children of our node to the queue, we dequeue that element.
They are also simultaneously added to each vertical in the map.

Pseudocode :

```
verticalOrderTraversal (Node * root) {
```

```
vector<vector<int>> ans;
map<int, map<int, multiset<int>>> nodes;
queue<pair<Node*, pair<int, int>>> todo;
todo.push(root, {0,0});
while(!todo.empty()) {
    auto p = todo.front();
    todo.pop();
    Node* n = p.first;
    int x = p.second.first;
    int y = p.second.second;
    nodes[x][y].insert(n->val);
    if(n->left != NULL) {
        todo.push(n->left, {x-1, y+1});
    }
    if(n->right != NULL) {
        todo.push(n->right, {x+1, y+1});
    }
}
for(auto p : nodes) {
    vector<int> col;
    for(auto q : p.second) {
        col.insert(col.end(), q.second.begin(),
                               q.second.end());
    }
    ans.push_back(col);
}
return ans;
}
```