PW SKILLS

Lesson Plan

BST-1

PW SKILLS

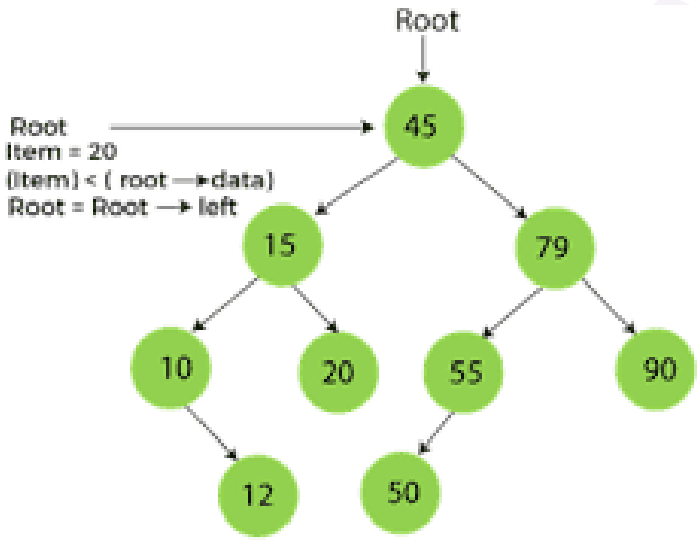
Prerequisites:

- Understanding of basic data structures like arrays and linked lists.
- Understanding of recursion.
- Proficiency in a C++ programming language

Today's Checklist:

- Introduction to Binary Search Tree
- Search in a BST [Leetcode-700]
- Insert into a BST [Leetcode-701]
- Traversal in BST (Preorder, Inorder, Postorder)
- Lowest Common Ancestor of a BST [Leetcode-235]
- Validate BST [Leetcode-98]
- Binary Search Tree to Greater Sum Tree [Leetcode-1038]
- Convert Sorted Array to Balanced BST [Leetcode-108]
- Construct BST from preorder traversal [Leetcode-1008]

Introduction to Binary Search Tree



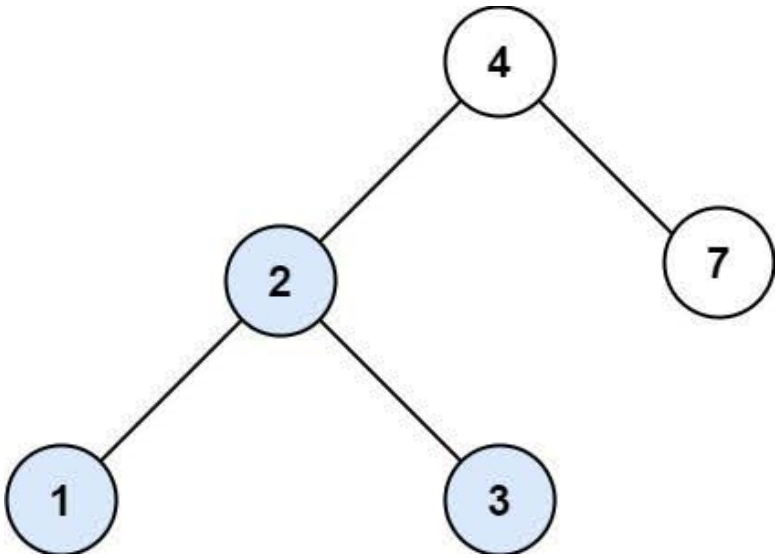
Binary Search Tree is a node-based binary tree data structure which has the following properties:

- 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.

Search in a Binary Search Tree [Leetcode-700]

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.
Search in a Binary Search Tree

Example 1:



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

Code:

```
class Solution {
public:
    TreeNode* searchBST(TreeNode* root, int val) {
        // Base case: If the root is NULL, the tree is empty, or the
value is not found.
        // Return NULL to indicate that the value is not present in
the tree.
        if (root == NULL) {
            return NULL;
        }

        if (root->val == val) {
            return root;
        }
        // If the target value is smaller, search in the left
subtree.
        else if (root->val > val) {
            // Make a recursive call to searchBST with the left
subtree as the new root.
            return searchBST(root->left, val);
        }

        else {

            return searchBST(root->right, val);
        }
    }
};
```

Explanation:

