11. Write a Program to verify and validate mirrored trees or not.

```
#include <stdio.h>
#include <stdlib.h>

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

// Function to create a new node
struct Node* newNode(int data) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

int areMirrors(struct Node* root1, struct Node* root2) {
    if (root1 == NULL && root2 == NULL)
        return 1;



    if (root1 == NULL || root2 == NULL)
        return 0;



    return (root1->data == root2->data) &&
           areMirrors(root1->left, root2->right) &&
           areMirrors(root1->right, root2->left);
}
```

```
// Main
int main() {
    // Creating two sample trees
    struct Node* root1 = newNode(1);
    root1->left = newNode(2);
    root1->right = newNode(3);
    root1->left->left = newNode(4);
    root1->left->right = newNode(5);

    struct Node* root2 = newNode(1);
    root2->left = newNode(3);
    root2->right = newNode(2);
    root2->right->left = newNode(5);
    root2->right->right = newNode(4);

    if (areMirrors(root1, root2))
        printf("The trees are mirrors of each other.\n");
    else
        printf("The trees are NOT mirrors of each other.\n");

    return 0;
}
```

Output: The trees are mirrors of each other.

12. Write a Program to determine the depth of a given Tree by Implementing MAXDEPTH

```
#include <stdio.h>
#include <stdlib.h>
```

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

struct Node* newNode(int data) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

int maxDepth(struct Node* root) {
    if (root == NULL)
        return 0;
    int leftDepth = maxDepth(root->left);
    int rightDepth = maxDepth(root->right);


    return (leftDepth > rightDepth ? leftDepth : rightDepth) + 1;
}

int main() {
    // Creating a sample tree
    struct Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
```

```
    root->left->right = newNode(5);
    root->right->right = newNode(6);
    root->left->left->left = newNode(7);

    printf("The depth of the tree is: %d\n", maxDepth(root));

    return 0;
}
```

Output: The depth of the tree is: 4

```
          1
         / \
        2   3
       / \    \
      4   5    6
     /
    7
```

13. Write a program for Lowest Common Ancestors

```
#include <stdio.h>
#include <stdlib.h>

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

```
struct Node* newNode(int data) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->data = data;
    node->left = node->right = NULL;
    return node;
}
struct Node* findLCA(struct Node* root, int n1, int n2) {
    if (root == NULL)
        return NULL;

    if (root->data == n1 || root->data == n2)
        return root;
    struct Node* leftLCA = findLCA(root->left, n1, n2);
    struct Node* rightLCA = findLCA(root->right, n1, n2);

    if (leftLCA != NULL && rightLCA != NULL)
        return root;



    return (leftLCA != NULL) ? leftLCA : rightLCA;
}

int main() {
    // Creating a sample tree
    struct Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
```

```
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);

    int n1 = 4, n2 = 5;
    struct Node* lca = findLCA(root, n1, n2);

    if (lca != NULL)
        printf("The Lowest Common Ancestor of %d and %d is: %d\n", n1, n2, lca->data);
    else
        printf("The Lowest Common Ancestor of %d and %d is not found.\n", n1, n2);

    return 0;
}
```

Output: The Lowest Common Ancestor of 4 and 5 is: 2

```
       1
      / \
     2   3
    / \  / \
   4   5 6  7
```

14. Write a Program to Build BST

```
#include <stdio.h>
#include <stdlib.h>

struct Node {
```

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

struct Node* newNode(int data) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

struct Node* insert(struct Node* root, int data) {
    if (root == NULL)
        return newNode(data);

    if (data < root->data)
        root->left = insert(root->left, data);
    else if (data > root->data)
        root->right = insert(root->right, data);

    return root;
}

void inorderTraversal(struct Node* root) {
    if (root == NULL)
        return;

    inorderTraversal(root->left);
```

```
    printf("%d ", root->data);
    inorderTraversal(root->right);
}

int main() {
    struct Node* root = NULL;

    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 70);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 60);
    root = insert(root, 80);



    printf("Inorder traversal of the BST: ");
    inorderTraversal(root);
    printf("\n");

    return 0;
}
```

Output: Inorder traversal of the BST: 20 30 40 50 60 70 80

Explaination :

Nodes are inserted into the Binary Search Tree in the following order: 50, 30, 70, 20, 40, 60, 80.

1. An inorder traversal of a BST visits nodes in ascending order.

2. The program traverses the BST recursively:
    - First visits the left subtree.
    - Then prints the root node.
    - Finally visits the right subtree.

Hence, the nodes are printed in sorted order: 20, 30, 40, 50, 60, 70, 80.

15. Write a Program for Building a Function ISVALID to VALIDATE BST

```
 #include <stdio.h>
#include <stdlib.h>

struct Node {
    int data;
    struct Node *left, *right;
};
struct Node* newNode(int data) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

struct Node* insert(struct Node* root, int data) {
    if (root == NULL)
        return newNode(data);

    if (data < root->data)
        root->left = insert(root->left, data);
```

```
    else if (data > root->data)
        root->right = insert(root->right, data);

    return root;
}

int isValidBSTUtil(struct Node* root, struct Node* minNode, struct Node* maxNode) {
    if (root == NULL)
        return 1;



    if ((minNode != NULL && root->data <= minNode->data) ||
        (maxNode != NULL && root->data >= maxNode->data))
        return 0;

    return isValidBSTUtil(root->left, minNode, root) &&
           isValidBSTUtil(root->right, root, maxNode);
}

int isValidBST(struct Node* root) {
    return isValidBSTUtil(root, NULL, NULL);
}

void inorderTraversal(struct Node* root) {
    if (root == NULL)
        return;

    inorderTraversal(root->left);
```

```
    printf("%d ", root->data);
    inorderTraversal(root->right);
}


int main() {
    struct Node* root = NULL;

    // Inserting nodes into the BST
    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 70);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 60);
    root = insert(root, 80);

    // Displaying the BST using inorder traversal
    printf("Inorder traversal of the BST: ");
    inorderTraversal(root);
    printf("\n");

    // Validating if the tree is a BST
    if (isValidBST(root))
        printf("The tree is a valid BST.\n");
    else
        printf("The tree is NOT a valid BST.\n");
```

```
    return 0;
}
```

Output: Inorder traversal of the BST: 20 30 40 50 60 70 80

The tree is a valid BST.

Explanation:

1. **Inorder Traversal**:
    - The program traverses the BST in an inorder manner, which for a valid BST will result in sorted node values. The nodes are displayed as 20, 30, 40, 50, 60, 70, 80.
2. **Validation**:
    - The isValidBST function checks whether the tree satisfies the BST properties:
        - For every node, all nodes in the left subtree must have values less than the node.
        - All nodes in the right subtree must have values greater than the node.
    - Since the tree meets these criteria, it is declared as a valid BST.