PW SKILLS

Lesson Plan

Binary tree-3

PW SKILLS

Prerequisites:

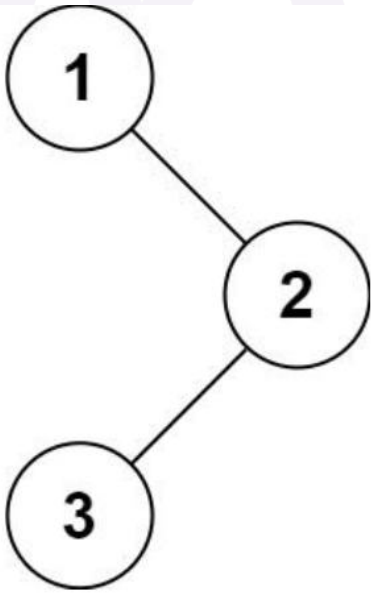
- Understanding of basic data structures like arrays and linked lists.
- Understanding of recursion.
- Proficiency in a C++ programming language

Today's Checklist:

- Traversal - Iterative (Preorder, Inorder, Postorder) [Leetcode-144,94,145]
- Boundary Traversal
- Binary Tree Right Side View [Leetcode-199]
- Path Sum-II [Leetcode-113]
- Path Sum-III [Leetcode-437]
- Construct Binary Tree from Preorder and Inorder Traversal [Leetcode-105]
- Preorder Traversal In Tree [Leetcode-144] (Iterative)

Preorder Traversal In Tree [Leetcode-144] (Iterative)

Given the root of a binary tree, return the preorder traversal of its nodes' values.



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

Example 2:
Input: root = []
Output: []

Code:

```
class Solution {
public:

    vector<int> preorderTraversal(TreeNode* root) {

        // res will store the preorder traversal of the binary
tree

        vector<int> res;

        // declare a stack

        stack<TreeNode*> st;

        TreeNode* curr = root;

        while(!st.empty() || curr)
        {
            // push the value in the res and curr node into stack
and move the curr to the left

            while(curr)
            {
                res.push_back(curr -> val);

                st.push(curr);

                curr = curr -> left;
            }

            // move the curr to the right

            if(st.empty() == false)
            {
                curr = st.top();

                st.pop();

  curr = curr -> right;
            }
        }

        return res;
    }
};
```

Explanation:

- Purpose: Return the preorder traversal of a binary tree.
- Data Structures: Stack (st), Vector (res).
- Traverse left subtree, push nodes and add values to res.
- Move to the right subtree by popping from the stack.
- Iteration: Loop until stack is empty or curr is null.
- Termination: Ends when all nodes are processed.
- Return Value: Vector res containing preorder traversal.

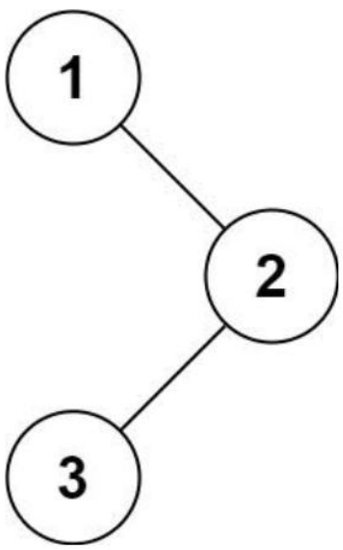
Time Complexity:
O(N), where N is the number of nodes in the binary tree.
Each node is visited once, and the while loop iterates through all nodes.

Space Complexity:
O(H), where H is the height of the binary tree.
The space used by the stack is proportional to the height of the tree.
In the worst case (skewed tree), the height approaches N, resulting in O(N) space complexity.
In a balanced tree, the height is log(N), leading to O(log N) space complexity.

Inorder Traversal In Tree [Leetcode-94] (Iterative)

Given the root of a binary tree, return the inorder traversal of its nodes' values.

