Imad’s Interview Prep Guide

McGill University

Table of Contents

1. Introduction

2. Trees

2.1. Binary Trees

2.1.1. What is a Binary Tree?

*A binary tree is a tree where each non-leaf node has a **left** subtree and a **right** subtree*

*A binary tree is **not** a binary search tree - which maintains the above property but also says that: * Everything in the **left** subtree is **less than** the value of the node. * Everything in the **righ** subtree is **greater than** the value of the node.*

In Java we can define a node as:

```
class TreeNode
{
    int value;
    TreeNode left;
    TreeNode right;

    public TreeNode(int value){
        this.value = value;
        left = right = null;
    }
}
```

And the tree as a whole with:

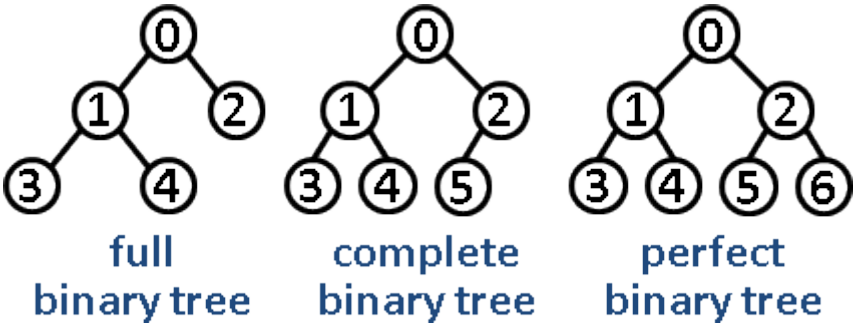
```
class BinaryTree
{
    TreeNode root;

    public BinaryTree(){
        root = null;
    }
}
```

2.1.2. Words you should know:

- **Depth** - The **depth** of a node is **how many edges it is away from the root of the whole tree.** - Note that this means **root's depth is 0**.
- **Height** - The **height** of a node is **maximum number of edges between it and a leaf**.
- **Tree Height** - The **height of the root of the tree**.
- **Full** - A **full** binary tree is a binary tree where each of the **non-leaf nodes has two children**.
- **Complete** - A **complete** binary tree is a binary tree where in the tree, **each level is completely filled** - **except possibly the last**. (i.e. **its fine if the last level isn't filled**.)
- **Perfect** - A **perfect** tree is a **full** tree where **all leafs are on the same level**.

- **Balanced Tree** - A tree where the height is O(logn).
- **Degenerate Tree** - A linked list.



From CodeKaksha: Comparison of Full, Complete and Perfect Trees

2.1.3. Questions on Binary Trees

- **Symmetric Tree**
- **Maximum Depth of Binary Tree**
- **Invert Binary Tree**

2.2. Binary Tree Traversals

*There are **four** basic Tree Traversals. The three **Depth-First Traversals** - because they go as far deep down the tree as they can before coming back up, and **Breadth-First Traversal** a.k.a. **Level-Order Traversal** because they survey all possible things at a given level before moving down.*

2.2.1. Depth First Tree Traversals

A note before we get started: often, you will hear that recursive implementations for these depth first traversals are pretty trivial. This is because the recursive implementations are essentially just the different combinations of three statements:

- Visit current node
- Recurse on current node's left
- Recurse on current node's right

Where visit is some action you want to do on the node, in the Java code I provide, it's just printing the value - but it could be something else in practice like adding the node's value to a List, or a Stack or a HashMap or whatever…

You will also note below that, by convention, **we always recurse left before right**. For In-Order Traversals of Binary Search trees, this will actually effect whether we print out the values in sorted ascending or sorted descending order, but that is for another time…

Pre-Order Traversal

Visit the **current node before the left then right child**.

Recursively:

```
void preOrderPrint(TreeNode node){
    System.out.println(node.val);
    preOrderPrint(node.left);
    preOrderPrint(node.right);
}
```

In very straight-forward pseudo-code to help you remember it's:

```
visit(me)
recurse(me.left)
recurse(me.right)
```

The iterative (non-recursive) solution is not so easy:

```
void iterativePreOrder(TreeNode node){
    Stack<TreeNode> st = new Stack<>();
    if(node != null) st.push(node);

    while(!st.isEmpty(){
        TreeNode me = st.pop();
        System.out.println(me.val);

        if(me.right != null) st.push(me.right);
        if(me.left != null) st.push(me.left):
    }
}
```

Essentially, what we are doing is maintaining a stack to keep track of the order of visitation of the nodes.

For a Pre-Order traversal we must ensure the following properties:

- Visit current before either sub-tree.
    - Pretty trivial to see this above. We don't even touch the left or right children before printing (visiting) the current node.
- Visit entire left sub-tree before right sub-tree.
    - Stacks conserve LIFO access (Last In First Out) - by pushing right node and then left node, we ensure that at the next iteration the left-node is visited. This repeats so at the iteration after that, *it's* left node is visited. Inductively, we visit all left subtrees before right subtrees.

Post-Order Traversal

Visit the **current node after the left then right child**.

Recursively:

```
void postOrderPrint(TreeNode node){
    postOrderPrint(node.left);
    postOrderPrint(node.right);
    System.out.println(node.val);
}
```

Again, with pseudo-code:

```
recurse(me.left);
recurse(me.right);
visit(me);
```

Iteratively:

```
void iterativePostOrder(TreeNode node){
    Stack<TreeNode> st = new Stack<>();

    if(node != null){ st.push(node); }
    TreeNode prev = null;
    while(!st.isEmpty()){
        TreeNode current = st.peek();
        if(prev == null || prev.left == current || prev.right == current){
            if(current.left != null) st.push(current.left);
            else if (current.right!= null) st.push(current.right);
            else{
                st.pop();
                System.out.println(current.val);
            }
        else if(current.left == prev){
            if(current.right != null) st.push(current.right);
            else {
                st.pop();
                System.out.println();
            }
        }
        else if(current.right == prev){
            st.pop()
            System.out.println();
        }

        prev = current;
    }
}
```

In-Order Traversal

In-order traversal has its name because **if you call it on a Binary-Search Tree (not an arbitrary Binary Tree), it will print out the values in order**. It is essentially, the last combination of the three instructions in our above pseudocode, i.e.: where you **recurse on the left child, then visit the current node, then visit the right child**.

Recursively:

```
void inOrderPrint(TreeNode node){
    inOrderPrint(node.left);
    System.out.printl(node.val);
    inOrderPrint(node.right);
}
```

Pseudo-code:

```
recurse(me.left);
visit(me);
recurse(me.right);
```

Iteratively:

```
void iterativeInOrder(TreeNode node){
    Stack<TreeNode> st = new Stack<>();

    Node current = node;

    while(current != null || !st.isEmpty()){
        while(current != null){
            st.push(current);
            current = current.left;
        }

        current = st.pop();
        System.out.println(current.val);
        current = current.right;
    }
}
```

Questions on Depth First Traversals

- **In-Order Traversal** - Recursively then Iteratively.
- **Pre-Order Traversal** - Recursively then Iteratively.
- **Post-Order Traversal** - Recursively then Iteratively.
- **House Robber III**

- **Reconstruct Tree from In-Order and Post-Order Traversals**