Name – Vanshaj Rana

UID – 22BCS15417

Section – 22BCS_IOT_620(B)

Date – 27 Dec 2024

1. Binary Order Traversal

Code :

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

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

// Recursive function for inorder traversal
void inorderTraversal(TreeNode* root) {
   if (root == NULL) return;

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

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

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

// Main function
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);

    cout << "Inorder Traversal: ";
    inorderTraversal(root); // Output: 4 2 5 1 3
    cout << endl;

    return 0;
}
```

Output

```
Output

Inorder Traversal: 4 2 5 1 3


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

2. Count Complete Tree Node

Code:

```
#include <iostream>
#include <cmath> // For pow()
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) {}
};

// Function to calculate the height of a subtree
int getHeight(TreeNode* node) {
    int height = 0;
    while (node) {
    height++;
    node = node->left; // Traverse only the left subtree
```

```
    }
    return height;
}

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

    int leftHeight = getHeight(root->left);
    int rightHeight = getHeight(root->right);

    if (leftHeight == rightHeight) {
        // Left subtree is a perfect binary tree
        return (1 << leftHeight) + countNodes(root->right);
    } else {
        // Right subtree is a perfect binary tree
        return (1 << rightHeight) + countNodes(root->left);
    }
}

// Main function to test the code
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 << "Number of nodes: " << countNodes(root) << endl; // Output: 6

    return 0;
}
```

OUTPUT

```
Output

Number of nodes: 6

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

3. Binary Tree – Find Maximum Depth

Code:

```
#include <iostream>
#include <algorithm> // For max()
using namespace std;

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

int maxDepth(TreeNode* root) {
   if (!root) return 0; // Base case: empty tree has depth 0

   int leftDepth = maxDepth(root->left); // Depth of the left subtree
   int rightDepth = maxDepth(root->right); // Depth of the right subtree

   return 1 + max(leftDepth, rightDepth); // Add 1 for the current node
}

int main() {

   TreeNode* root = new TreeNode(3);
   root->left = new TreeNode(9);
   root->right = new TreeNode(20);
   root->right->left = new TreeNode(15);
   root->right->right = new TreeNode(7);

   cout << "Maximum Depth: " << maxDepth(root) << endl; // Output: 3

   return 0;
}
```

OUTPUT

Output

```
Maximum Depth: 3


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

4. Binary Order Pre Traversal

Code:

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

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

// Iterative function for preorder traversal
vector<int> preorderTraversal(TreeNode* root) {
   vector<int> result;
   if (root == NULL) return result;

   stack<TreeNode*> stk;
   stk.push(root);

   while (!stk.empty()) {
      TreeNode* node = stk.top();
      stk.pop();
      result.push_back(node->val); // Visit the root node

      if (node->right) stk.push(node->right);
      if (node->left) stk.push(node->left);
   }

   return result;
}

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);

   vector<int> result = preorderTraversal(root);
```

```
    cout << "Preorder Traversal: ";
    for (int val : result) { cout <<
    val << " ";
    }
    cout << endl;

    return 0;
}
```

OUTPUT

Output

```
Preorder Traversal: 1 2 4 5 3


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

5. Binary Tree – Sum of all Nodes

Code:

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

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

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

    int sum = 0;
    queue<TreeNode*> q;
    q.push(root);
```

```
 while (!q.empty()) {
      TreeNode* current = q.front();
      q.pop();

      sum += current->val;
      if (current->left) q.push(current->left);
      if (current->right) q.push(current->right);
   }

   return sum;
}

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->right = new TreeNode(6);

   cout << "Sum of all nodes: " << sumOfNodes(root) << endl; // Output: 21

   return 0;
}
```

OUTPUT

Output

```
Sum of all nodes: 21


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

6. Same Tree

Code:

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

// Definition of a binary tree node
struct TreeNode {
   int val;
```

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

// Iterative function to check if two trees are the same
bool isSameTree(TreeNode* p, TreeNode* q) {
    queue<TreeNode*> qp, qq;
    qp.push(p);
    qq.push(q);

    while (!qp.empty() && !qq.empty()) {
        TreeNode* nodeP = qp.front(); qp.pop();
        TreeNode* nodeQ = qq.front(); qq.pop();

        if (!nodeP && !nodeQ) continue; // Both nodes are NULL, continue
        if (!nodeP || !nodeQ || nodeP->val != nodeQ->val) return false; // Structural or value
mismatch

        // Enqueue left and right children
        qp.push(nodeP->left);
        qp.push(nodeP->right);
        qq.push(nodeQ->left);
        qq.push(nodeQ->right);
    }

    return qp.empty() && qq.empty();
}

int main() {

    TreeNode* p = new TreeNode(1);
    p->left = new TreeNode(2);
    p->right = new TreeNode(3);

    TreeNode* q = new TreeNode(1);
    q->left = new TreeNode(2);
    q->right = new TreeNode(3);

    cout << (isSameTree(p, q) ? "true" : "false") << endl;

    return 0;
}
```

