Day 6

Binary Tree Inorder Traversal

```
#include <iostream>
#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(nullptr), right(nullptr) {}
};

// Iterative function to perform inorder traversal
void inorderTraversal(TreeNode* root) {
    stack<TreeNode*> s;
    TreeNode* curr = root;

    while (curr != nullptr || !s.empty()) {
        // Reach the leftmost node of the current node
        while (curr != nullptr) {
            s.push(curr);
            curr = curr->left;
        }

        // Current node is null, process the node at the top of the stack
        curr = s.top();
        s.pop();

        // Visit the node
        cout << curr->val << " ";

        // Move to the right subtree
        curr = curr->right;
    }
}

// Helper function to insert nodes into the binary tree (for testing)
TreeNode* insert(TreeNode* root, int val) {
    if (root == nullptr) {
        return new TreeNode(val);
    }

    if (val < root->val) {
        root->left = insert(root->left, val);
    } else {
        root->right = insert(root->right, val);
    }

    return root;
}
```

```
int main() {
    TreeNode* root = nullptr;

    // Insert nodes into the binary search tree
    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 70);
    root = insert(root, 60);
    root = insert(root, 80);

    // Perform inorder traversal and print the result
    cout << "Inorder Traversal (Iterative): ";
    inorderTraversal(root);
    cout << endl;

    return 0;
}
```

Output

```
Inorder Traversal (Iterative): 20 30 40 50 60 70 80


=== Code Execution Successful ===
```

Count Complete Tree Nodes

```
#include <iostream>
using namespace std;

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

// Function to calculate the height of the tree
int getHeight(TreeNode* root) {
    int height = 0;
    while (root) {
        height++;
        root = root->left;  // Move to the left child
    }
```

```
    return height;
}

// Function to count the nodes in a complete binary tree
int countNodes(TreeNode* root) {
    if (!root) return 0;

    int height = getHeight(root);

    if (height == 1) {
        return 1;  // Only one node in the tree
    }

    // Binary search for the last level
    int left = (1 << (height - 2)), right = (1 << (height - 1)) - 1;
    int result = 0;

    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (nodeExists(root, height - 1, mid)) {
            // If node at position `mid` exists, all nodes up to `mid` exist in this level
            result = mid + 1;  // Including the current node
            left = mid + 1;
        } else {
            // Otherwise, exclude the right half of the search space
            right = mid - 1;
        }
    }

    // The total number of nodes is the number of nodes in the complete part of
the tree (height - 1)
    // plus the number of nodes found in the last level (result).
    return (1 << (height - 1)) - 1 + result;
}

// Function to check if a node exists at a particular index at a given height
bool nodeExists(TreeNode* root, int height, int index) {
    int left = 0, right = (1 << height) - 1;

    for (int i = 0; i < height; ++i) {
        int mid = left + (right - left) / 2;

        if (index <= mid) {
            root = root->left;
            right = mid;
        } else {
```

```
            root = root->right;
            left = mid + 1;
        }
    }

    return root != nullptr;
}

int main() {

    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    root->left->left = new TreeNode(4);
    root->left->right = new TreeNode(5);
    root->right->left = new TreeNode(6);

    cout << "The number of nodes in the tree is: " << countNodes(root) <<
endl;

    return 0;
}
```

Binary Tree Preorder Traversal

```
#include <iostream>
using namespace std;

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

// Recursive function to perform preorder traversal
void preorderTraversal(TreeNode* root) {
    if (root == nullptr) {
        return;
    }

    // Visit the root node
    cout << root->val << " ";
```

```
    // Traverse the left subtree
    preorderTraversal(root->left);

    // Traverse the right subtree
    preorderTraversal(root->right);
}

// Helper function to insert nodes into the binary tree (for testing)
TreeNode* insert(TreeNode* root, int val) {
    if (root == nullptr) {
        return new TreeNode(val);
    }

    if (val < root->val) {
        root->left = insert(root->left, val);
    } else {
        root->right = insert(root->right, val);
    }

    return root;
}

int main() {
    TreeNode* root = nullptr;

    // Insert nodes into the binary search tree
    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 70);
    root = insert(root, 60);
    root = insert(root, 80);

    // Perform preorder traversal and print the result
    cout << "Preorder Traversal: ";
    preorderTraversal(root);
    cout << endl;

    return 0;
}
```

