Imad’s Interview Prep Guide

A guide to understanding algorithms you're probably never going to touch again.

Table of Contents

1. Introduction

1.1. My Rambles

Hi! Welcome to my Interview Prep Guide! If you don't already know me, my name is Imad Dodin, I'm in the fourth year of my Bachelor's of Engineering (Software) at McGill University. At McGill, I think I'm mostly recognizable as Co-President and ICPC Coach for Compete McGill, McGill's Competitive Programming club. In my 3rd year, I acted as Training Lead, and tried to make algorithms as accessible as possible for people - once you get them, they're really fun!

In the Summer of my 3rd Year, I interned at Amazon in Vancouver, where I worked on the Amazon Web Services S3 Index Control Plane team - it was a really cool summer and I learned a lot while working with a whole bunch of really smart and friendly people! I was lucky enough to get a return offer for full-time, but due to some delays with my graduation date, I decided that I'd try and get experience at another company this Summer and was thrilled to accept an offer last month to intern at Microsoft in Redmond.

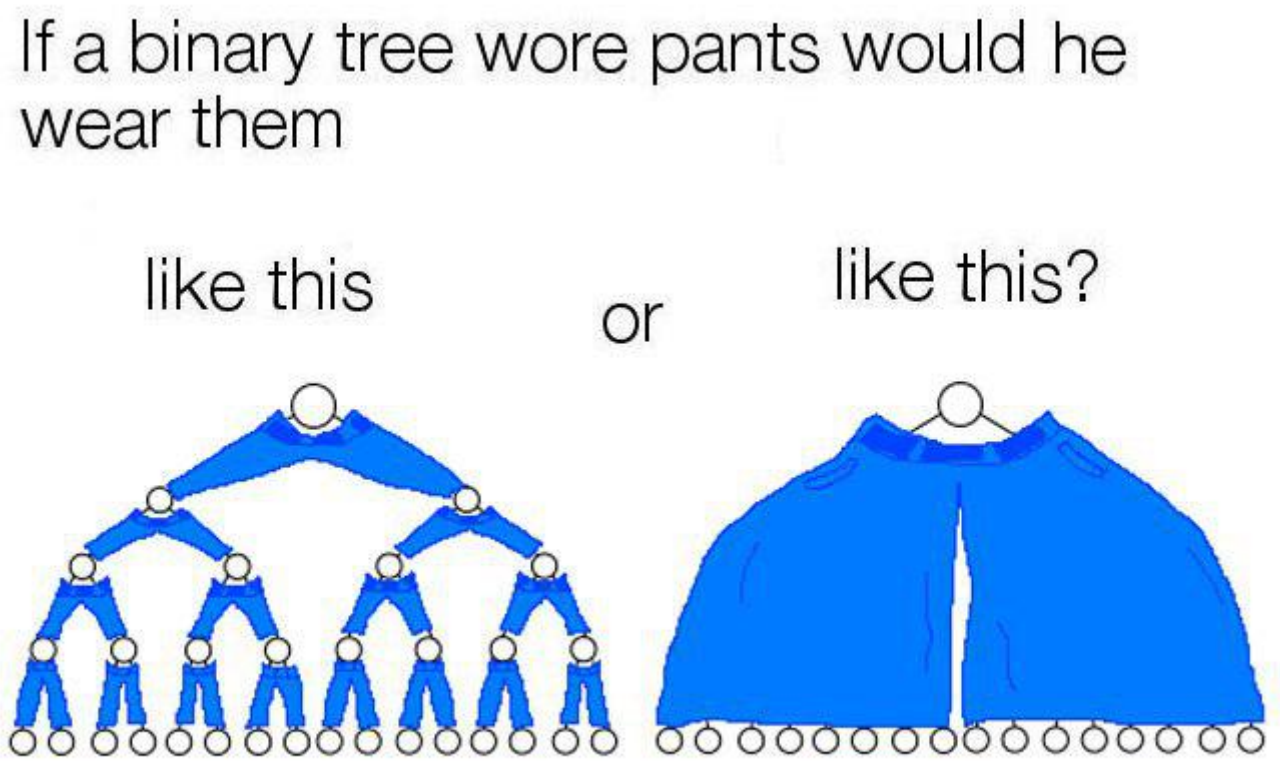
I decided to start writing this, largely because I miss writing and talking about Algorithms in a non-competitive environment (aside: I don't really even do much coaching, our representatives are much too smart to listen to me). Interview Prep is really scary for a bunch of reasons, and I wanted to attempt to lessen the load on people going through it. This guide aims to try and explain Data Structures and Algorithms as simply as possible and without any added complexity (you need them for your interviews, let's have fun and look at harder stuff another time), it aims to be a one-stop shop for pseudo-code, Java code, runtimes and LeetCode problems that I found useful when prepping myself and others.