Example 1:



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

Example 2:

Input: root = []
Output: []

Example 3:

Input: root = [1]
Output: [1]

Code:

```
class Solution {
public:
    vector<int> inorderTraversal(TreeNode* root) {
        // l n r

        vector<int> traversal;
        if (root==NULL)return traversal;
        stack<TreeNode*> stk;
        TreeNode* node= root;
        while (true){
            if (node!= NULL){

  stk.push(node);
                node=node->left;
            }
            else{
                if (stk.empty())break;
                node= stk.top();
                stk.pop();
                traversal.push_back(node->val);
                node=node->right;
            }
        }
        return traversal;
    }
};
```

Explanation:

- The code performs an in-order traversal of a binary tree iteratively using a stack.
- It initializes an empty vector to store the traversal result.
- It uses a stack to keep track of nodes and iterates until the stack is empty.
- It pushes left children onto the stack until reaching the leftmost node.
- When there are no more left children, it pops a node, adds its value to the result vector, and moves to the right child.
- The process continues until all nodes are processed, and the final in-order traversal is returned.

Time Complexity: O(N), where N is the number of nodes in the binary tree.
Each node is visited once, and the while loop runs in O(1) time for each node.

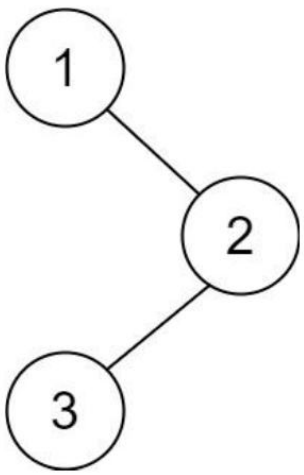
Space Complexity: O(H), where H is the height of the binary tree, and in the worst case, H can be equal to N in a skewed tree.

The space is used for the stack, and in the worst case, it can be as large as the height of the tree.
For a balanced tree, the space complexity is O(log N), as the maximum height is logarithmic in the number of nodes.

Postorder Traversal In Tree (Iterative) [Leetcode-145]

Given the root of a binary tree, return the postorder traversal of its nodes' values.

Example 1:



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

Example 2:

Input: root = []
Output: []

Example 3:

Input: root = [1]
Output: [1]

Code:

```
#include <iostream>
#include <vector>
#include <stack>

using namespace std;

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

```
vector<int> postorderTraversal(TreeNode* root) {
    vector<int> result;
    stack<TreeNode*> st;
    TreeNode* prev = nullptr;

    while (root ≠ nullptr || !st.empty()) {
        while (root ≠ nullptr) {
            st.push(root);
            root = root→left;
        }

        root = st.top();

        // If the right subtree is empty or has been processed
        if (root→right == nullptr || root→right == prev) {
            result.push_back(root→val);
            st.pop();
            prev = root;
            root = nullptr;  // Set to nullptr to avoid
revisiting the left subtree
        } else {
            root = root→right;
        }
    }