Output

```
Preorder Traversal: 50 30 20 40 70 60 80


=== Code Execution Successful ===
```

Leaf Nodes of a Binary Tree

```
#include <iostream>
using namespace std;

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

// Recursive function to print leaf nodes
void printLeafNodes(TreeNode* root) {
    // Base case: if the node is null, just return
    if (root == nullptr) {
        return;
    }

    // If the node is a leaf node (no left and no right child), print its value
    if (root->left == nullptr && root->right == nullptr) {
        cout << root->val << " ";
    }

    // Recursively call on the left and right subtree
    printLeafNodes(root->left);
    printLeafNodes(root->right);
}

// Helper function to insert nodes into the binary tree (for testing)
TreeNode* insert(TreeNode* root, int val) {
    if (root == nullptr) {
        return new TreeNode(val);
    }
```

```
    if (val < root->val) {
        root->left = insert(root->left, val);
    } else {
        root->right = insert(root->right, val);
    }

    return root;
}

int main() {
    TreeNode* root = nullptr;

    // Insert nodes into the binary search tree
    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 70);
    root = insert(root, 60);
    root = insert(root, 80);

    // Print all leaf nodes
    cout << "Leaf nodes of the tree: ";
    printLeafNodes(root);
    cout << endl;

    return 0;
}
```

Output

```
Leaf nodes of the tree: 20 40 60 80


=== Code Execution Successful ===
```

Construct Binary Tree from Preorder and Inorder Traversal

```
#include <iostream>
#include <unordered_map>
#include <vector>
using namespace std;

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

```
};

// Helper function to build the tree
TreeNode* buildTreeHelper(vector<int>& preorder, int preStart, int preEnd,
                          vector<int>& inorder, int inStart, int inEnd,
                          unordered_map<int, int>& inOrderIndexMap) {
    if (preStart > preEnd || inStart > inEnd) {
        return nullptr;
    }

    // The first element in preorder is the root node
    int rootValue = preorder[preStart];
    TreeNode* root = new TreeNode(rootValue);

    // Get the index of the root in the inorder traversal
    int rootIndexInInorder = inOrderIndexMap[rootValue];

    // Number of nodes in the left subtree
    int leftSubtreeSize = rootIndexInInorder - inStart;

    // Recursively build the left and right subtrees
    root->left = buildTreeHelper(preorder, preStart + 1, preStart +
leftSubtreeSize,
                                 inorder, inStart, rootIndexInInorder - 1,
inOrderIndexMap);
    root->right = buildTreeHelper(preorder, preStart + leftSubtreeSize + 1,
preEnd,
                                  inorder, rootIndexInInorder + 1, inEnd,
inOrderIndexMap);

    return root;
}

// Function to build the tree from preorder and inorder traversals
TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
    // Create a map to store the index of each value in inorder traversal for O(1)
lookup
    unordered_map<int, int> inOrderIndexMap;
    for (int i = 0; i < inorder.size(); ++i) {
        inOrderIndexMap[inorder[i]] = i;
    }

    // Call the helper function to build the tree
    return buildTreeHelper(preorder, 0, preorder.size() - 1,
                           inorder, 0, inorder.size() - 1,
                           inOrderIndexMap);
```

```
}

// Helper function to print the tree (Inorder Traversal)
void printInorder(TreeNode* root) {
    if (root != nullptr) {
        printInorder(root->left);
        cout << root->val << " ";
        printInorder(root->right);
    }
}

int main() {
    // Example Preorder and Inorder Traversals
    vector<int> preorder = { 1, 2, 4, 5, 3, 6, 7 };
    vector<int> inorder = { 4, 2, 5, 1, 6, 3, 7 };

    // Build the tree
    TreeNode* root = buildTree(preorder, inorder);

