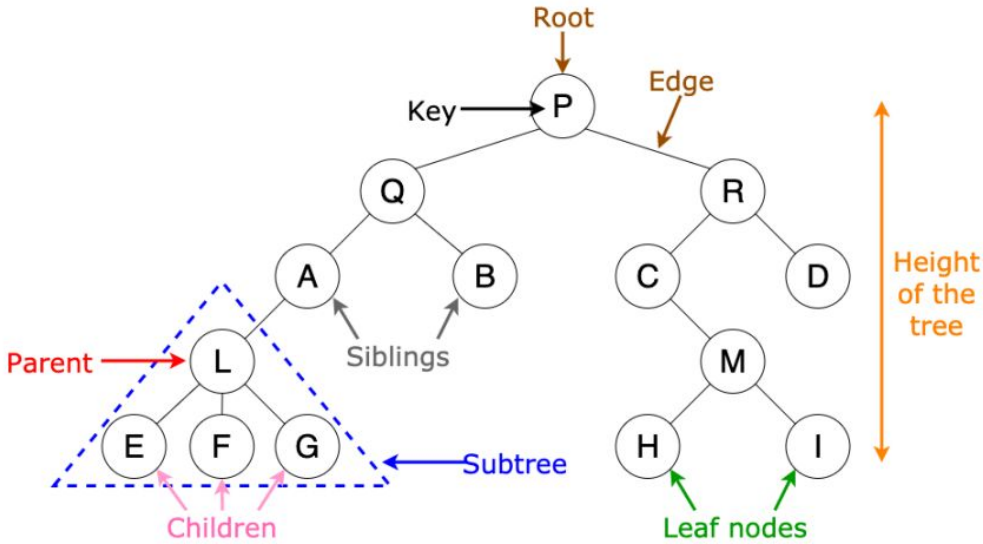
Binary Trees on Java

Introduction to Binary Trees

What is a Binary Tree?

A tree whose elements have at most 2 children is called a binary tree. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.

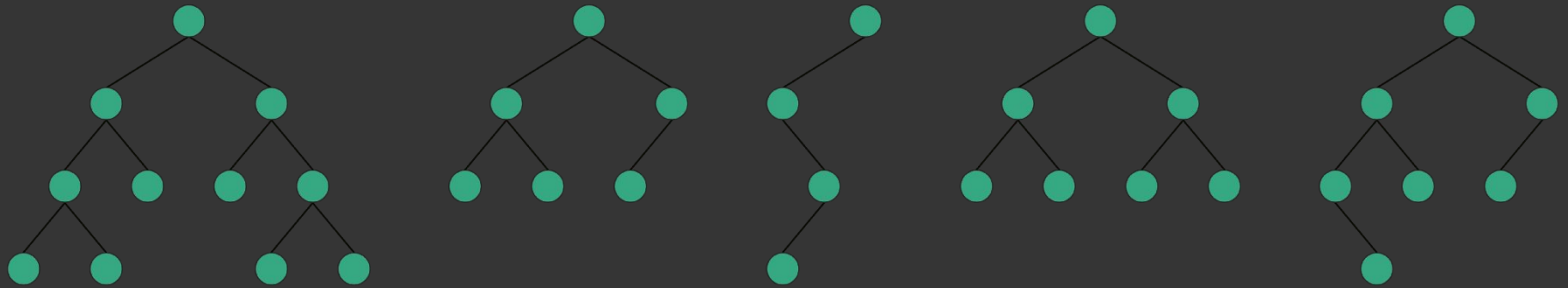


1. Intro

A Binary Tree node contains following parts.

- **Data**
  main uses of trees include maintaining hierarchical data
- **Pointer to left child**
  providing moderate access and insert/delete operations
- **Pointer to right child**
  doesn't need both children

Different Types of Binary Trees



Full

Complete

Degenerate

Perfect

Balanced

Full Binary Tree : Every node has 0 or 2 children. We can also say a full binary tree is a binary tree in which all nodes except leaf nodes have two children.

Complete Binary Tree: All the levels are completely filled except possibly the last level and the last level has all keys as left as possible.

Degenerate (or pathological) Tree: Where every internal node has one child. Such trees are performance-wise same as linked list.

Perfect Binary Tree: The internal nodes have two children and all leaf nodes are at the same level.

Balanced Binary Tree: The height of the left and right subtree of any node differ by not more than 1.

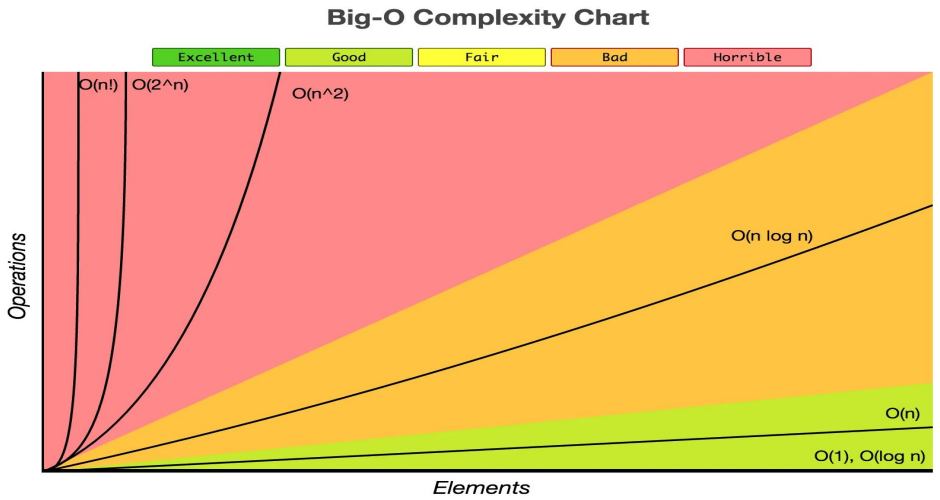
Tip

Binary Trees require you to think non-linearly because they are a branched data structure and in reverse because to traverse a binary tree often means using recursion going depth first.

