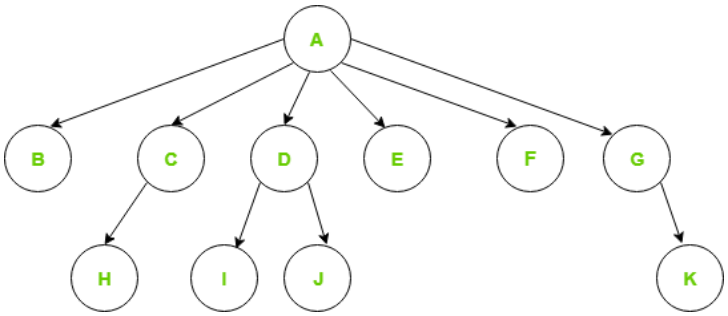
# BINARY TREE

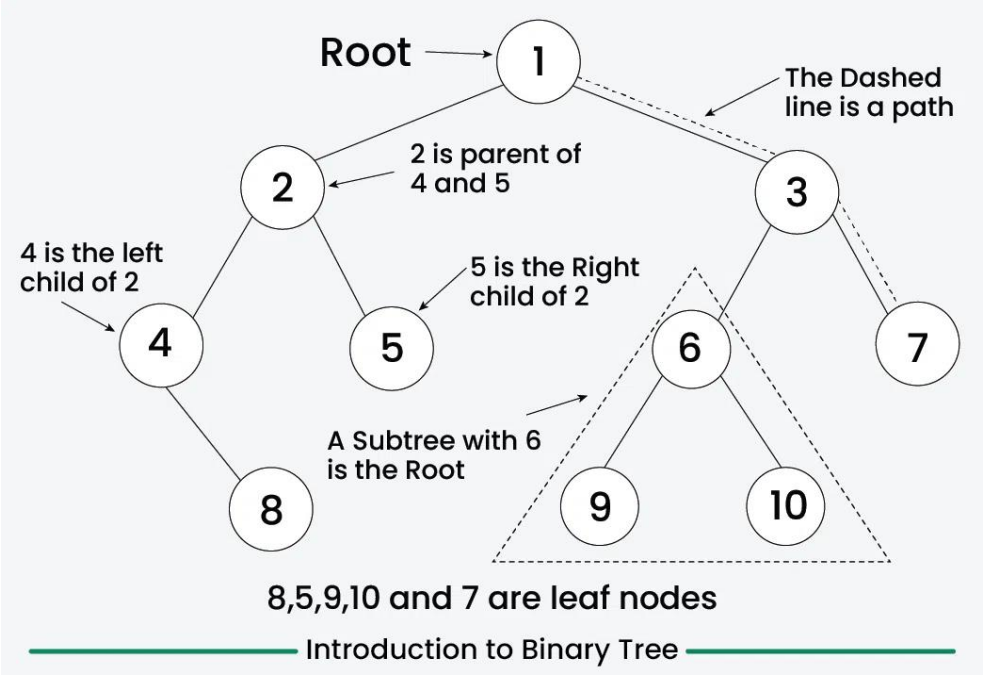
## Generic Tree:

Generic trees are a collection of nodes, and each node is a data structure that consists of records and a list of references to its children (duplicate references are not allowed). Unlike the linked list, each node stores the address of multiple nodes. Every node stores the address of its children, and the very first node's address is stored in a separate pointer called root.




## Binary Tree:

Binary Tree is a non-linear data structure where each node has at most two children, referred to as the left and correct child. The topmost node in a binary tree is called the root, and the bottom-most nodes are called leaves.



**Operations in Binary Tree:**

→ **Creating a Binary Tree:**

**Code:**

```
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
class Solution:
    def bstFromPreorder(self, preorder: List[int]) -> Optional[TreeNode]:
        if not preorder:
            return None
        root = TreeNode(preorder[0])
        root.left = self.bstFromPreorder([i for i in preorder if i < preorder[0]])
        root.right = self.bstFromPreorder([i for i in preorder if i > preorder[0]])

        return root
```

**Output:** 2, 3, 4, 5

**Explanation:**

Data= data, Left =None, Right = None.