    // Print the inorder traversal of the constructed tree
    cout << "Inorder traversal of the constructed tree: ";
    printInorder(root);
    cout << endl;

    return 0;
}
```

Output

```
Inorder traversal of the constructed tree: 4 2 5 1 6 3 7


=== Code Execution Successful ===
```

Lowest Common Ancestor of a Binary Tree

```
#include <iostream>
using namespace std;

// Definition for a binary tree node.
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
```

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

// Recursive function to find the LCA of two nodes
TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p,
TreeNode* q) {
    // Base case: if root is null, or root is one of the nodes
    if (root == nullptr || root == p || root == q) {
        return root;
    }

    // Recur for left and right subtrees
    TreeNode* left = lowestCommonAncestor(root->left, p, q);
    TreeNode* right = lowestCommonAncestor(root->right, p, q);

    // If both left and right subtrees returned non-null values,
    // then the current node is the LCA
    if (left != nullptr && right != nullptr) {
        return root;
    }

    // Otherwise, return the non-null node
    return left != nullptr ? left : right;
}

// Helper function to insert nodes into the binary tree
TreeNode* insert(TreeNode* root, int val) {
    if (root == nullptr) {
        return new TreeNode(val);
    }

    if (val < root->val) {
        root->left = insert(root->left, val);
    } else {
        root->right = insert(root->right, val);
    }

    return root;
}

// Helper function to print inorder traversal of the tree
void printInorder(TreeNode* root) {
    if (root != nullptr) {
        printInorder(root->left);
        cout << root->val << " ";
        printInorder(root->right);
```

```
    }
}

int main() {
    TreeNode* root = nullptr;

    // Insert nodes into the binary search tree
    root = insert(root, 20);
    root = insert(root, 10);
    root = insert(root, 30);
    root = insert(root, 5);
    root = insert(root, 15);
    root = insert(root, 25);
    root = insert(root, 35);

    // Find the LCA of two nodes
    TreeNode* p = root->left->left; // Node with value 5
    TreeNode* q = root->left->right; // Node with value 15

    TreeNode* lca = lowestCommonAncestor(root, p, q);

    if (lca != nullptr) {
        cout << "The LCA of " << p->val << " and " << q->val << " is: " << lca-
>val << endl;
    } else {
        cout << "No LCA found." << endl;
    }

    return 0;
}
```

Output

```
The LCA of 5 and 15 is: 10


=== Code Execution Successful ===
```

Lowest Common Ancestor of a Binary Tree

```
#include <iostream>
using namespace std;
```

```
    }

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

// Recursive function to find the LCA of two nodes
TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p,
TreeNode* q) {
    // Base case: if root is null, or root is one of the nodes
    if (root == nullptr || root == p || root == q) {
        return root;
    }

    // Recur for left and right subtrees
    TreeNode* left = lowestCommonAncestor(root->left, p, q);
    TreeNode* right = lowestCommonAncestor(root->right, p, q);

    // If both left and right subtrees returned non-null values,
    // then the current node is the LCA
    if (left != nullptr && right != nullptr) {
        return root;
    }

    // Otherwise, return the non-null node
    return left != nullptr ? left : right;
}

// Helper function to insert nodes into the binary tree
TreeNode* insert(TreeNode* root, int val) {
    if (root == nullptr) {
        return new TreeNode(val);
    }

    if (val < root->val) {
        root->left = insert(root->left, val);
    } else {
        root->right = insert(root->right, val);
    }

    return root;
}

// Helper function to print inorder traversal of the tree
```

```
void printInorder(TreeNode* root) {
    if (root != nullptr) {
        printInorder(root->left);
        cout << root->val << " ";
        printInorder(root->right);
    }
}

int main() {
    TreeNode* root = nullptr;

    // Insert nodes into the binary search tree
    root = insert(root, 20);
    root = insert(root, 10);
    root = insert(root, 30);
    root = insert(root, 5);
    root = insert(root, 15);
    root = insert(root, 25);
    root = insert(root, 35);

    // Find the LCA of two nodes
    TreeNode* p = root->left->left; // Node with value 5
    TreeNode* q = root->left->right; // Node with value 15

    TreeNode* lca = lowestCommonAncestor(root, p, q);

    if (lca != nullptr) {
        cout << "The LCA of " << p->val << " and " << q->val << " is: " << lca->val << endl;
    } else {
        cout << "No LCA found." << endl;
    }

    return 0;
}
```