Why Use Trees?

You can search, and insert/delete items quickly in a tree

- Ordered Arrays are bad at Insertions/Deletions
- Finding items in a Linked List is slow
- Time needed to perform an operation on a tree is O(log N)
- On Average a tree is more efficient if you need to perform many different types of operations



Hint:

Big O notation is a mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity.

2. In Order Traversal

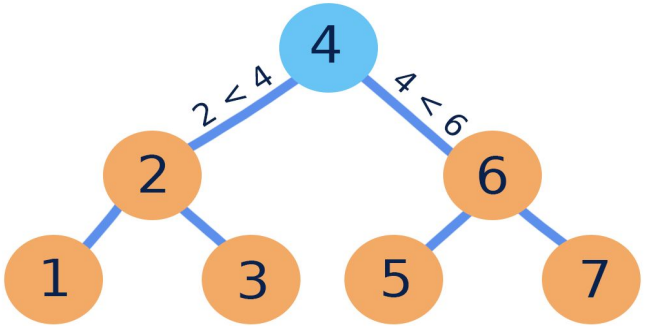
Aim for the smallest value First

➔ **START**

Start at 1st Left Child

➔ **Null**

When Null is reached then move up in value



In Order Traversal: 1 2 3 4 5 6 7

3. Preorder Traversal

Aim for Pre-order

➔ **START**

Start at **Root**

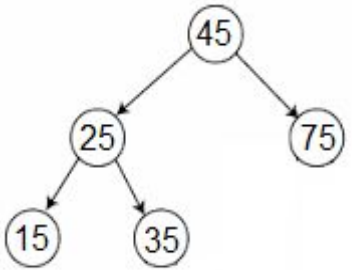
➔ **Left**

Cycle down through all of our **Left Children**

➔ **Null**

Jump up one **Parent** and go to our **Right Child**

Binary search Tree - Preorder Traversal



Preorder traversal:

45, 25, 15, 35, 75

4. Postorder Traversal

Aim for Post-order

➔ **Left**

Start at leftmost **Child**

➔ **Null**

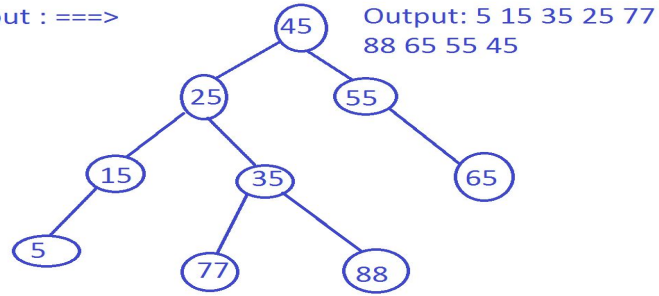
Cycle through **Right Child** of **Parent Node**

➔ **Last**

Our **Root** will comes last

Post Order Traversal of a Binary Tree in Java

Input : ===>

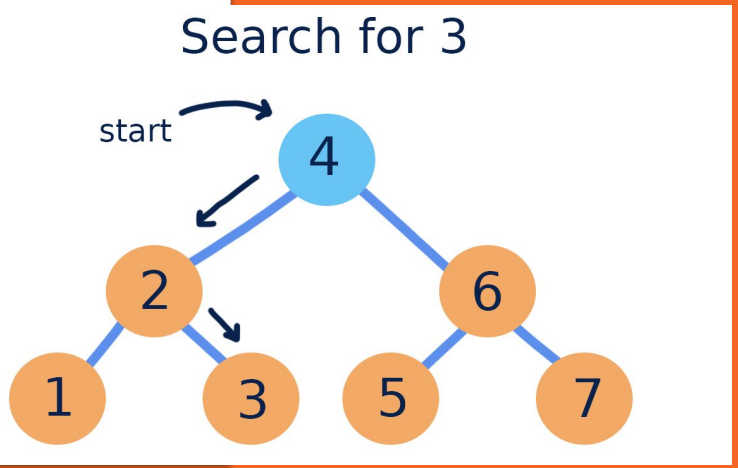


Output: 5 15 35 25 77 88 65 55 45

4. Binary Search Trees

Aim to find Node

- **START**
  Start at **Root**
- **Find**
  Cycle through **Binary Tree**
- **Less than**
  Cycle through **Left Children**
- **Greater than**
  Cycle through **Right Children**
- **Null**
  The **Node** wasn't found



Node-Based Binary Tree

- ➔ The left subtree of a node only contains nodes with keys less than the node's key.
- ➔ The right subtree of a node only contains nodes with keys greater than the node's key.

Tip

The left and right subtree **each** must also be a binary search tree.