    return result;
}
```

Explanation:

- The code performs iterative postorder traversal of a binary tree using a stack.
- It maintains a prev pointer to track the previously processed node.
- The outer loop handles stack processing until both the current node and stack are empty.
- The inner loop pushes left children onto the stack and processes nodes when right subtree is empty or processed.
- Nodes are added to the result vector, and prev is updated to avoid revisiting the left subtree.
- The function returns a vector representing the postorder traversal.

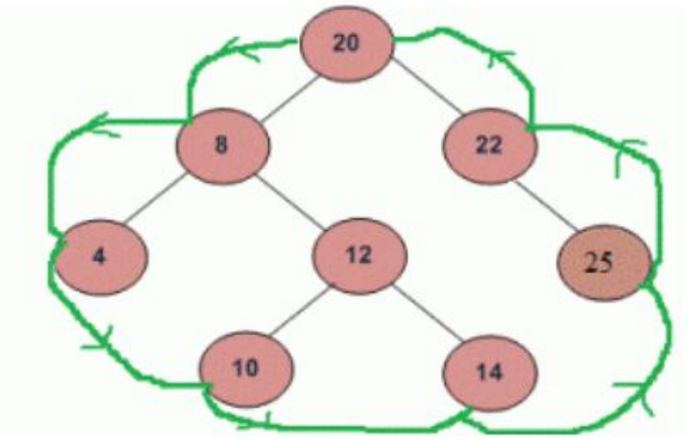
Time Complexity:
The time complexity of the provided code is O(n), where n is the number of nodes in the binary tree.Each node is visited once, and for each visit, constant time operations are performed.

Space Complexity:
The space complexity is O(h), where h is the height of the binary tree.
In the worst case, the stack can have at most h nodes, where h is the height of the binary tree.
In the average case, for a balanced binary tree, the space complexity is O(log n), where n is the number of nodes.

Boundary Traversal

Given a binary tree, print boundary nodes of the binary tree Anti-Clockwise starting from the root.

The boundary includes
left boundary (nodes on left excluding leaf nodes)
leaves (consist of only the leaf nodes)
right boundary (nodes on right excluding leaf nodes)



Code:

```
#include <bits/stdc++.h>
using namespace std;

struct Node {
    int data;
    struct Node *left, *right;
};

Node* newNode(int data

{
    Node* temp = new Node;
    temp->data = data;
    temp->left = temp->right = nullptr;
    return temp;
}

bool isLeaf(Node* node){
  if(node->left == NULL && node->right==NULL){
    return true;
  }
  return false;
}

void addLeftBound(Node * root, vector<int>& ans){
  //Go left left and no left then right but again check from left
  root = root->left;
  while(root){
    if(!isLeaf(root)) ans.push_back(root->data);
    if(root->left) root = root->left;
    else root = root->right;
  }
}
```

```
void addRightBound(Node * root, vector<int>& ans){
  //Go right right and no right then left but again check from
right
  root = root→right;
  stack<int> stk;
  while(root){
    if(!isLeaf(root)) stk.push(root→data);
    if(root→right) root = root→right;
    else root = root→left;
  }
  while(!stk.empty()){
    ans.push_back(stk.top());
    stk.pop();
  }
}

//its kind of preorder
void addLeaves(Node * root, vector<int>& ans){
  if(root==NULL){
    return;
  }
  if(isLeaf(root)){
    ans.push_back(root→data); //just store leaf nodes
    return;
  }
  addLeaves(root→left,ans);
  addLeaves(root→right,ans);
}

vector <int> boundary(Node *root)
{
  //Your code here
  vector<int> ans;
  if(root==NULL) return ans;
  if(!isLeaf(root)) ans.push_back(root→data);
  addLeftBound(root,ans);
  addLeaves(root,ans);
  addRightBound(root,ans);

  return ans;