→ Node Initialization:

Four nodes are created:

firstNode = Node(2)

secondNode = Node(3)

thirdNode = Node(4)

fourthNode = Node(5)

→ Tree Structure:

```
    2
   / \
  3   4
 /
5
```

→ **Traversal in Binary Tree:**

Traversal in Binary Tree involves visiting all the nodes of the binary tree. Tree Traversal algorithms can be classified broadly into two categories, DFS and BFS:

**Depth-First Search (DFS) algorithms:** DFS explores as far down a branch as possible before backtracking. It is implemented using recursion. The main traversal methods in DFS for binary trees are:

*Preorder Traversal (current-left-right):* Visits the node first, then left subtree, then right subtree.
*Inorder Traversal (left-current-right):* Visits left subtree, then the node, then the right subtree.
*Post order Traversal (left-right-current):* Visits left subtree, then right subtree, then the node.

**Breadth-First Search (BFS) algorithms:** BFS explores all nodes at the present depth before moving on to nodes at the next depth level. It is typically implemented using a queue. BFS in a binary tree is commonly referred to as Level Order Traversal.

**Code:**

```
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
class Solution:
    def bstFromPreorder(self, preorder: List[int]) -> Optional[TreeNode]:
        if not preorder:
            return None
        root = TreeNode(preorder[0])
        root.left = self.bstFromPreorder([i for i in preorder if i < preorder[0]])
        root.right = self.bstFromPreorder([i for i in preorder if i > preorder[0]])

        return root
```

**Output:**

In-order DFS: 5 3 2 4
Pre-order DFS: 2 3 5 4
Post-order DFS: 5 3 4 2

Level order: 2 3 4 5

**Explanation:**

→ In-order DFS (Left, Root, Right):
Visit 5, then 3, then 2, then 4.
Output: 5 3 2 4

→ Pre-order DFS (Root, Left, Right):
Visit 2, then 3, then 5, then 4.
Output: 2 3 5 4

→ Post-order DFS (Left, Right, Root):
Visit 5, then 3, then 4, then 2.
Output: 5 3 4 2

→ BFS (Level-order):
Visit 2, then 3, then 4, then 5.
Output: 2 3 4 5

→ **Insertion in Binary Tree:**

Inserting elements means add a new node into the binary tree. We would first create a root node in case of empty tree. Then subsequent insertions involve iteratively searching for an empty place at each level of the tree. When an empty left or right child is found then new node is inserted there.

**Code:**

```
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
class Solution:
    def bstFromPreorder(self, preorder: List[int]) -> Optional[TreeNode]:
        if not preorder:
            return None
        root = TreeNode(preorder[0])
        root.left = self.bstFromPreorder([i for i in preorder if i < preorder[0]])
        root.right = self.bstFromPreorder([i for i in preorder if i > preorder[0]])
```

```
        return root
```

**Output:**

Inorder traversal before insertion: 5 3 2 4
Inorder traversal after insertion: 5 3 6 2 4

**Explanation:**

Start with the root node (2).
The queue initially contains [2].
Dequeue 2, and check its children.
Enqueue its children [3, 4].
Dequeue 3, and check its children.
Node 3 has a left child (5) but no right child.
Insert node 6 as the right child of node 3.

→ **Deletion in Binary Tree:**

Deleting a node from a binary tree means removing a specific node while keeping the tree's structure. First, we need to find the node that want to delete by traversing through the tree using any traversal method. Then replace the node's value with the value of the last node in the tree (found by traversing to the rightmost leaf), and then delete that last node.

**Code:**

```
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
class Solution:
    def bstFromPreorder(self, preorder: List[int]) -> Optional[TreeNode]:
        if not preorder:
            return None
        root = TreeNode(preorder[0])
        root.left = self.bstFromPreorder([i for i in preorder if i < preorder[0]])
        root.right = self.bstFromPreorder([i for i in preorder if i > preorder[0]])

        return root
```

**Output:**
Original tree (in-order): 5 3 6 2 4
Tree after deleting 3 (in-order): 5 6 2 4

**Explanation:**

→ Start at the root (node with data 2).

