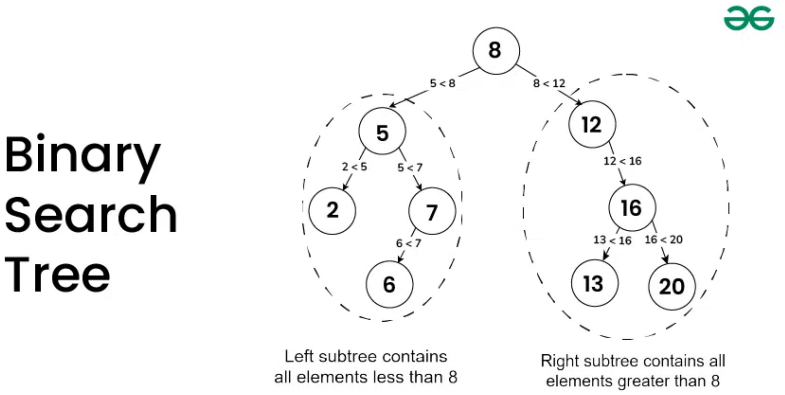
BINARY SEARCH TREES (BST)

Binary Search Tree (BST) is a special type of binary tree in which the left child of a node has a value less than the node's value and the right child has a value greater than the node's value. This property is called the BST property, and it makes it possible to efficiently search, insert, and delete elements in the tree.



Properties of Binary Search Tree:
→ The left subtree of a node contains only nodes with keys lesser than the node's key.
→ The right subtree of a node contains only nodes with keys greater than the node's key.
→ The left and right subtree each must also be a binary search tree.
→There must be no duplicate nodes (BST may have duplicate values with different handling approaches).

Basic Operations on Binary Search Tree:

→ Searching a Node in BST:
Searching in BST means to locate a specific node in the data structure. In Binary search tree, searching a node is easy because of its a specific order.

- First, compare the element to be searched with the root element of the tree.
- Repeat the above procedure recursively until the match is found.
- If the element is not found or not present in the tree, then return NULL.

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:
Not Found
Found

→ Insert a Node into a BST:

A new key is always inserted at the leaf. Start searching a key from the root till a leaf node. Once a leaf node is found, the new node is added as a child of the leaf 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: 20 30 40 50 60 70 80

→ Delete a Node from a BST:

It is used to delete a node with specific key from the BST and return the new BST.

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 10 12 18

→ **Traversal in BST:**

In case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order. We visit the left child first, then the root, and then the right child.

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 10 12 15 18

Leet Code Problems

Problem: Convert Sorted Array to Binary Search Tree

Given an integer array nums where the elements are sorted in ascending order, convert it to a height-balanced binary search tree.

Input: nums = [-10, -3,0,5,9]
Output: [0, -3,9, -10, null, 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: [0, -3,9, -10, null, 5]

Explanation:

→ s = 0, e = 4, mid = (0 + 4) // 2 = 2

→ Call sorted(nums, 0, 1) to build the left subtree.

s = 0, e = 1, mid = (0 + 1) // 2 = 0

→Call sorted(nums, 0, -1) to build the left subtree of -10.

s = 0, e = -1 returns None (base case).

→ Call sorted(nums, 1, 1) to build the right subtree of -10.

s = 1, e = 1, mid = (1 + 1) // 2 = 1

→ Call sorted(nums, 1, 0) to build the left subtree of -3.

s = 1, e = 0 returns None (base case).

→ Call sorted(nums, 2, 1) to build the right subtree of -3.

s = 2, e = 1 returns None (base case).

→ Call sorted(nums, 3, 4) to build the right subtree.

s = 3, e = 4, mid = (3 + 4) // 2 = 3

→ Call sorted(nums, 3, 2) to build the left subtree of 5.

s = 3, e = 2 returns None (base case).

→ Call sorted(nums, 4, 4) to build the right subtree of 5.

s = 4, e = 4, mid = (4 + 4) // 2 = 4

→ Call sorted(nums, 4, 3) to build the left subtree of 9.

s = 4, e = 3 returns None (base case).

→ Call sorted(nums, 5, 4) to build the right subtree of 9.

s = 5, e = 4 returns None (base case).