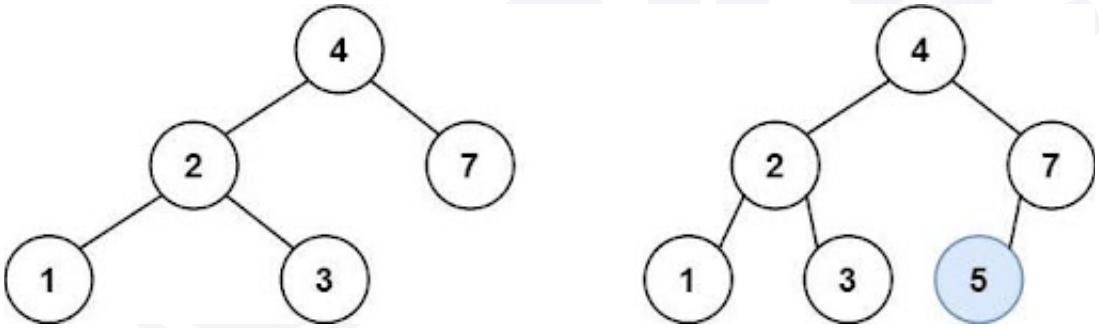
- The code defines a class named Solution.
- The class has a public member function searchBST that takes a TreeNode* (root of a binary search tree) and an integer val as parameters.
- If the current node is NULL (empty tree) or the target value is not found, the function returns NULL.
- If the current node's value matches the target value, the function returns the current node.
- If the target value is smaller than the current node's value, the function recursively calls itself with the left subtree as the new root.
- If the target value is larger, the function recursively calls itself with the right subtree as the new root.

Time Complexity: O(h) in the average case, O(N) in the worst case, where h is the height of the binary search tree and N is the number of nodes.

Space Complexity: O(h) in the average case, O(N) in the worst case, where h is the height of the binary search tree and N is the number of nodes. This accounts for the recursion stack during the recursive calls.

Insert into a Binary Search Tree [Leetcode-701]

You are given the root node of a binary search tree (BST) and a value to insert into the tree. Return the root node of the BST after the insertion. It is guaranteed that the new value does not exist in the original BST. Notice that there may exist multiple valid ways for the insertion, as long as the tree remains a BST after insertion. You can return any of them.



Input: root = [4,2,7,1,3], val = 5
Output: [4,2,7,1,3,5]
Explanation: Another accepted tree is:

Code:

```
class Solution {
public:
    TreeNode* insertIntoBST(TreeNode* root, int val) {
        if(root==NULL) return new TreeNode(val);
        if(root->val > val) root->left = insertIntoBST(root->left,
val);
        else  root->right = insertIntoBST(root->right, val);
        return root;
    }
};
```

Explanation:

- The function insertIntoBST inserts a new node with value val into a binary search tree.
- If the root is null, it creates a new node with value val and returns it.
- If the value to be inserted (val) is less than the root's value, it recursively calls the function on the left subtree.
- If the value to be inserted is greater than or equal to the root's value, it recursively calls the function on the right subtree.
- Finally, it returns the root of the modified binary search tree.

Time Complexity:
Average Case: O(log n) - when the binary search tree is balanced.
Worst Case: O(n) - when the binary search tree is skewed.

Space Complexity:
Average Case: O(log n) - due to the recursion stack.
Worst Case: O(n) - when the binary search tree is skewed.

Preorder traversal of a BST

Code:

```
#include <iostream>

// Definition for a binary tree node
struct TreeNode {
    int data;
    TreeNode* left;
    TreeNode* right;

    TreeNode(int val) : data(val), left(nullptr), right(nullptr) {}
};

// Function to perform preorder traversal of a BST
void preorderTraversal(TreeNode* root) {
    if (root == nullptr) {
        return;
    }

    // Process the current node
    std::cout << root->data << " ";

    // Recursively traverse the left subtree
    preorderTraversal(root->left);

    // Recursively traverse the right subtree
    preorderTraversal(root->right);
}
```

Explanation:

- TreeNode struct: Represents a node in a binary tree, containing data, a pointer to the left child, and a pointer to the right child.
- preorderTraversal function: Performs a preorder traversal of the binary tree, printing each node's data as it is visited.
- Calls preorderTraversal to print the nodes in preorder.
- Deletes allocated memory for proper cleanup.