Output

```
The LCA of 5 and 15 is: 10


=== Code Execution Successful ===
```

Sum Root to Leaf Numbers

```
#include <iostream>
using namespace std;

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

// Helper function to perform DFS and sum root-to-leaf numbers
void dfs(TreeNode* root, int currentSum, int &totalSum) {
    if (root == nullptr) {
        return;
    }

    // Update the current number by adding the current node's value
    currentSum = currentSum * 10 + root->val;

    // If it's a leaf node, add the current number to totalSum
    if (root->left == nullptr && root->right == nullptr) {
        totalSum += currentSum;
        return;
    }

    // Recur for left and right subtrees
    dfs(root->left, currentSum, totalSum);
    dfs(root->right, currentSum, totalSum);
}

// Function to calculate the sum of all root-to-leaf numbers
int sumNumbers(TreeNode* root) {
    int totalSum = 0;
    dfs(root, 0, totalSum);
    return totalSum;
}

// Helper function to insert nodes into the binary tree (for testing)
TreeNode* insert(TreeNode* root, int val) {
    if (root == nullptr) {
        return new TreeNode(val);
    }
```

```
    if (val < root->val) {
        root->left = insert(root->left, val);
    } else {
        root->right = insert(root->right, val);
    }

    return root;
}

// Example usage
int main() {
    TreeNode* root = nullptr;

    // Insert nodes into the binary tree
    root = insert(root, 1);
    root = insert(root, 2);
    root = insert(root, 3);

    // Find the sum of all root-to-leaf numbers
    int result = sumNumbers(root);
    cout << "Sum of all root-to-leaf numbers: " << result << endl;

    return 0;
}
```

Output

```
Sum of all root-to-leaf numbers: 123


=== Code Execution Successful ===
```

Binary Tree Right Side View

```
#include <iostream>
#include <queue>
#include <vector>
using namespace std;
```

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

// Function to get the right side view of the binary tree
vector<int> rightSideView(TreeNode* root) {
    vector<int> result;
    if (root == nullptr) {
        return result;
    }

    queue<TreeNode*> q;
    q.push(root);

    while (!q.empty()) {
        int levelSize = q.size(); // Number of nodes at the current level

        // Traverse all nodes of the current level
        for (int i = 0; i < levelSize; i++) {
            TreeNode* node = q.front();
            q.pop();

            // If it's the last node of this level, add it to the result
            if (i == levelSize - 1) {
                result.push_back(node->val);
            }

            // Add left and right children to the queue
            if (node->left != nullptr) {
                q.push(node->left);
            }
            if (node->right != nullptr) {
                q.push(node->right);
            }
        }
    }

    return result;
}

// Helper function to insert nodes into the binary tree (for testing)
TreeNode* insert(TreeNode* root, int val) {
```

```
    if (root == nullptr) {
        return new TreeNode(val);
    }

    if (val < root->val) {
        root->left = insert(root->left, val);
    } else {
        root->right = insert(root->right, val);
    }

    return root;
}

// Example usage
int main() {
    TreeNode* root = nullptr;

    // Insert nodes into the binary tree
    root = insert(root, 1);
    root = insert(root, 2);
    root = insert(root, 3);
    root = insert(root, 4);
    root = insert(root, 5);
    root = insert(root, 6);
    root = insert(root, 7);

    // Get the right side view of the binary tree
    vector<int> result = rightSideView(root);

    // Print the right side view
    cout << "Right side view of the tree: ";
    for (int val : result) {
        cout << val << " ";
    }
    cout << endl;

    return 0;
}
```

Output

```
Right side view of the tree: 1 2 3 4 5 6 7


=== Code Execution Successful ===
```