Trees

Binary Trees

Binary Tree :

```
struct BinaryTreeNode {
    int data;
    struct BinaryTreeNode *left;
    struct BinaryTreeNode *right;
}
```

Preorder Traversal :

Recursive :

```
void preOrder(struct BinaryTreeNode *root) {
    if(root) {
        cout << root->data;
        preOrder(root->left);
        preOrder(root->right);
    }
}
```

Non Recursive :

```
void preOrder(struct BinaryTreeNode *root) {
    struct Stack *S = createStack();
    while(root || !isEmpty(S)) {
        while(root) {
            cout << root->data;
            push(S, root);
            root = root->left;
        }
        if(isEmpty(S)) {
            break;
        }
        root = pop(S);
        root = root->right;
    }
    deleteStack(S);
}
```

Inorder Traversal :

Recursive :

```
void inOrder(struct BinaryTreeNode *root) {
    if(root) {
        inOrder(root->left);
```

```
        cout << root->data;
        inOrder(root->right);
    }
}
```

Non Recursive :

```
void inOrder(struct BinaryTreeNode *root) {
    struct Stack *S = createStack();
    while(root || !isEmpty(S)) {
        while(root) {
            push(S, root);
            root = root->left;
        }
        if(isEmpty(S)) {
            break;
        }
        root = pop(S);
        cout << root->data;
        root = root->right;
    }
    deleteStack(S);
}
```

Post Order Traversal :

Recursive :

```
void postOrder(struct BinaryTreeNode *root) {
    if(root) {
        postOrder(root->left);
        postOrder(root->right);
        cout << root->data;
    }
}
```

Non Recursive :

```
void postOrder(struct BinaryTreeNode *root) {
    struct Stack *S = createStack();
    struct BinaryTreeNode *previous = NULL;
    do{
        while(root!= NULL) {
            push(S, root);
            root = root->left;
        }
        while(root == NULL && !isEmpty(S)) {
            root = top(S);
            if(root->right == NULL || root->right == previous) {
```

```
                cout << root->data;
                pop(S);
                previous = root;
                root = NULL;
            } else {
                root = root->right;
            }
        }
    } while(!isEmpty(S));
}
```

Level Order Traversal :

```
void levelOrder(struct BinaryTreeNode *root) {
    struct BinaryTreeNode *temp;
    struct Queue *Q = createQueue();
    if(!root) {
        return;
    }
    enQueue(Q, root);
    while(!isEmpty(Q)) {
        temp = deQueue(Q);
        cout << temp->data;
        if(temp->left) {
            enQueue(Q, temp->left);
        }
        if(temp->right) {
            enQueue(Q, temp->right);
        }
    }
    deleteQueue(Q);
}
```

Problems

Find Maximum element in a Binary Tree
Recursion :

```
int findMax(struct BinaryTreeNode *root) {
    int root_val, left, right, max = INT_MIN;
    if(root) {
        root_val = root->data;
        left = findMax(root->left);
        right = findMax(root->right);
        if(left > right) {
            max = left;
        } else {
```

```
            max = right;
        }
        if(root_val > max) {
            max = root_val;
        }
    }
    return max;
}
```

Non Recursive :

```
// Using levelOrder
int findMax(struct BinaryTreeNode *root) {
    struct BinaryTreeNode *temp;
    int max = INT_MIN;
    struct Queue *Q = createQueue();
    enQueue(Q, root);
    while(!isEmpty(Q)) {
        temp = deQueue(Q);
        if(max < temp->data) {
            max = temp_data;
        }
        if(temp->left) {
            enQueue(Q, temp->left);
        }
        if(temp->right) {
            enQueue(Q, temp->right);
        }
    }
    deleteQueue(Q);
    return max;
}
```

Searching an element in a Binary Tree (return 0 if not found, 1 if found) :
Recursive :

```
int find(struct BinaryTreeNode *root, int data) {
    int temp;
    if(root == NULL) {
        return 0;
    } else {
        if(data == root->data) {
            return 1;
        } else {
            temp = find(root->left, data);
            if(temp != 0) {
```

```
                return temp;
            } else {
                return find(root->right, data);
            }
        }
    }
    return 0;
}
```