In short:

- I love algorithms and need more human interaction.
- It's annoying to spend a whole bunch of time looking for stuff - I'll try to put everything you need here (you might still need to Google if stuff is a little confusing).
- I've interviewed quite a bit (others have interviewed even more than me) - but I've also spent a whole bunch of time coaching, mock interviewing and presenting algorithms to students.

2. Trees

2.1. Binary Trees



Pants

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 **right** 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;
    }
}
```

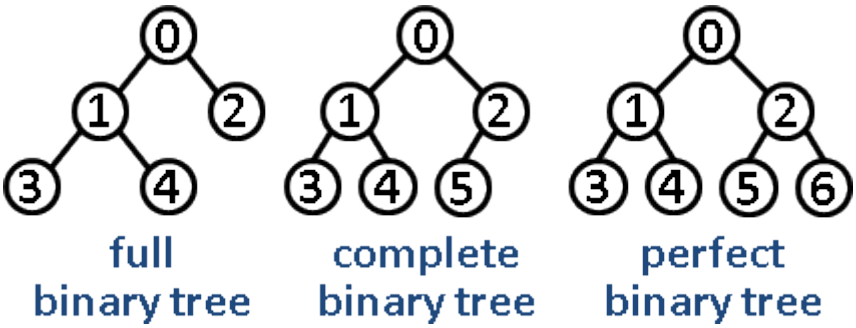
And the tree as a whole with:

```
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**. - **Sometimes people define it as number of nodes, in which case the root's depth is 1 - clarify this with your interviewer**.
- **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 Search Trees

As mentioned above, **Binary Search Trees** are binary trees where:

- Everything in **left subtree is less than the parent node**
- Everything in **right subtree is greater than the parent node**

With a binary search tree we can, well, binary search!

2.2.1. Binary Searching on a BST

The algorithm is pretty straightforward for Binary Search Trees:

```
binarySearch(int key, TreeNode node):
    if(node == null):
        return node ①

    if(node.val == key):
        return node ②
    else if(node.val < key):
        return binarySearch(node.left)
    else if(node.val > key):
        return binarySearch(node.right)
```

① Change this to `false` if you want to return a boolean instead.

② Change this to `true` if you want to return a boolean instead.

Where we start the function at root, i.e. `binarySearch(key, root)`. Note that the above function returns the node where we found the value, which will be null if the value doesn't exist in the binary search tree. I provide comments to demonstrate how to change the function if we want to return a boolean instead.

In Java Code:

```
TreeNode binarySearch(int key, TreeNode node){
    if(node == null || node.val == key){
        return node;
    }

    if(node.val < key) return binarySearch(node.left);
    else return binarySearch(node.right);
```

2.2.2. Questions on Binary Search Trees

- **Unique Binary Search Trees**
- **Validate Binary Search Trees**

2.2.3. Inserting and Deleting Nodes in BSTs

Inserting and Deleting Nodes in a Binary Search Trees are common operations that you should be able to do.

Inserting Nodes in BST

We have a **value, the key** which we want to insert into a Binary Search Tree. In order to insert it into the BST, we need to travel down the tree until we **find an appropriate slot for our new leaf**. This means we insert the values as **leafs at the bottom of the tree and not as intermediate nodes somewhere in the middle** - because it makes our life simpler.

Pseudocode:

```
insertNode(int key, TreeNode node):
    if(node == null): ①
        return new TreeNode(key)

    if(key <= node.val): ②
        node.left = insertNode(key, node.left)
    else: ③
        node.right = insertNode(key, node.right)

    return node
```

① If the current node is null, it means we moved down the tree and found a slot for the new leaf.

② If the key is less than or equal to the current node's value, its appropriate slot should be somewhere in the left sub-tree, so we recurse down to the left sub-tree.

③ If the key is greater than the current node's value, its appropriate slot should be somewhere in the right sub-tree, so we recurse down to the right sub-tree.

Java Code:

```
TreeNode insertNode(int key, TreeNode node){
    if(node == null) return new TreeNode(key);

    if(key <= node.val) node.left = isertNode(key, node.left);
    else node.right = insertNode(key, node.right);

    return node;
```

Deleting Nodes in BST

When deleting nodes in a BST, we have a value that we want to find and delete from the Binary Search Tree. There are 4 cases:

- The value doesn't exist in the tree.
- The node we want to delete has 0 children (leaf)
- The node we want to delete has 1 child
- The node we want to delete has 2 children

From Geeks4Geeks, here are some illustration for these cases:

Leaf Node:

```
            50                            50
         /     \         delete(20)      /   \
        30      70       --------->    30     70
       /  \    /  \                     \    /  \
      20   40  60   80                   40  60   80
```

Pretty easy: just remove the node!