-→ **Time Complexity:** O(n)

-→ **Space Complexity:** O(n)

Problem: Search in a Binary Search Tree

You are given the root of a binary search tree (BST) and an integer val.

Find the node in the BST that the node's value equals Val and return the subtree rooted with that node. If such a node does not exist, return null.

Input: root = [4,2,7,1,3], val = 2

Output: [2,1,3]

Code:

```
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, 1, 3]

Explanation:

→**Initial Call**:

- searchBST(root, 2) is called with root having value 4.
- root.val = 4 which is greater than val = 2.
- Since the value 4 is greater than 2, the function recursively searches the left subtree.

→ **Second Call**:

- searchBST(root.left, 2) is called with root.left having value 2.
- root.val = 2 which is equal to val = 2.
- Since the value 2 is found, the function returns the root node which has value 2.

→ **Return Value**:

- The node returned has the value 2 with its left child 1 and right child 3.

-→ **Time Complexity:** O(n)

-→ **Space Complexity:** O(n)

Problem: Increasing Order Search Tree

Given the root of a binary search tree, rearrange the tree in in-order so that the leftmost node in the tree is now the root of the tree, and every node has no left child and only one right child.

Input: root = [5,1,7]
Output: [1,null,5,null,7]

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: [1, null,5, null,7]

Explanation:

1. Start at the root node 5.
2. Traverse to the left child 1 (left subtree).
3. Since 1 has no left child, add 1 to ans and move back to 5.
4. Add 5 to ans and move to the right child 7 (right subtree).
5. Since 7 has no left child, add 7 to ans.

After the inorder traversal, ans contains the nodes in this order: [1, 5, 7].

Reconstruction of Tree

1. Start rearranging pointers:

- ans[0] (node 1): Set left to None and right to ans[1] (node 5).
- ans[1] (node 5): Set left to None and right to ans[2] (node 7).
- ans[2] (node 7): Set left to None and right to None.

-→ Time Complexity: O(n)

-→ Space Complexity: O(n)

Problem: Two Sum IV - Input is a BST

Given the root of a binary search tree and an integer k, return true if there exist two elements in the BST such that their sum is equal to k, or false otherwise.

Input: root = [5,3,6,2,4,null,7], k = 9
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:

Initialize i = 0 (pointing to 2) and j = 5 (pointing to 7).
Sum = tree[0] + tree[5] = 2 + 7 = 9
Since 9 equals the target k, the function returns True.

-→ **Time Complexity:** O(n)

-→ **Space Complexity:** O(n)

Problem: Construct Binary Search Tree from Preorder Traversal

Given an array of integers preorder, which represents the preorder traversal of a BST (i.e., binary search tree), construct the tree and return its root.

It is guaranteed that there is always possible to find a binary search tree with the given requirements for the given test cases.

A binary search tree is a binary tree where for every node, any descendant of Node.left has a value strictly less than Node.val, and any descendant of Node.right has a value strictly greater than Node.val.

A preorder traversal of a binary tree displays the value of the node first, then traverses Node.left, then traverses Node.right.

Input: preorder = [8,5,1,7,10,12]
Output: [8,5,10,1,7,null,12]

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: [8,5,10,1,7, null,12]

Explanation:

→ **First Call** (preorder = [8, 5, 1, 7, 10, 12]):
- root.val = 8
- Left subtree: [5, 1, 7] (all elements < 8)
- Right subtree: [10, 12] (all elements > 8)

→ **Left Subtree of 8** (preorder = [5, 1, 7]):
- root.val = 5
- Left subtree: [1] (all elements < 5)
- Right subtree: [7] (all elements > 5)

→ **Left Subtree of 5** (preorder = [1]):
- root.val = 1
- No left or right subtree as the list is exhausted.

→ **Right Subtree of 5** (preorder = [7]):
- root.val = 7
- No left or right subtree as the list is exhausted.

→ **Right Subtree of 8** (preorder = [10, 12]):
- root.val = 10
- Left subtree: [] (all elements < 10)
- Right subtree: [12] (all elements > 10)

→ **Right Subtree of 10** (preorder = [12]):
- root.val = 12

-→ **Time Complexity:** $O(n^2)$

-→ **Space Complexity:** $O(n^2)$

Problem: Balance a Binary Search Tree

Given the root of a binary search tree, return a balanced binary search tree with the same node values. If there is more than one answer, return any of them.

A binary search tree is balanced if the depth of the two subtrees of every node never differs by more than 1.

Input: root = [1,null,2,null,3,null,4,null,null]

Output: [2,1,3,null,null,null,4]

Code:

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
class Solution:
    def balanceBST(self, root: TreeNode) -> TreeNode:
        arr=[]
        def inorder(node):
            if(not node):
                return

            inorder(node.left)
            arr.append(node.val)
            inorder(node.right)
            return arr
        def buildbst(l):
            if(not l):
                return None
            mid=len(l)//2
            root = TreeNode(l[mid])
            root.left = buildbst(l[:mid])
            root.right= buildbst(l[mid+1:])
            return root
        l=inorder(root)
        print(l)
        return buildbst(l)
```