}
```

Explanation:

- The code defines a binary tree node structure with data, left, and right pointers.
- It implements a function boundary that returns the boundary elements of a binary tree in anti-clockwise order.
- The boundary elements include the root, left boundary (excluding leaves), leaves (left to right), and right boundary (excluding leaves).
- Three helper functions (addLeftBound, addRightBound, addLeaves) are used to traverse and add elements to the result vector accordingly.
- The result vector is then returned as the final output of the boundary function.

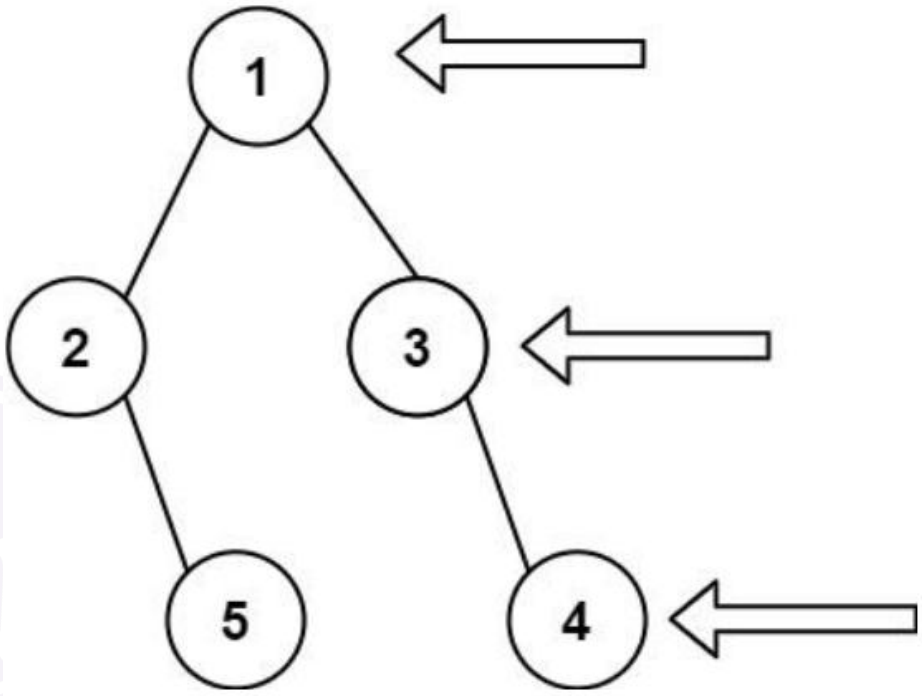
Time Complexity:
The time complexity is O(n), where n is the number of nodes in the binary tree.
This is because each node is visited once during the boundary traversal, and the traversal involves a constant amount of work for each node.

Space Complexity:
The space complexity is O(h), where h is the height of the binary tree.
This is due to the space used by the recursive call stack during the depth-first traversal of the tree.In the worst case, where the tree is skewed (essentially a linked list), the height is equal to the number of nodes, resulting in O(n) space complexity.In a balanced tree, the height is log(n), leading to a space complexity of O(log(n)).

Binary Tree Right Side View [Leetcode-199]

Given the root of a binary tree, imagine yourself standing on the right side of it, return the values of the nodes you can see ordered from top to bottom.

Example 1:



Input: root = [1,2,3,null,5,null,4]
Output: [1,3,4]

Example 2:

Input: root = [1,null,3]
Output: [1,3]

Example 3:

Input: root = []
Output: []

Code:

```
class Solution {
public:
    vector<int> rightSideView(TreeNode* root) {
        vector<int>ans;
        queue<TreeNode*> q;
        if(root==NULL)
        return ans;
        q.push(root);
        while(1)
        {
            int size=q.size();
            if(size==0)
            return ans;
            vector<int> data;
            while(size--)
            {
                TreeNode* temp=q.front();
                q.pop();
                data.push_back(temp->val);
                if(temp->left!=NULL)
                q.push(temp->left);
                if(temp->right!=NULL)
                q.push(temp->right);
            }
            ans.push_back(data.back());
        }
    }
};
```

Explanation:

- This C++ solution defines a class Solution with a method rightSideView to obtain the right view of a binary tree.
- It uses a breadth-first search (BFS) approach with a queue to traverse the tree level by level.
- It initializes an empty vector ans to store the result and a queue q for BFS. If the input tree is empty, it returns an empty vector.
- The algorithm iteratively processes each level of the tree, storing the rightmost element of each level in the result vector ans.
- The process continues until all levels are processed, and the final result vector is returned.

Time Complexity: O(N), where N is the number of nodes in the binary tree. Each node is visited once during the BFS traversal.

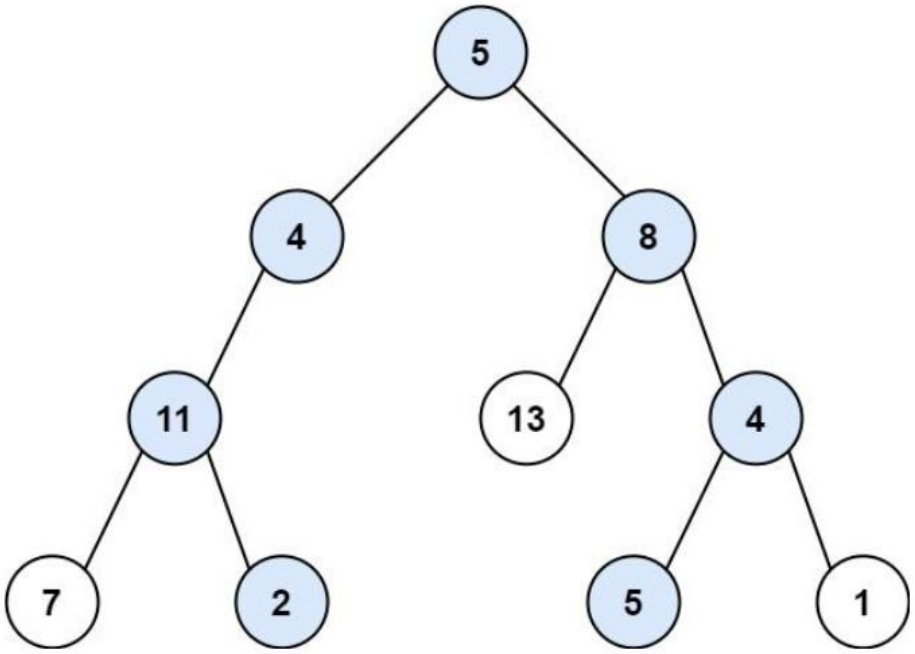
Space Complexity: O(W), where W is the maximum width (number of nodes at the same level) of the binary tree. In the worst case, the queue can store all nodes at a particular level. In a balanced binary tree, the maximum width is typically N/2, and in the worst case (skewed tree), it can be N.

Path Sum II [Leetcode-113]

Given the root of a binary tree and an integer targetSum, return all root-to-leaf paths where the sum of the node values in the path equals targetSum. Each path should be returned as a list of the node values, not node references.

A root-to-leaf path is a path starting from the root and ending at any leaf node. A leaf is a node with no children.

Example 1:



Input: root = [5,4,8,11,null,13,4,7,2,null,null,5,1], targetSum = 22
Output: [[5,4,11,2],[5,8,4,5]]

Explanation: There are two paths whose sum equals targetSum:
5 + 4 + 11 + 2 = 22
5 + 8 + 4 + 5 = 22

Code:

```
class Solution {
public:
    vector<vector<int>> path;
    void hasPathSum(TreeNode* root, int targetSum, vector<int>
vec) {
        if(!root) return;
        if(!root→right && !root→left)
        {
```

```
if(targetSum == root->val)
            {
                vec.push_back(root->val);
                path.push_back(vec);
            }
            return;
        }
        vector<int> ans = vec;