Non Recursive :

```
// Using level order
int search(struct BinaryTreeNode *root, int data) {
    struct BinaryTreeNode *temp;
    struct Queue *Q = createQueue();
    enQueue(Q, root);
    while(isEmpty(Q)) {
        temp = deQueue(Q);
        if(data == temp->data) {
            deleteQueue(Q);
            return 1;
        }
        if(temp->left) {
            enQueue(Q, temp->left);
        }
        if(temp->right) {
            enQueue(Q, temp->right);
        }
    }
    deleteQueue(Q);
    return 0;
}
```

Inserting an element into Binary Tree :

```
void insertInBinaryTree(BinaryTreeNode*& root, int data) {
    BinaryTreeNode* newNode = new BinaryTreeNode();
    if (!newNode) {
        cout << "Memory Error" << endl;
        return;
    }
    newNode->data = data;
    newNode->left = nullptr;
    newNode->right = nullptr;

    if (!root) {
```

```
        root = newNode;
        return;
    }

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

    while (!q.empty()) {
        BinaryTreeNode* temp = q.front();
        q.pop();

        if (temp->left) {
            q.push(temp->left);
        } else {
            temp->left = newNode;
            return;
        }

        if (temp->right) {
            q.push(temp->right);
        } else {
            temp->right = newNode;
            return;
        }
    }
}
```

Find the size of the Binary Tree :

Recursion :

```
int size(struct BinaryTreeNode *root) {
    if(root == nullptr) {
        return 0;
    } else {
        return size(root->left) + 1 + size(root->right);
    }
}
```

Non Recursion :

```
// Using level order
struct BinaryTreeNode {
    int data;
    BinaryTreeNode* left;
    BinaryTreeNode* right;
};
```

```
int sizeOfBTUsingLevelOrder(BinaryTreeNode* root) {
    if (!root)
        return 0;

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

    int count = 0;

    while (!q.empty()) {
        BinaryTreeNode* temp = q.front();
        q.pop();
        count++;

        if (temp->left)
            q.push(temp->left);

        if (temp->right)
            q.push(temp->right);
    }

    return count;
}
```

Print Level Order data in reverse order :

```
void levelOrderTraversalInReverse(BinaryTreeNode* root) {
    if (!root)
        return;

    queue<BinaryTreeNode*> Q;
    stack<BinaryTreeNode*> S;

    Q.push(root);

    while (!Q.empty()) {
        BinaryTreeNode* temp = Q.front();
        Q.pop();
        S.push(temp);
        if (temp->right)
            Q.push(temp->right);

        if (temp->left)
            Q.push(temp->left);
    }
    while (!S.empty()) {
```

```
        cout << S.top()->data << " ";
        S.pop();
    }
}
```

Deleting a tree :

```
void deleteBT(struct BinaryTreeNode *root) {
    if (root == nullptr) {
        return;
    }
    deleteBT(root->left);
    deleteBT(root->right);
    delete root; // Use delete for memory allocated with new
}
```

Height (or depth) of Binary Tree :

Recursion :

```
int heightOfBinaryTree(BinaryTreeNode* root) {
    if (root == NULL)
        return 0;
    else {
        int leftHeight = heightOfBinaryTree(root->left);
        int rightHeight = heightOfBinaryTree(root->right);

        if (leftHeight > rightHeight)
            return leftHeight + 1;
        else
            return rightHeight + 1;
    }
}
```

Non Recursion :

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

struct BinaryTreeNode {
    int data;
    BinaryTreeNode* left;
    BinaryTreeNode* right;
};

int findHeightOfBinaryTree(BinaryTreeNode* root) {
    if (root == NULL)
        return 0;
```

```
    int level = 0;
    queue<BinaryTreeNode*> Q;
    Q.push(root);
    Q.push(NULL);

    while (!Q.empty()) {
        BinaryTreeNode* node = Q.front();
        Q.pop();

        if (node == NULL) {
            level++;
            if (!Q.empty())
                Q.push(NULL);
        } else {
            if (node->left)
                Q.push(node->left);
            if (node->right)
                Q.push(node->right);
        }
    }
    return level;
}
```

Finding deepest node of the binary tree :