- 2 is not equal to 6, so proceed to check the left and right subtrees.

→ Move to the left child of the root (node with data 3).

- 3 is not equal to 6, so proceed to check the left and right subtrees.

→ Move to the left child of node 3 (node with data 5).

- 5 is not equal to 6, and since 5 is a leaf node, return False.

→ Move to the right child of node 3 (node with data 6).

- 6 is equal to 6, so return True.

## Leet Code Problems

**Problem: Evaluate Boolean Binary Tree**

You are given the root of a **full binary tree** with the following properties:

- **Leaf nodes** have either the value 0 or 1,
  where 0 represents False and 1 represents True.
- **Non-leaf nodes** have either the value 2 or 3, where 2 represents the boolean OR and 3 represents the boolean AND.

The **evaluation** of a node is as follows:

- If the node is a leaf node, the evaluation is the **value** of the node,
  i.e. True or False.
- Otherwise, **evaluate** the node's two children and **apply** the boolean operation of its value with the children's evaluations.

Return *the boolean result of* ***evaluating*** *the* root *node*.

A **full binary tree** is a binary tree where each node has either 0 or 2 children.

A **leaf node** is a node that has zero children.

**Input:** root = [2, 1, 3, null, null, 0, 1]
**Output:** true

**Code:**

```
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
class Solution:
    def bstFromPreorder(self, preorder: List[int]) -> Optional[TreeNode]:
        if not preorder:
            return None
        root = TreeNode(preorder[0])
        root.left = self.bstFromPreorder([i for i in preorder if i < preorder[0]])
        root.right = self.bstFromPreorder([i for i in preorder if i > preorder[0]])

        return root
```

**Output:** true
**Explanation:**

→ root.val is 2, evaluate the left and right subtrees.
→ root.val is 1, return True.
→ root.val is 3, evaluate the left and right subtrees.
→ root.val is 0, return False.
→ root.val is 1, return True.

→ Now, for the node with value 3:

- It represents the logical AND operation (since root.val is 3).
- The left subtree returns False and the right subtree returns True.
- Therefore, False AND True results in False.
-

→Finally, for the root node with value 2:

- It represents the logical OR operation (since root.val is 2).
- The left subtree returns True and the right subtree returns False.
- Therefore, True OR False results in True.

-→ **Time Complexity:** O(n)
-→ **Space Complexity:** O(log n)

**Problem: Root Equals Sum of Children**

You are given the root of a **binary tree** that consists of exactly 3 nodes: the root, its left child, and its right child.
Return true *if the value of the root is equal to the* ***sum*** *of the values of its two children, or* false *otherwise*.

**Input:** root = [10,4,6]
**Output:** true

**Code:**

```
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
class Solution:
    def bstFromPreorder(self, preorder: List[int]) -> Optional[TreeNode]:
        if not preorder:
            return None
        root = TreeNode(preorder[0])
        root.left = self.bstFromPreorder([i for i in preorder if i < preorder[0]])
        root.right = self.bstFromPreorder([i for i in preorder if i > preorder[0]])

        return root
```

**Output:** true

**Explanation:**

- root.val = 10
- root.left.val = 4

- root.right.val = 6

1. The method first computes a = root.left.val + root.right.val = 4 + 6 = 10.
2. Then, it compares a with root.val.
3. Since a (which is 10) is equal to root.val (which is also 10), the method returns True.

-→ **Time Complexity:** O(1)
-→ **Space Complexity:** O(1)

**Problem: Find a Corresponding Node of a Binary Tree in a Clone of That Tree**

Given two binary trees original and cloned and given a reference to a node target in the original tree.
The cloned tree is a **copy of** the original tree.
Return *a reference to the same node* in the cloned tree.
**Note** that you are **not allowed** to change any of the two trees or the target node and the answer **must be** a reference to a node in the cloned tree.