 vec.push_back(root->val);
        ans.push_back(root->val);
        hasPathSum(root->right, targetSum-root->val, vec) ;
        hasPathSum(root->left, targetSum-root->val, ans);
    }
    vector<vector<int>> pathSum(TreeNode* root, int targetSum) {
        hasPathSum(root, targetSum, {});
        return path;
    }
};
```

Explanation:

- This C++ code defines a Solution class with a public method pathSum to find paths in a binary tree that sum up to a given target.
- The class has a private member variable path to store the result as a vector of vectors of integers representing the paths.
- The hasPathSum method recursively explores the binary tree, checking for paths with the specified targetSum.
- If a leaf node is reached with a value equal to the remaining target sum, the path is added to the result.
- The pathSum method initializes the recursive process with an empty vector and returns the final result.

Time Complexity: O(N), where N is the number of nodes in the binary tree.

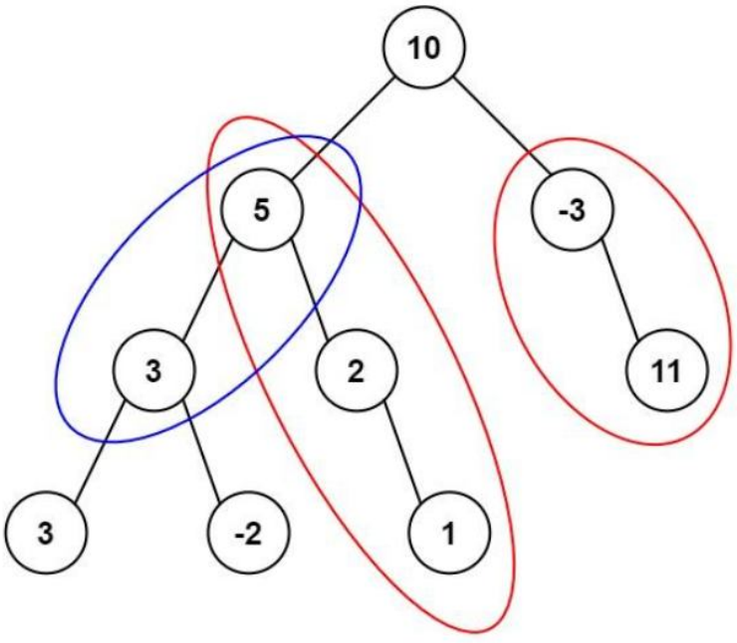
Space Complexity: O(H), where H is the height of the binary tree; in the worst case, it can be O(N) for an unbalanced tree.

Path Sum III [Leetcode-437]

Given the root of a binary tree and an integer targetSum, return the number of paths where the sum of the values along the path equals targetSum.

The path does not need to start or end at the root or a leaf, but it must go downwards (i.e., traveling only from parent nodes to child nodes).

Example 1:



Input: root = [10,5,-3,3,2,null,11,3,-2,null,1], targetSum = 8
Output: 3

Explanation: The paths that sum to 8 are shown.

Code:

```
class Solution {
public:
    int get(TreeNode *root, int t, map<long long, int> &mp, long
long pr = 0) {
        if (root == nullptr) return 0;
        pr += root->val;
        int ans = mp[pr - t];
        mp[pr]++;

        // Recursively calculate the count of paths for left and
right subtrees
        ans += get(root->left, t, mp, pr);
        ans += get(root->right, t, mp, pr);


        // Decrement the frequency of the current prefix sum
(backtrack)
        mp[pr]--;
        return ans;
    }

    int pathSum(TreeNode* root, int targetSum) {
        map<long long, int> mp;
        mp[0] = 1;
        return get(root, targetSum, mp);
    }
};
```

Explanation:

- The code defines a Solution class for solving a specific problem related to binary trees.
- The pathSum function is the main entry point for the solution, taking a binary tree root and a target sum as parameters.
- It initializes a map mp to keep track of prefix sums and their frequencies, with an initial entry for 0.
- Calls the helper function get with the root, target sum, and the prefix sum map.
- The get function recursively calculates the count of paths that sum to the target in the binary tree.
- It uses a prefix sum (pr) to keep track of the cumulative sum along the path.
- The frequency of each prefix sum is stored in the map mp.
- The function backtracks by decrementing the frequency of the current prefix sum after processing the left and right subtrees.
- The result is the total count of paths that sum to the target in the binary tree.**Time Complexity: O(N)**, where N is the number of nodes in the binary tree.

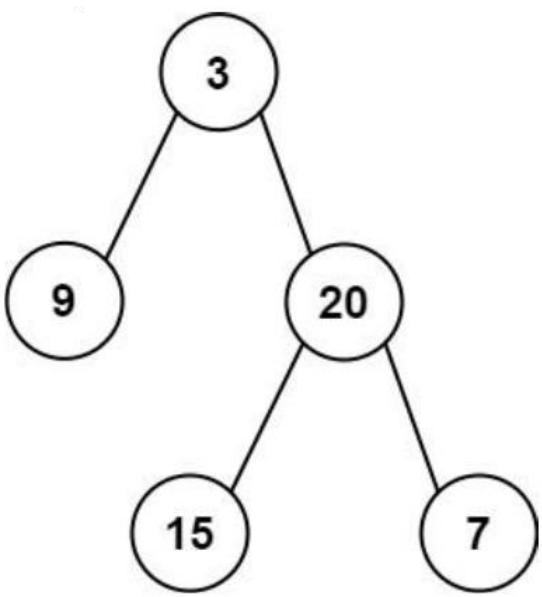
Time Complexity: O(N), where N is the number of nodes in the binary tree. The algorithm visits each node once during the recursive traversal.

Space Complexity: O(N), where N is the number of nodes in the binary tree. The space is primarily used for the prefix sum map (mp) to store the frequencies of prefix sums. In the worst case, the map could contain O(N) entries. Additionally, the recursion stack contributes to the space complexity, but since it's a binary tree, the maximum height of the recursion stack is O(log N) for a balanced tree, making the overall space complexity O(N + log N), which simplifies to O(N).

Construct Binary Tree from Preorder and Inorder Traversal [Leetcode-105]

Given two integer arrays preorder and inorder where preorder is the preorder traversal of a binary tree and inorder is the inorder traversal of the same tree, construct and return the binary tree.

Example 1:



Input: preorder = [3,9,20,15,7], inorder = [9,3,15,20,7]
Output: [3,9,20,null,null,15,7]

Code:

```
class Solution {
public:
TreeNode* constructTree(vector < int > & preorder, int preStart,
int preEnd, vector
 < int > & inorder, int inStart, int inEnd, map < int, int > &
mp) {
  if (preStart > preEnd || inStart > inEnd) return NULL;

  TreeNode* root = new TreeNode(preorder[preStart]);
  int elem = mp[root -> val];
  int nElem = elem - inStart;

 root -> left = constructTree(preorder, preStart + 1, preStart +
nElem, inorder,
  inStart, elem - 1, mp);
  root -> right = constructTree(preorder, preStart + nElem + 1,
preEnd, inorder,
  elem + 1, inEnd, mp);

  return root;
}

TreeNode* buildTree(vector < int > & preorder, vector < int > &
inorder) {
  int preStart = 0, preEnd = preorder.size() - 1;
  int inStart = 0, inEnd = inorder.size() - 1;

  map < int, int > mp;
  for (int i = inStart; i <= inEnd; i++) {
    mp[inorder[i]] = i;
  }
return constructTree(preorder, preStart, preEnd, inorder,
inStart, inEnd, mp);
}
};
```

Explanation:

- The code defines a C++ class named Solution.
- The class has a method constructTree that recursively constructs a binary tree from given preorder and inorder traversal arrays, along with the start and end indices for both arrays.
- The buildTree method initializes the start and end indices for both preorder and inorder arrays, creates a map to store the indices of inorder elements, and calls the constructTree method to build and return the binary tree.
- The constructed tree is built by selecting the root from the preorder array, finding its position in the inorder array using a map, and recursively constructing the left and right subtrees.
- The base case checks for empty subarrays (preStart > preEnd or inStart > inEnd) and returns NULL.
- Overall, the code implements the construction of a binary tree from its preorder and inorder traversals using recursion and a map to optimize finding the position of elements in the inorder array.

Time Complexity: O(n), where n is the total number of nodes in the binary tree.
The recursive function constructTree is called for each node exactly once, and each call performs constant time operations.
Since each node is processed once, the overall time complexity is linear.

Space Complexity: O(n), where n is the total number of nodes in the binary tree.
The space complexity is dominated by the space used for the recursive call stack.
In the worst case, the recursion depth is equal to the height of the binary tree, which can be O(n) in the case of a skewed tree.

Additionally, the space complexity includes the map (mp) used to store the indices of elements in the inorder array, which also requires O(n) space.
Therefore, the overall space complexity is O(n).