1 Child:

```
            50                            50
         /     \         delete(30)      /   \
        30      70       --------->    40     70
          \    /  \                          /  \
          40  60   80                       60   80
```

Again, easy: remove the node and replace it with the child.

2 Children:

```
            50                            60
         /     \         delete(50)      /   \
        40      70       --------->    40    70
               /  \                           \
              60   80                          80
```

Here, its a little more tricky. What we'll do is remove the node (obviously) - we then need to find an appropriate node in one of the subtrees to replace this node. Logically there are two options:

- The smallest value in the right sub-tree (smallest thing that is bigger than the current node) - i.e. next biggest value.
- The largest value in the left sub-tree (largest thing that is smaller than the current node) - i.e. next smallest value.

This is called finding the **in-order successor** of the current node. I'll provide Java code below that does the first option.

```
TreeNode delete(int key, TreeNode node){
    if(node == null) return node; ①

    if(key < node.val) node.left = delete(key, node.left); ②
    else if (key > node.val) node.right = delete(key, node.right); ③
    else {
        if(root.left == null) return node.right; ④
        else if(root.right == null) return node.left; ⑤
        node.val = findMin(node.right); ⑥
        node.right = delete(node.val, node.right); ⑦
    }

    return node;
}

int findMin(TreeNode node){
    while(node.left!=null){ ⑧
        node = node.left;
    }
    return node.left.val;
}
```

① If the current node is null, we've recursed the whole tree and didn't find the value we want to delete.

② If the value we want to delete is less than the current node's val, then if a node exists for it, it will be in the left subtree - recurse left.

③ If the value we want to delete is greater than the current node's val, then if a node exists for it, it will be in the right subtree - recurse right.

④ We are on the node we want to delete - if it has one child - replace the node with the node that does exist, because (2) and (3) set a node's child to be whatever is returned by this call, its enough to just return the child (which would cause the current node to be abandoned).

⑤ Same as above, note that if we have 0 children, then (4) will be executed and we just replace the current node with null which is the same as just deleting it.

⑥ Now we are in the case that we have two children - we need to find the minimum value in the right subtree. We set the current's node value to be that value - essentially copying that minimum node.

⑦ Now we have two copies of the minimum node in the right subtree, we need to delete the one that exists in the right sub-tree - now its as if we swapped the nodes.

⑧ Finding the minimum value in a tree is trivial, just keep going to the left child until you hit the left-most leaf. Return that value. === 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. _

2.2.4. 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);①

        if(me.right != null) st.push(me.right);
        if(me.left != null) st.push(me.left):②
    }
}
```

① Print (Visit) - before touching sub trees.

② Push Right then Left so that Left is popped first (Last In First Out).

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 (this one is the most difficult - I suggest drawing it out):

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

    if(node != null){ st.push(node); } ①
    TreeNode prev = null; ②
    while(!st.isEmpty()){
        TreeNode current = st.peek();③
        if(prev == null || prev.left == current || prev.right == current){ ④
            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){⑤
            if(current.right != null) st.push(current.right);
            else {
                st.pop();
                System.out.println();
            }
        }
        else if(current.right == prev){⑥
            st.pop()
            System.out.println();
        }

        prev = current;⑦
    }
}
```

① If root not null, start the traversal.

② On any given iteration of the while-loop, keep track of the node we visited on the iteration before. This helps us know whether we have moved up the Binary Tree or are still moving down.

③ We peek at the Stack (get the top value without popping it) - We only decide to remove it if we visit the node - which will only ever be done if the left and right sub-trees are visited (in that order).

④ This is the big one - this block says the following: Prioritize moving down the left subtree first, then the right - Only if there are no sub-trees (we're at a leaf), do you pop the node from the stack and visit it.

⑤ If the last node we visited was left of the current node, it means that we've visited everything in the left sub-tree. In this case, let's start visiting the right sub-tree if it exists, otherwise pop the node and visit it.

⑥ If the last node we visited was right of the current node, it means we've visited all the sub-trees (because we already enforced that left sub-tree needs to have been visited for us to start visiting

the right sub-tree). In this case, there is nothing left for us to do but visit the current node and pop it from the stack.

⑦ This iteration's current is next iteration's previous.

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){ ①
            st.push(current);
            current = current.left;
        }

        current = st.pop(); ②
        System.out.println(current.val);
        current = current.right; ③
    }
}
```

① Keep pushing left node's onto the stack - this ensures that we visit the left subtrees first.

② Pop the current node from the Stack and visit it, because of Last In First Out ordering and (1), if a node is popped, it's left sub-tree needs to have been visited.

③ Set the current node to be the right child so that we can now start doing the In-Order traversal for the right sub-tree.

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**
- **Kth Smalles Element in a Binary Search Tree**