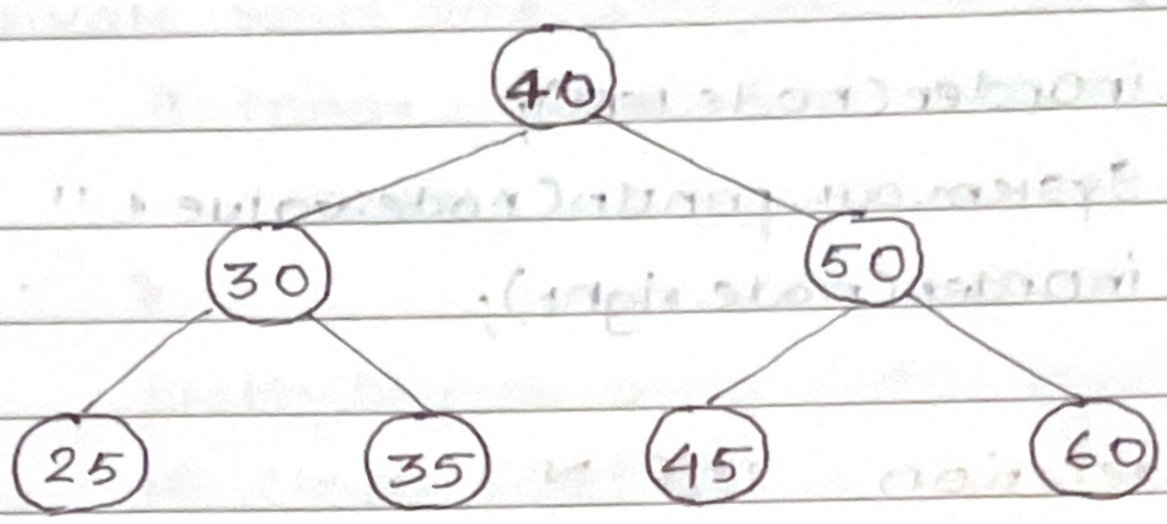
Binary Search Tree (BST)

A binary search tree follows some order to arrange the elements. In Binary search tree, the value of left node must be smaller than parent node and the value of right node must be greater than the parent node.



Advantage of Binary search tree

- Searching an element in Binary search tree is easy as we always have a hint that which subtree has the desired element.
- As compared to array and linked list insertion and deletion operations are faster in BST.

Implementation

```
class BST {
    public class Node {
        private int value;
        private Node left;
        private Node right;
        private int height;
        public Node(int value) {
            this.value = value;
        }
        public int getValue() {
            return value;
        }
}
```

```
private Node root;
if (node == null) {
    return -1;
}

public BST() { }
public int height(Node node) {
    if (node == null) {
        return -1;
    }
    return node.height;
}

public boolean isEmpty() {
    return root == null;
}

public void insert(int value) {
    root = insert(value, root);
}

private Node insert(int value, Node node) {
    if (node == null) {
        node = new Node(value);
        return node;
    }
    if (value < node.value) {
        node.left = insert(value, node.left);
    }
    if (value > node.value) {
        node.right = insert(value, node.right);
    }
    node.height = Math.max(height(node.left), height(node.right)) + 1;
    return node;
}
```

```
    public void populate(int[] nums) {
        for(int i= 0; i< nums.length; i++) {
            this.insert(nums[i]);
        }
    }
    public void populatedSorted() {
        populatedSorted(nums, 0, nums.length);
    }
    private void populatedSorted(int[] nums, int start, int
        end) {
        if (start >= end) {
            return;
        }
        int mid = (start + end)/2;
        this.insert(nums[mid]);
        populatedSorted(nums, start, mid);
        populatedSorted(nums, mid+1, end);
    }
    public boolean balanced() {
        return balanced(root);
    }
    private boolean balanced(Node node) {
        if (node == null) {
            return true;
        }
        return Math.abs(height(node.left) - height(node.right))
            <=1 && balanced(node.left) && balanced
            (node.right);
    }
    public void display() {
        display(this.root, "Root Node:");
    }
```

```
private void display (Node node, String details) {
    if (node == null) {
        return;
    }
    System.out.println (details + node.value);
    display (node.left, "left child of " + node.value + ":");
    display (node.right, "Right child of " + node.value + ":");
}
}
```

Problems with Binary Search Tree.

Unbalanced trees: In a standard BST nodes are inserted in a way that can lead to formation of long chains. The result is unbalanced tree, where the height can become bad as O(N) in the worst case, making operations like search insert and delete inefficent compared to the ideal O(log N) time complexity.

Solution: Self Balancing Binary Tree

- Balanced Trees

A balanced tree is one where for every node in the tree, the difference in the height of the left and right subtree is -1, +1, or 0.

This condition ensures that the tree remains approximately balanced at all times.