Time Complexity: O(N) (linear) where N is the number of nodes.

Space Complexity: O(H) (height of the tree), worst-case O(N) for a skewed tree, best-case O(log N) for a balanced tree.

Inorder traversal of a BST

Code:

```
#include <iostream>

// Definition for a binary tree node.
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};


void inorderTraversal(TreeNode* root) {
    if (root == nullptr) {
        return;
    }

    inorderTraversal(root->left);

    std::cout << root->val << " ";

       inorderTraversal(root->right);
}
```

Explanation:

- The code defines a TreeNode structure to represent nodes in the Binary Search Tree (BST).
- The inorderTraversal function recursively traverses the BST in inorder (left subtree, current node, right subtree) and prints the values.
- The main function creates a sample BST, invokes inorderTraversal, and prints the elements in ascending order.

Time Complexity:
The time complexity of the inorder traversal in a Binary Search Tree is O(n), where n is the number of nodes in the tree.
This is because each node is visited once during the traversal.

Space Complexity:
The space complexity is O(h), where h is the height of the BST.
In the worst case, if the tree is skewed (all nodes have only one child), the space complexity becomes O(n).
In the average case for a balanced BST, the space complexity is O(log n), where log n is the height of a balanced tree.

Postorder traversal of a BST

Code:

```
#include <iostream>

// Node structure for BST
struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int value) : data(value), left(nullptr), right(nullptr) {}
};

// Function to perform postorder traversal of BST
void postorderTraversal(Node* root) {
    if (root == nullptr) {
        return;
    }

    // Traverse the left subtree
    postorderTraversal(root->left);

    // Traverse the right subtree
    postorderTraversal(root->right);

    // Visit the current node (print the data, for example)
    std::cout << root→data << " ";
}
```

Explanation:

- Node Structure: Defines a structure for a Binary Search Tree (BST) node with integer data, left, and right pointers.
- postorderTraversal Function: Recursively traverses BST in postorder (left subtree, right subtree, root), printing node data.
- main Function: Creates a sample BST, performs postorder traversal, and prints the result.
- Memory Cleanup (Optional): Properly frees allocated memory for the sample BST in the main function.

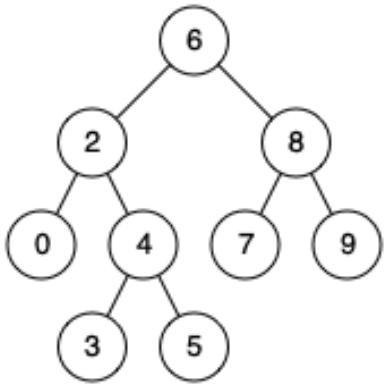
Time Complexity: O(n) - Linear time, as each node is visited exactly once during postorder traversal.

Space Complexity:
Recursive Stack Space: O(h), where 'h' is the height of the BST. Worst case O(n), average case O(log n).
Additional Space: O(1), constant space for variables and pointers during traversal.

Lowest Common Ancestor of a Binary Search Tree [Leetcode-235]

Given a binary search tree (BST), find the lowest common ancestor (LCA) node of two given nodes in the BST.
According to the definition of LCA on Wikipedia: “The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow a node to be a descendant of itself).”

Example 1:



Input: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8
Output: 6
Explanation: The LCA of nodes 2 and 8 is 6.

Code:

```
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q)
{
            if(root==NULL) return NULL;
             int cur = root->val;
           if(cur<p->val && cur<q->val){ //Both on the right
              return lowestCommonAncestor(root->right,p,q);
           }
           if(cur>p->val && cur>q->val){ //Both on the left
              return lowestCommonAncestor(root->left,p,q);
           }
           //Cannot determine so this is the intersection
            return root;

    }
};
```

Explanation:

- The code defines a Solution class with a function lowestCommonAncestor that finds the lowest common ancestor of two nodes in a binary search tree.
- The function recursively traverses the tree, comparing values of the current node (cur) with the values of the two given nodes (p and q).
- If both nodes are on the right, the function recursively calls itself with the right subtree.
- If both nodes are on the left, the function recursively calls itself with the left subtree.
- If the current node's value is between the values of p and q, it is the lowest common ancestor, and the function returns the current node.