```
BinaryTreeNode* deepestNodeInBinaryTree(BinaryTreeNode* root) {
    if (root == NULL)
        return NULL;

    queue<BinaryTreeNode*> Q;
    BinaryTreeNode* temp = NULL;

    Q.push(root);

    while (!Q.empty()) {
        temp = Q.front();
        Q.pop();

        if (temp->left)
            Q.push(temp->left);

        if (temp->right)
            Q.push(temp->right);
    }
```

```
    return temp;
}
```

****Deleting an element from the binary tree :**

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

struct BinaryTreeNode {
    int data;
    BinaryTreeNode* left;
    BinaryTreeNode* right;
};

// Function to find the deepest node in the binary tree
BinaryTreeNode* findDeepestNode(BinaryTreeNode* root) {
    if (root == NULL)
        return NULL;

    queue<BinaryTreeNode*> Q;
    BinaryTreeNode* temp = NULL;
    Q.push(root);

    while (!Q.empty()) {
        temp = Q.front();
        Q.pop();

        if (temp->left)
            Q.push(temp->left);

        if (temp->right)
            Q.push(temp->right);
    }

    return temp;
}

// Function to delete the deepest node in the binary tree
void deleteDeepestNode(BinaryTreeNode* root, BinaryTreeNode* delNode)
{
    queue<BinaryTreeNode*> Q;
    Q.push(root);

    while (!Q.empty()) {
        BinaryTreeNode* temp = Q.front();
        Q.pop();
```

```
        if (temp == delNode) {
            temp = NULL;
            delete delNode;
            return;
        }

        if (temp->left) {
            if (temp->left == delNode) {
                temp->left = NULL;
                delete delNode;
                return;
            } else {
                Q.push(temp->left);
            }
        }

        if (temp->right) {
            if (temp->right == delNode) {
                temp->right = NULL;
                delete delNode;
                return;
            } else {
                Q.push(temp->right);
            }
        }
    }
}

// Function to delete a node from the binary tree
BinaryTreeNode* deleteNode(BinaryTreeNode* root, int key) {
    if (root == NULL)
        return NULL;

    if (root->left == NULL && root->right == NULL) {
        if (root->data == key) {
            delete root;
            return NULL;
        } else {
            return root;
        }
    }

    queue<BinaryTreeNode*> Q;
    Q.push(root);
```

```
    BinaryTreeNode* keyNode = NULL;
    BinaryTreeNode* temp;

    // Find the node to delete and perform level-order traversal
    while (!Q.empty()) {
        temp = Q.front();
        Q.pop();

        if (temp->data == key)
            keyNode = temp;

        if (temp->left)
            Q.push(temp->left);

        if (temp->right)
            Q.push(temp->right);
    }

    if (keyNode) {
        // Find the deepest node
        BinaryTreeNode* deepest = findDeepestNode(root);
        // Replace the keyNode's data with the deepest node's data
        keyNode->data = deepest->data;
        // Delete the deepest node
        deleteDeepestNode(root, deepest);
    }

    return root;
}
```

Finding no of leaves w/o recursion :

```
int countLeaves(BinaryTreeNode* root) {
    if (root == NULL)
        return 0;

    queue<BinaryTreeNode*> Q;
    Q.push(root);

    int leafCount = 0;

    while (!Q.empty()) {
        BinaryTreeNode* temp = Q.front();
        Q.pop();
```

```
        if (temp->left == NULL && temp->right == NULL)
            leafCount++;

        if (temp->left)
            Q.push(temp->left);

        if (temp->right)
            Q.push(temp->right);
    }

    return leafCount;
}
```

Finding no of full nodes w/o recursion :

```
int countFullNodes(BinaryTreeNode* root) {
    if (root == NULL)
        return 0;

    queue<BinaryTreeNode*> Q;
    Q.push(root);

    int fullNodeCount = 0;

    while (!Q.empty()) {
        BinaryTreeNode* temp = Q.front();
        Q.pop();

        if (temp->left != NULL && temp->right != NULL)
            fullNodeCount++;

        if (temp->left)
            Q.push(temp->left);

        if (temp->right)
            Q.push(temp->right);
    }

    return fullNodeCount;
}
```