**Input:** tree = [7,4,3,null,null,6,19], target = 3
**Output:** 3

**Code:**

```
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
class Solution:
    def bstFromPreorder(self, preorder: List[int]) -> Optional[TreeNode]:
        if not preorder:
            return None
        root = TreeNode(preorder[0])
        root.left = self.bstFromPreorder([i for i in preorder if i < preorder[0]])
        root.right = self.bstFromPreorder([i for i in preorder if i > preorder[0]])

        return root
```

**Output:** 3

**Explanation:**

→ Start at the root of both original and cloned trees. The root node value is 7, which is not equal to the target 3.
→ Recur to the left subtree. The left child value is 4, which is also not equal to the target 3.
→ Recur to the left child of node 4, which is null. This returns None.
→ Recur to the right child of node 4, which is also null. This returns None.
→ Backtrack to the root node and recur to the right subtree. The right child value is 3, which matches the target 3.
→ Return the corresponding node in the cloned tree with value 3.

-→ **Time Complexity:** O(n)
-→ **Space Complexity:** O(n)

**Problem: Range Sum of BST**
Given the root node of a binary search tree and two integers low and high, return *the sum of values of all nodes with a value in the* ***inclusive*** *range* [low, high].

**Input:** root = [10,5,15,3,7, null,18], low = 7, high = 15
**Output:** 32

**Code:**

```
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
class Solution:
    def bstFromPreorder(self, preorder: List[int]) -> Optional[TreeNode]:
        if not preorder:
            return None
        root = TreeNode(preorder[0])
        root.left = self.bstFromPreorder([i for i in preorder if i < preorder[0]])
        root.right = self.bstFromPreorder([i for i in preorder if i > preorder[0]])
```

```
        return root
```

**Output:** 32

**Explanation:**

→ Node 10:
- 10 is within the range [7, 15], a = 10

→ Left Subtree (Node 5):
- 5 is not within the range [7, 15],

→ Left Subtree of Node 5 (Node 3):
- 3 is not within the range [7, 15]

→Right Subtree of Node 5 (Node 7):
- 7 is within the range [7, 15] → Total for left subtree: 7

→ Right Subtree (Node 15):
- 15 is within the range [7, 15]

→ Right Subtree of Node 15 (Node 18):
- 18 is not within the range [7, 15], →Total for right subtree: 15.

→ Final Sum:
10 + 7 + 15 = 3

-→ **Time Complexity:** O(n)
-→ **Space Complexity:** O(n)

**Problem: Count Nodes Equal to Average of Subtree**

Given the root of a binary tree, return *the number of nodes where the value of the node is equal to the **average** of the values in its **subtree***.

**Note:**
- The **average** of n elements is the **sum** of the n elements divided by n and **rounded down** to the nearest integer.
- A **subtree** of root is a tree consisting of root and all of its descendants.

**Input:** root = [4,8,5,0,1, null,6]
**Output:** 5

**Code:**

```
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
```

```
        self.val = val
        self.left = left
        self.right = right
class Solution:
    def bstFromPreorder(self, preorder: List[int]) -> Optional[TreeNode]:
        if not preorder:
            return None
        root = TreeNode(preorder[0])
        root.left = self.bstFromPreorder([i for i in preorder if i < preorder[0]])
        root.right = self.bstFromPreorder([i for i in preorder if i > preorder[0]])

        return root
```

**Output:** 5

**Explanation:**

→ **Node 0**:

- Subtree sum: 0 (only this node)
- Subtree count: 1 (only this node)
- Average: 0 / 1 = 0

→ **Node 1**:

- Subtree sum: 1 (only this node)
- Subtree count: 1 (only this node)
- Average: 1 / 1 = 1

→ **Node 8**:

- Subtree sum: 8 + 0 + 1 = 9
- Subtree count: 1 (itself) + 1 (left child 0) + 1 (right child 1) = 3
- Average: 9 / 3 = 3

→ **Node 6**:

- Subtree sum: 6 (only this node)
- Subtree count: 1 (only this node)
- Average: 6 / 1 = 6

→ **Node 5**:

- Subtree sum: 5 + 6 = 11
- Subtree count: 1 (itself) + 1 (right child 6) = 2
- Average: 11 / 2 = 5 (integer division)

→**Node 4**:

- Subtree sum: 4 + 9 (left subtree) + 11 (right subtree) = 24
- Subtree count: 1 (itself) + 3 (left subtree) + 2 (right subtree) = 6
- Average: 24 / 6 = 4

-→ **Time Complexity:** O(n)
-→ **Space Complexity:** O(n)

**Problem: Reverse Odd Levels of Binary Tree**

Given the root of a **perfect** binary tree, reverse the node values at each **odd** level of the tree.