Output: [2, 1, 3, null, null, null, 4]

Explanation:

→ Inorder Traversal (inorder function):

    - Visit 1 -> Add to arr: [1]
    - Visit 2 -> Add to arr: [1, 2]
    - Visit 3 -> Add to arr: [1, 2, 3]
    - Visit 4 -> Add to arr: [1, 2, 3, 4]
- The result of inorder traversal is l = [1, 2, 3, 4].

→ Building the Balanced BST

- The build bst function constructs a balanced binary search tree from the array [1, 2, 3, 4]:
    - Choose the middle element 2 as the root.
    - Recursively do the same for the left subarray [1] and the right subarray [3, 4].
    - The constructed balanced BST:

-→ **Time Complexity:** O(n)

-→ **Space Complexity:** O(n)

Challenges faced during implementation of Binary Search Tree

→ Handling Edge Cases

- **Challenge:** BST problems often include edge cases such as empty trees, single-node trees, or trees where all nodes are either to the left or right. These edge cases can lead to incorrect outputs if not handled properly.
- **Solution:** Always consider and test for edge cases when designing your solution. Write specific test cases to cover scenarios like empty trees, skewed trees, or trees with duplicate values (if applicable).

→ Recursive vs. Iterative Approaches

- **Challenge:** Many BST problems can be solved using either recursion or iteration. While recursion is often more intuitive, it can lead to stack overflow issues on deep trees, whereas iterative solutions can be more complex to implement.
- **Solution:** Choose the approach based on the problem requirements and constraints. For problems with deep trees, consider using iterative methods

to avoid stack overflow. Alternatively, use tail recursion optimization if your language supports it.

→ Balancing the BST

- **Challenge:** In some problems, maintaining a balanced BST is crucial for optimal performance. Without balancing, the tree can degenerate into a linked list, resulting in O(n)O(n) operations instead of O(log n)O(log n).
- **Solution:** Implement self-balancing techniques, such as AVL trees or Red-Black trees, if the problem requires maintaining balance. For simpler problems, ensure the input data is structured to maintain balance naturally.

→ In-Place Modifications

- **Challenge:** Some problems require modifying the BST in place (e.g., inserting or deleting nodes) while maintaining the BST properties. This can be tricky, especially when dealing with node deletions where you need to manage the replacement of nodes.
- **Solution:** Understand the specific rules for insertion, deletion, and balancing in a BST. Practice these operations to ensure you can implement them correctly in a problem-solving scenario.

→ Optimizing for Time and Space Complexity

- **Challenge:** Ensuring that your solution is both time and space-efficient can be difficult, especially with large datasets. Recursive solutions can lead to higher space complexity due to the call stack.
- **Solution:** Analyze the time and space complexity of your solution upfront. If the space complexity is high, consider converting recursive solutions to iterative ones or using memorization techniques to optimize performance.

→ Traversal Techniques

- **Challenge:** Different problems may require different traversal methods (in-order, pre-order, post-order, level-order) to solve correctly. Choosing the wrong traversal technique can lead to incorrect solutions.
- **Solution:** Understand the properties of each traversal technique and their use cases. For example, in-order traversal is useful for BST problems where elements need to be processed in sorted order. Practice identifying the appropriate traversal method for a given problem.