Finding no of half nodes w/o recursion :

```
int countHalfNodes(BinaryTreeNode* root) {
    if (root == NULL)
        return 0;
```

```
    queue<BinaryTreeNode*> Q;
    Q.push(root);

    int halfNodeCount = 0;

    while (!Q.empty()) {
        BinaryTreeNode* temp = Q.front();
        Q.pop();

        if ((temp->left != NULL && temp->right == NULL) ||
            (temp->left == NULL && temp->right != NULL)) {
            halfNodeCount++;
        }

        if (temp->left)
            Q.push(temp->left);

        if (temp->right)
            Q.push(temp->right);
    }

    return halfNodeCount;
}
```

Given two binary trees, return true if they are structurally similar :

```
bool areStructurallySimilar(BinaryTreeNode* root1, BinaryTreeNode*
root2) {
    if (root1 == NULL && root2 == NULL)
        return true;

    if (root1 == NULL || root2 == NULL)
        return false;

    // Use two queues for level-order traversal of both trees
    queue<BinaryTreeNode*> Q1, Q2;
    Q1.push(root1);
    Q2.push(root2);

    while (!Q1.empty() && !Q2.empty()) {
        BinaryTreeNode* temp1 = Q1.front();
        BinaryTreeNode* temp2 = Q2.front();
        Q1.pop();
        Q2.pop();

        // If one node is null and the other isn't, return false
```

```
        if ((temp1 == NULL && temp2 != NULL) || (temp1 != NULL &&
temp2 == NULL))
            return false;

        if (temp1 != NULL && temp2 != NULL) {
            Q1.push(temp1->left);
            Q1.push(temp1->right);

            Q2.push(temp2->left);
            Q2.push(temp2->right);
        }
    }

    return Q1.empty() && Q2.empty();
}
```

For finding the diameter of a BT. Diameter or width is the no of nodes on the longest path between two leaves in the tree :

Approach 1 : O(n2)

```
int height(TreeNode* root) {
    if (root == nullptr) return 0;
    return 1 + max(height(root->left), height(root->right));
}

int diameter(TreeNode* root) {
    if (root == nullptr) return 0;

    int lHeight = height(root->left);
    int rHeight = height(root->right);

    int lDiameter = diameter(root->left);
    int rDiameter = diameter(root->right);

    return max(lHeight + rHeight + 1, max(lDiameter, rDiameter));
}
```

Approach 2 : O(n)

```
struct TreeNode {
    int data;
    TreeNode* left;
    TreeNode* right;
    int nMaxLeft;
    int nMaxRight;
};
```

```
int findMaxLen(TreeNode* root) {
    if (root == nullptr) return 0;

    int nMaxLen = 0;
    if (root->left != nullptr) findMaxLen(root->left);
    if (root->right != nullptr) findMaxLen(root->right);

    if (root->left != nullptr) {
        root->nMaxLeft = max(root->left->nMaxLeft,
root->left->nMaxRight) + 1;
    } else {
        root->nMaxLeft = 0;
    }

    if (root->right != nullptr) {
        root->nMaxRight = max(root->right->nMaxLeft,
root->right->nMaxRight) + 1;
    } else {
        root->nMaxRight = 0;
    }

    nMaxLen = root->nMaxLeft + root->nMaxRight + 1;
    return nMaxLen;
}
```

Finding the level that has maximum sum in the BT?

```
struct TreeNode {
    int data;
    TreeNode* left;
    TreeNode* right;
};

// Function to find the level with the maximum sum
int FindLevelWithMaxSum(TreeNode* root) {
    if (!root) return 0;

    queue<TreeNode*> q;
    q.push(root);
    q.push(nullptr); // Marker for the end of a level

    int level = 0, maxLevel = 0, currentSum = 0, maxSum = 0;

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

```
        // If we reach the end of a level
        if (temp == nullptr) {
            if (currentSum > maxSum) {
                maxSum = currentSum;
                maxLevel = level;
            }
            currentSum = 0;
            if (!q.empty()) q.push(nullptr); // Add marker for the
next level
            level++;
        } else {
            currentSum += temp->data;
            if (temp->left) q.push(temp->left);
            if (temp->right) q.push(temp->right);
        }
    }

    return maxLevel;
}
```

Given a binary tree, print out all its root-to-leaf paths.