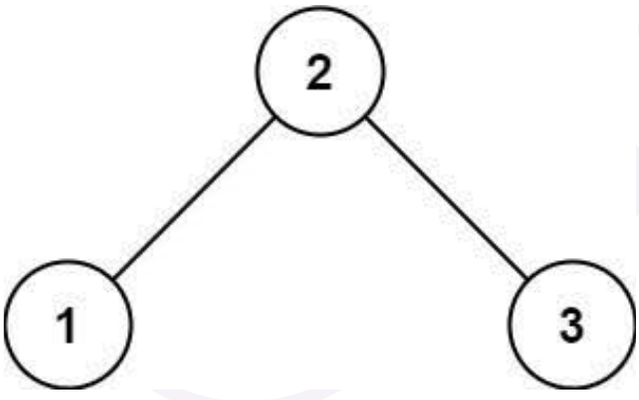
Time Complexity: O(log N) on average, O(N) in the worst case (unbalanced tree).

Space Complexity: O(log N) on average, O(N) in the worst case (due to recursive call stack).

Validate Binary Search Tree [Leetcode-98]

Given the root of a binary tree, determine if it is a valid binary search tree (BST).
A valid BST is defined as follows:
The left subtree of a node contains only nodes with keys less than the node's key.
The right subtree of a node contains only nodes with keys greater than the node's key.
Both the left and right subtrees must also be binary search trees.

Example 1:



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

Code:

```
class Solution {

bool isPossible(TreeNode* root, long long l, long long r){
    if(root == nullptr)  return true;
    if(root->val < r and root->val > l)
        return isPossible(root->left, l, root->val) and
                                isPossible(root->right, root->val, r);
    else return false;
}

public:
    bool isValidBST(TreeNode* root) {
        long long int min = -1000000000000, max = 1000000000000;
        return isPossible(root, min, max);
    }
};
```

Explanation:

- The code defines a class named Solution.
- It contains a private helper function isPossible to recursively check if the given root node and its descendants form a valid binary search tree (BST) within the specified range [l, r].
- The isValidBST function initializes the range with minimum and maximum values and calls the isPossible function to determine if the binary tree rooted at root is a valid BST.
- The code utilizes a long long data type for the range to handle large integer values.
- The function returns true if the tree is a valid BST, and false otherwise.

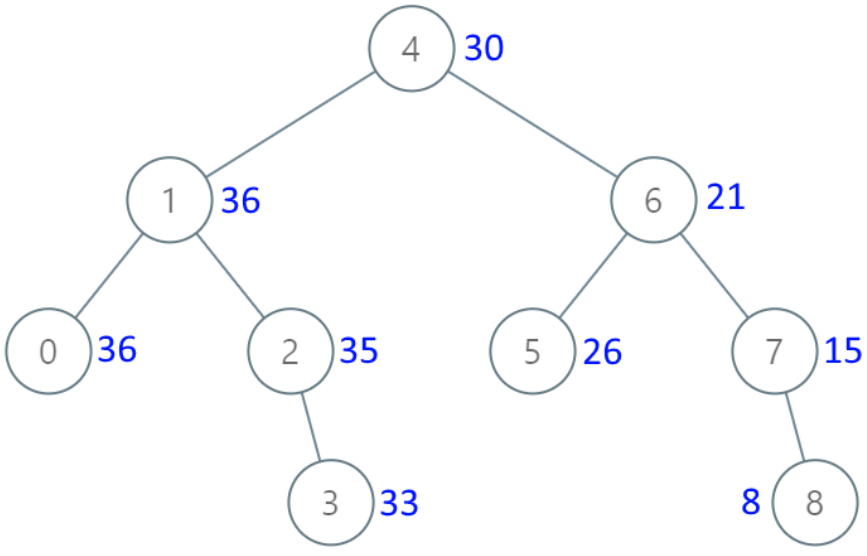
Time Complexity:
The time complexity of the isValidBST function is O(N), where N is the number of nodes in the binary tree. This is because the function performs a recursive traversal of each node exactly once.

Space Complexity:
The space complexity is O(H), where H is the height of the binary tree.
The space complexity is determined by the depth of the recursion stack.
In the worst case, when the tree is skewed (like a linked list), the height H can be equal to the number of nodes N, resulting in O(N) space complexity.

