**Tree**

**Binary Tree** – each parent node has exactly 2 children nodes.

**Full Tree** – each parent node has either 0 or 2 children nodes.

Left is a full tree.

Right is not a full tree.

A screenshot of a computer

Description automatically generated

**Perfect Tree** – left and right side of the **root** node has the exact same amount of nodes.

Left is a perfect tree.

Right is not a perfect tree.

A screenshot of a computer

Description automatically generated

**Complete Tree** – with each row of nodes starting from the root, if there is at least one node at the left side of the tree, it is complete. Has all the levels filled with nodes except the last level. The last level are filled with nodes starting from the left side to the right.

The left side is a complete tree.

The right side is not a complete tree.

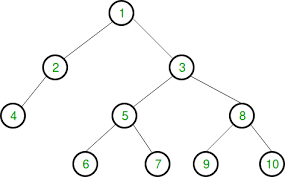
A screenshot of a computer

Description automatically generated

**Leaf Nodes**

These are nodes that do not have any children.

Nodes 4, 6, 7, 9, 10 = leaf nodes



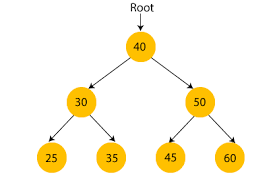
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Binary Search Tree**

Starting from the root, all nodes less than parent node in on the left and all nodes greater than parent are on the right.

node **<** parent = left

node **>** parent = right



This tree has 3 levels:

1. 40
2. 30, 50
3. 25, 35, 45, 60

Each level can be viewed as:

40 🡪 2^1 – 1 = 1 (1 node in tree)

30, 50 🡪 2^2 – 1 = 3 (3 node in tree)

25, 35, 45, 60 🡪 = 2^3 – 1 = 7 (7 node in tree)

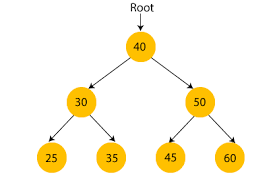
Steps it would take to find, add, or remove a given node:

* 40 = 1 step
* 50 = 2 steps
* 45 = 3 steps

This would mean that it is O(log n)

**O(log n)**

Divide and Conquer algorithm



Say we want to find node 45:

We start at the root which is 40.

Since 45 > 40, this means that we can eliminate everything on the left side.

Now we go right which is 50.

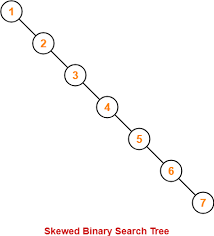
Since 45 < 50, this means that we can eliminate everything on the right side.

Finally, we find node 45.

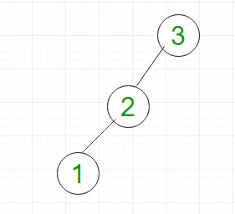
This is the divide and conquer algorithm at work resulting in =O(log n).

The worst-case scenario is if the node we are looking for is either all the way left, or all the way right resulting in O(n).

Searching for node 7:



Searching for node 1:



**O(log n)**

* insert() – better with linked list because if we insert at the end, it is an O(1)
* lookup() – better with binary search tree O(log n)
* remove() – better with binary search tree O(log n)

class Node {

    constructor(value) {

        this.value = value;

        this.left = null;

        this.right = null;

    }

}

class BST {

    constructor() {

        this.root = null;

    }

    insert(value) {

        const newNode = new Node(value);

        if (this.root === null) {

            this.root = newNode;

            return this;

        }

        let temp = this.root;

        while (true) {

            if (newNode.value === temp.value) return undefined;

            if (newNode.value < temp.value) {

                if (temp.left === null) {

                    temp.left = newNode;

                    return this;

                }

                temp = temp.left;

            } else {

                if (temp.right === null) {

                    temp.right = newNode;

                    return this;

                }

                temp = temp.right;

            }

        }

    }

    contains(value) {

        if (this.root === null) return false;

        let temp = this.root;

        while (temp) {

            if (value < temp.value) {

                temp = temp.left;

            } else if (value > temp.value) {

                temp = temp.right;

            } else {

                return true;

            }

        }

        return false;

    }

    minValueNode(currentNode) {

        while (currentNode.left !== null) {

            currentNode = currentNode.left;

        }

        return currentNode;

    }

}