```
struct TreeNode {
    int data;
    TreeNode* left;
    TreeNode* right;
};

void PrintArray(int ints[], int len) {
    for (int i = 0; i < len; i++)
        cout << ints[i] << " ";
    cout << endl;
}
void PrintPathsRecur(TreeNode* root, int path[], int pathLen) {
    if (root == nullptr)
        return;

    path[pathLen] = root->data;
    pathLen++;

    if (root->left == nullptr && root->right == nullptr)
        PrintArray(path, pathLen);
    else {
        PrintPathsRecur(root->left, path, pathLen);
        PrintPathsRecur(root->right, path, pathLen);
```

```
    }
}
```

Given an algorithm for checking the existence of a path with a given sum. That means, given a sum, check whether there exists a path from the root to any of the nodes where the sum of the values equals the given sum :

```
struct TreeNode {
    int data;
    TreeNode* left;
    TreeNode* right;
};

bool HasPathSum(TreeNode* root, int remainingSum) {
    if (root == nullptr)
        return false;

    remainingSum -= root->data;

    if (root->left == nullptr && root->right == nullptr)
        return (remainingSum == 0);

    return HasPathSum(root->left, remainingSum) ||
HasPathSum(root->right, remainingSum);
}
```

Finding the sum of all elements in a BT :

Recursion :

```
int add(struct BinaryTreeNode *root) {
     if(root == NULL) {
          return 0;
     } else {
          return root->data + add(root->left) + add(root->right);
     }
}
```

Non Recursion :

```
struct TreeNode {
    int data;
    TreeNode* left;
    TreeNode* right;
};

int SumOfBTUsingLevelOrder(TreeNode* root) {
    if (!root)
        return 0;
```

```
    int sum = 0;
    queue<TreeNode*> Q;

    Q.push(root);

    while (!Q.empty()) {
        TreeNode* temp = Q.front();
        Q.pop();

        sum += temp->data;

        if (temp->left)
            Q.push(temp->left);

        if (temp->right)
            Q.push(temp->right);
    }

    return sum;
}
```

Give an algorithm for converting a tree to its mirror. A mirror of a tree is another tree with the left and right children of all non-leaf nodes interchanged :

```
TreeNode* MirrorOfBinaryTree(TreeNode* root) {
    if (root) {
        MirrorOfBinaryTree(root->left);
        MirrorOfBinaryTree(root->right);

        TreeNode* temp = root->left;
        root->left = root->right;
        root->right = temp;
    }
    return root;
}
```

Given two trees, give an algorithm for checking whether they are mirrors of each other :

```
bool AreMirrors(TreeNode* root1, TreeNode* root2) {
    if (root1 == nullptr && root2 == nullptr)
        return true;
    if (root1 == nullptr || root2 == nullptr)
        return false;
    if (root1->data != root2->data)
        return false;
```

```
    return AreMirrors(root1->left, root2->right) &&
AreMirrors(root1->right, root2->left);
}
```

Give an algorithm for finding the LCA (Least Common Ancestor) of two nodes in a Binary Tree :

```
TreeNode* LCA(TreeNode* root, TreeNode* a, TreeNode* b) {
    if (root == nullptr)
        return root;
    if (root == a || root == b)
        return root;

    TreeNode* left = LCA(root->left, a, b);
    TreeNode* right = LCA(root->right, a, b);

    if (left && right)
        return root;
    return (left) ? left : right;
}
```

Give an algorithm for constructing a binary tree from given preorder and inorder traversals :

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

TreeNode* buildTreeHelper(int preorder[], int inorder[], int&
preorderIndex, int inorderStart, int inorderEnd, unordered_map<int,
int>& inorderMap) {
    if (inorderStart > inorderEnd) {
        return nullptr;
    }

    int rootValue = preorder[preorderIndex++];
    TreeNode* root = new TreeNode(rootValue);

    int inorderIndex = inorderMap[rootValue];

    root->left = buildTreeHelper(preorder, inorder, preorderIndex,
inorderStart, inorderIndex - 1, inorderMap);
    root->right = buildTreeHelper(preorder, inorder, preorderIndex,
inorderIndex + 1, inorderEnd, inorderMap);
```

```
    return root;
}

TreeNode* buildTree(int preorder[], int inorder[], int n) {
    unordered_map<int, int> inorderMap;
    for (int i = 0; i < n; i++) {
        inorderMap[inorder[i]] = i;
    }

    int preorderIndex = 0;
    return buildTreeHelper(preorder, inorder, preorderIndex, 0, n -
1, inorderMap);
}
```