OUTPUT

Output

```
true


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

7. Invert Binary Tree

Code:

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

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

// Iterative function to invert a binary tree
TreeNode* invertTree(TreeNode* root) {
   if (root == NULL) return NULL;

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

   while (!q.empty()) {
      TreeNode* node = q.front();
      q.pop();

      // Swap the left and right children
      TreeNode* temp = node->left;
      node->left = node->right;
      node->right = temp;

      // Add children to the queue
      if (node->left) q.push(node->left);
      if (node->right) q.push(node->right);
   }
```

```cpp
    return root;
}

// Helper function to print the tree in level order
void printLevelOrder(TreeNode* root) {
    if (root == NULL) return;

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

    while (!q.empty()) {
        TreeNode* node = q.front();
        q.pop();

        if (node) {
            cout << node->val << " ";
            q.push(node->left);
            q.push(node->right);
        } else {
            cout << "null ";
        }
    }
}

// Main function to test the code
int main() {

    TreeNode* root = new TreeNode(4);
    root->left = new TreeNode(2);
    root->right = new TreeNode(7);
    root->left->left = new TreeNode(1);
    root->left->right = new TreeNode(3);
    root->right->left = new TreeNode(6);
    root->right->right = new TreeNode(9);

    cout << "Original tree (level order): ";
    printLevelOrder(root);
    cout << endl;

    // Invert the tree
    root = invertTree(root);

    cout << "Inverted tree (level order): ";
```

```
    printLevelOrder(root);
    cout << endl;

    return 0;
}
```

OUTPUT

```
Output                                                        Clear

Original tree (level order): 4 2 7 1 3 6 9 null null null null null null
    null null
Inverted tree (level order): 4 7 2 9 6 3 1 null null null null null null
    null null


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

8. Path Sum

Code:

```
#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(NULL), right(NULL) {}
};

// Function to check if a path with the given sum exists
bool hasPathSum(TreeNode* root, int sum) {
    if (root == NULL) return false;

    // Check if we have reached a leaf node and the sum matches
    if (root->left == NULL && root->right == NULL) {
        return sum == root->val;
    }

    // Recur on left and right subtrees with the remaining sum
    int remainingSum = sum - root->val;
    return hasPathSum(root->left, remainingSum) || hasPathSum(root->right, remainingSum);
}
```

```
// Main function to test the code
int main() {

   TreeNode* root = new TreeNode(5);
   root->left = new TreeNode(4);
   root->right = new TreeNode(8);
   root->left->left = new TreeNode(11);
   root->right->left = new TreeNode(13);
   root->right->right = new TreeNode(4);
   root->left->left->left = new TreeNode(7);
   root->left->left->right = new TreeNode(2);
   root->right->right->right = new TreeNode(1);

   int targetSum = 22;

   if (hasPathSum(root, targetSum)) {
      cout << "Path with sum " << targetSum << " exists." << endl;
   } else {
      cout << "No path with sum " << targetSum << " exists." << endl;
   }

   return 0;
}
```

OUTPUT

```
Output

Path with sum 22 exists.


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

9. Construct Binary Tree from preorder and inorder traversal

Code:

```
#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(NULL), right(NULL) {}
};

// 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>& inorderMap) {
    if (preStart > preEnd || inStart > inEnd) return NULL;

    // Root value from preorder
    int rootVal = preorder[preStart];
    TreeNode* root = new TreeNode(rootVal);

    // Find the root in the inorder array
    int inRootIndex = inorderMap[rootVal];
    int leftTreeSize = inRootIndex - inStart;

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

    return root;
}

// Main function to build the tree
TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
    unordered_map<int, int> inorderMap;
    for (int i = 0; i < inorder.size(); i++) {
        inorderMap[inorder[i]] = i; // Store value to index mapping for inorder
    }
    return buildTreeHelper(preorder, 0, preorder.size() - 1,
    inorder, 0, inorder.size() - 1, inorderMap);
}

// Helper function to print the tree (inorder traversal)
void printInorder(TreeNode* root) {
    if (root == NULL) return;
    printInorder(root->left);
```

```
    cout << root->val << " ";
    printInorder(root->right);
}

int main() {
    // Example input
    vector<int> preorder = {3, 9, 20, 15, 7};
    vector<int> inorder = {9, 3, 15, 20, 7};

    TreeNode* root = buildTree(preorder, inorder);

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

    return 0;
}
```

OUTPUT

```
Output

Inorder of the constructed tree: 9 3 15 20 7


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

10. Lowest Common Ancesstor Binary Tree

Code:

```
#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(NULL), right(NULL) {}
};

class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
```

```
        // Base case
        if (root == NULL || root == p || root == q) {
            return root;
        }

        // Recursively find LCA in left and right subtrees
        TreeNode* left = lowestCommonAncestor(root->left, p, q);
        TreeNode* right = lowestCommonAncestor(root->right, p, q);

        // If both left and right are non-NULL, root is the LCA
        if (left != NULL && right != NULL) {
            return root;
        }

        // Otherwise, return the non-NULL subtree (if any)
        return (left != NULL) ? left : right;
    }
};

// Helper function to create a simple binary tree for testing
TreeNode* createTree() {
    TreeNode* root = new TreeNode(3);
    root->left = new TreeNode(5);
    root->right = new TreeNode(1);
    root->left->left = new TreeNode(6);
    root->left->right = new TreeNode(2);
    root->right->left = new TreeNode(0);
    root->right->right = new TreeNode(8);
    root->left->right->left = new TreeNode(7);
    root->left->right->right = new TreeNode(4);
    return root;
}

int main() {
    Solution solution;
    TreeNode* root = createTree();

    // Example: Find LCA of nodes 5 and 1
    TreeNode* p = root->left; // Node 5
    TreeNode* q = root->right; // Node 1

    TreeNode* lca = solution.lowestCommonAncestor(root, p, q);
    if (lca != NULL) {
        cout << "The LCA of " << p->val << " and " << q->val << " is " << lca->val << endl;
```

```
    } else {
        cout << "No common ancestor found." << endl;
    }

    return 0;
}
```

OUTPUT

Output

```
The LCA of 5 and 1 is 3


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