Binary Search Tree to Greater Sum Tree [Leetcode-1038]

Given the root of a Binary Search Tree (BST), convert it to a Greater Tree such that every key of the original BST is changed to the original key plus the sum of all keys greater than the original key in BST.
As a reminder, a binary search tree is a tree that satisfies these constraints:
The left subtree of a node contains only nodes with keys less than the node's key.
The right subtree of a node contains only nodes with keys greater than the node's key.
Both the left and right subtrees must also be binary search trees.

Example 1:



Input: root = [4,1,6,0,2,5,7,null,null,null,3,null,null,null,8]
Output: [30,36,21,36,35,26,15,null,null,null,33,null,null,null,8]

Code:

```
class Solution {
public:
    void convertTree(TreeNode* &root, int &sum) {
        if (root == NULL) return;

        convertTree(root->left, sum);
        int currNodeVal = root->val;
        root->val = sum;
        sum-=currNodeVal;
        convertTree(root->right, sum);
    }

    int getSum(TreeNode* root) {
        if (root == NULL) return 0;

        return getSum(root->left) + root->val + getSum(root->right);
    }


    TreeNode* bstToGst(TreeNode* root) {
        int nodeSum = getSum(root);

        convertTree(root, nodeSum);
        return root;
    }
};
```

Explanation:

- The code defines a class Solution with three member functions for converting a Binary Search Tree (BST) to Greater Sum Tree (GST).
- convertTree recursively converts the BST to GST by updating each node's value to the sum of all greater values in the tree.
- getSum recursively calculates the sum of all node values in the tree.
- bstToGst uses getSum to get the total sum of the BST, then applies convertTree to modify the tree in-place, and finally returns the modified root.
- The conversion is achieved by traversing the BST in reverse in-order (right, root, left) and updating each node's value accordingly.

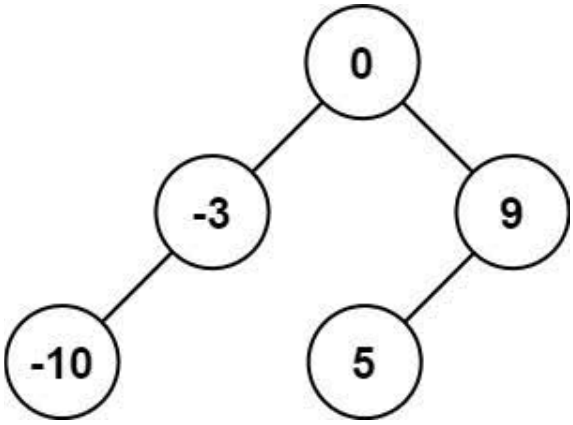
Time Complexity:
O(n) where n is the number of nodes in the tree.

Space Complexity:
O(n) in the worst case (skewed tree), O(log(n)) in the average case for a balanced BST.

Convert Sorted Array to Balanced BST [Leetcode-108]

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

Example 1:



Input: nums = [-10,-3,0,5,9]
Output: [0,-3,9,-10,null,5]
Explanation: [0,-10,5,null,-3,null,9] is also accepted:

Code:

```
class Solution {
 public:
     TreeNode* sortedArrayToBST(vector<int>& nums) {
         // We have to create a hight balanced BST
         // elements are strictly increasing
         if(nums.size()==0)return NULL;
         if(nums.size()==1) return new TreeNode(nums[0]);
         int middle = nums.size()/2;
         TreeNode* root = new TreeNode(nums[middle]);
         vector<int> leftsub(nums.begin(), nums.begin()+middle);
         vector<int> rightsub(nums.begin()+ middle+1, nums.end());
         root->left = sortedArrayToBST(leftsub);
         root->right = sortedArrayToBST(rightsub);
         return root;
     }
 };
```

Explanation:

- Takes a sorted array (nums) and returns the root of a height-balanced Binary Search Tree (BST).
- If the array is empty, returns NULL.
- If the array has only one element, creates a new TreeNode with that element and returns it.
- Finds the middle index of the array and creates a new TreeNode with the corresponding element as the root.
- Recursively constructs left and right subtrees using subarrays split at the middle index.
- Assigns the left and right subtrees as the left and right children of the root, respectively.
- Returns the root of the constructed BST.