Print all ancestors for a node :

```
bool PrintAllAncestors(TreeNode* root, TreeNode* node) {
    if (root == nullptr) return false;

    if (root->left == node || root->right == node ||
        PrintAllAncestors(root->left, node) ||
PrintAllAncestors(root->right, node)) {
        cout << root->data << " ";
        return true;
    }

    return false;
}
```

Zig-zag traversal :

```
void ZigZagTraversal(TreeNode* root) {
    if (root == nullptr) return;

    stack<TreeNode*> currentLevel, nextLevel;
    currentLevel.push(root);
    bool leftToRight = true;

    while (!currentLevel.empty()) {
        TreeNode* temp = currentLevel.top();
        currentLevel.pop();

        if (temp) {
            cout << temp->data << " ";
```

```
            if (leftToRight) {
                if (temp->left) nextLevel.push(temp->left);
                if (temp->right) nextLevel.push(temp->right);
            } else {
                if (temp->right) nextLevel.push(temp->right);
                if (temp->left) nextLevel.push(temp->left);
            }
        }

        if (currentLevel.empty()) {
            leftToRight = !leftToRight;
            swap(currentLevel, nextLevel);
        }
    }
}
```

Vertical Sum :

```
void VerticalSumInBinaryTree(TreeNode* root, int column, map<int,
int>& hash) {
    if (root == nullptr)
        return;

    VerticalSumInBinaryTree(root->left, column - 1, hash);
    hash[column] += root->data;
    VerticalSumInBinaryTree(root->right, column + 1, hash);
}
```

Given a tree with a special property where leaves are represented with 'L' and internal nodes with 'I'. Also, assume that each node has either 0 or 2 children. Given the preorder traversal of this tree, construct the tree :

```
TreeNode* BuildTreeFromPreOrder(char A[], int* i) {
    TreeNode* newNode = (TreeNode*)malloc(sizeof(TreeNode));
    newNode->data = A[*i];
    newNode->left = newNode->right = nullptr;
    if (A[*i] == 'L') {
        return newNode;
    }
    (*i)++; // Move to the next element
    newNode->left = BuildTreeFromPreOrder(A, i);  // Build left
subtree
    (*i)++; // Move to the next element
    newNode->right = BuildTreeFromPreOrder(A, i); // Build right
subtree

    return newNode;
```

```
}
```

Given a binary tree with three pointers (left, right, and nextSibling), provide an algorithm to fill the nextSibling pointers, assuming they are initially NULL :

Approach 1 :

```
void FillNextSibling(BinaryTreeNode *root) {
    if (!root)
        return;

    queue<BinaryTreeNode *> q;
    q.push(root);
    q.push(nullptr); // Marker for the end of the current level

    while (!q.empty()) {
        BinaryTreeNode *temp = q.front();
        q.pop();

        if (temp == nullptr) {
            // End of the current level
            if (!q.empty())
                q.push(nullptr); // Marker for the next level
        } else {
            temp->nextSibling = q.front(); // Assign the nextSibling
pointer

            if (temp->left)
                q.push(temp->left);

            if (temp->right)
                q.push(temp->right);
        }
    }
}
```

Approach 2 :

```
struct BinaryTreeNode {
    BinaryTreeNode *left;
    BinaryTreeNode *right;
    BinaryTreeNode *nextSibling;
};

void FillNextSibling(BinaryTreeNode *root) {
    if (!root)
        return;
```

```
    if (root->left)
        root->left->nextSibling = root->right;

    if (root->right)
        root->right->nextSibling = (root->nextSibling) ?
root->nextSibling->left : nullptr;

    FillNextSibling(root->left);
    FillNextSibling(root->right);
}
```