- For example, suppose the node values at level 3 are [2,1,3,4,7,11,29,18], then it should become [18,29,11,7,4,3,1,2].

Return *the root of the reversed tree*.

A binary tree is **perfect** if all parent nodes have two children, and all leaves are on the same level.

The **level** of a node is the number of edges along the path between it and the root node.

**Input:** root = [2,3,5,8,13,21,34]

**Output:** [2,5,3,8,13,21,34]

**Code:**

```
# Definition for a binary tree node.
class TreeNode:
```

```
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
class Solution:
    def bstFromPreorder(self, preorder: List[int]) -> Optional[TreeNode]:
        if not preorder:
            return None
        root = TreeNode(preorder[0])
        root.left = self.bstFromPreorder([i for i in preorder if i < preorder[0]])
        root.right = self.bstFromPreorder([i for i in preorder if i > preorder[0]])

        return root
```

**Output:** [2,5,3,8,13,21,34]

**Explanation:**

→ Traversal and Collection:

- Level 0: Node 2 (Even level, no collection)
- Level 1: Nodes 3, 5 (Odd level, collect these nodes)
- Level 2: Nodes 8, 13, 21, 34 (Even level, no collection)

→ Reversing Odd Levels:

- Reverse the values of nodes at Level 1: 3 <-> 5

-→ **Time Complexity:** O(n)
-→ **Space Complexity:** O(n)

## Challenges faced during implementation of Binary Tree

### → Understanding Tree Traversal Techniques

- **Challenge:** Binary tree problems often require specific traversal techniques such as in-order, pre-order, post-order, or level-order. Understanding when and how to use each traversal method is crucial but can be confusing, especially for beginners.

- **Example:** For a problem that requires processing nodes level by level, a level-order traversal (BFS) is necessary, but implementing it correctly with a queue can be challenging.

→ **Handling Recursive Depth and Stack Overflow**

- **Challenge:** Many binary tree problems are best solved using recursion. However, deep trees can lead to a large recursion depth, potentially causing stack overflow errors.
- **Example:** A deeply nested binary tree might exceed the maximum recursion depth in Python, requiring a switch to an iterative solution or using tail-recursion optimization.

→ **Managing Null or Leaf Nodes**

- **Challenge:** Binary trees can have nodes with only one child or none at all (leaf nodes). Handling these cases properly is crucial to avoid null pointer exceptions or incorrect logic.
- **Example:** In problems where you need to swap or compare nodes, overlooking null children can lead to runtime errors or incorrect comparisons.

→ **Balancing Trees**

- **Challenge:** Some problems require you to work with balanced binary trees (e.g., AVL trees), where the height difference between the left and right subtrees is within a certain limit. Maintaining this balance during insertions or deletions can be complex.
- **Example:** Implementing self-balancing trees requires careful adjustment of pointers and heights after each operation, which is prone to errors.

→ **Optimizing for Time and Space Complexity**

- **Challenge:** Binary tree operations can become inefficient, especially with large datasets. Optimizing both time and space complexity while solving

problems like finding the lowest common ancestor or diameter of a tree is often non-trivial.

- **Example:** A naive approach to finding the diameter of a binary tree might involve re-computing heights for each node, leading to an O(n2)O(n2) complexity instead of the optimal O(n)O(n).

→ **Understanding and Implementing Tree Transformations**

- **Challenge:** Some problems involve transforming the tree, such as inverting a binary tree, converting it to a linked list, or flattening it. Implementing these transformations correctly requires a deep understanding of tree structures and careful manipulation of pointers.
- **Example:** Flattening a binary tree to a linked list in-place without using extra space involves manipulating the tree's structure directly, which can be error-prone and requires careful attention to the tree's left and right pointers.