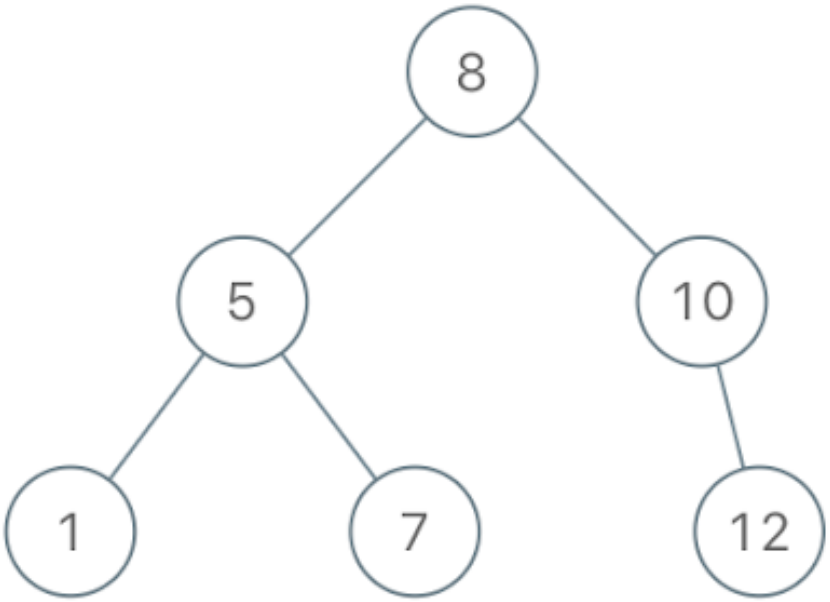
Time Complexity:
The time complexity is O(N), where N is the number of elements in the input sorted array.
Each element in the array is processed once, and for each element, a constant amount of work is done.

Space Complexity:
The space complexity is O(N) due to the recursive call stack.
In each recursive call, new vectors (leftsub and rightsub) are created, each containing roughly half of the elements of the current array.
The maximum depth of the recursion is log(N) for a balanced tree, and at each level, O(N) space is used for the vectors.

Construct Binary Search Tree from Preorder Traversal [Leetcode-1008]

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.

Example 1:



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

Code:

```
class Solution {
public:
    TreeNode* bstFromPreorder(vector<int>& preorder) {
    vector<int>inorder, v;
    v=preorder;
    sort(v.begin(), v.end());
    inorder=v;
    if(preorder.size()==0){return NULL;}
    map<int, int>m;
    for(int i=0; i<inorder.size(); i++){
        m[inorder[i]]=i;
    }
    TreeNode* root=formtree(preorder, 0, preorder.size()-1, inorder, 0,
inorder.size()-1, m);
    return root;
}
TreeNode* formtree(vector<int>& preorder, int pst, int pen, vector<int>& inorder,
int inst, int inen, map<int, int>& m){
    if(pst>pen || inst>inen){return NULL;}
    TreeNode* root=new TreeNode(preorder[pst]);
    int pos=m[preorder[pst]];
    int lft=pos-inst;
    root->left=formtree(preorder, pst+1, pst+lft, inorder, inst, pos-1, m);
    root->right=formtree(preorder, pst+lft+1, pen, inorder, pos+1, inen, m);
    return root;
}
};
```

Explanation:

- The code defines a Solution class with a public function bstFromPreorder that takes a vector of integers representing the preorder traversal of a binary search tree (BST) and returns the root of the BST.
- It sorts the preorder vector to get the inorder traversal, which helps in constructing the BST.
- It checks if the preorder vector is empty, and if so, returns NULL.
- It creates a map m to store the positions of elements in the inorder traversal.
- The formtree function recursively constructs the BST using the preorder and inorder traversals, updating the root, left, and right subtrees accordingly.
- The base case of recursion is when the start position is greater than the end position or the start position in the inorder traversal is greater than the end position, in which case it returns NULL.
- The constructed root is returned as the result.

Time Complexity:
Sorting: O(n log n)
Constructing Inorder: O(n)
Recursive Construction: O(n)

Space Complexity:
Sorting and Inorder Vectors: O(n)
Recursion Call Stack: O(n)
Map for Inorder Positions: O(n)

PW SKILLS

THANK YOU !

PW SKILLS