Experiment - 6

Name: Lucky Shukla **UID: 22BET10151**
Branch: BE-IT **Section: 22BET_IOT_703/A**
Semester: 6th **DOP: 25/02/25**
Subject: AP LAB-2 **Subject Code: 22ITH-359**

1. **Aim:** Maximum Depth of Binary Tree - To determine the maximum depth (or height) of a binary tree, which represents the longest path from the root node to a leaf node.

2. **Objective:**
1. Understand recursive depth-first traversal in binary trees.
2. Calculate the maximum depth by exploring all left and right subtrees.
3. Implement a solution that efficiently handles various tree structures.

3. **Code (C++):**

```
//HRITIK_22BET10113
#include <iostream>
using namespace std;

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;
   return 1 + max(maxDepth(root->left), maxDepth(root->right)); }
```

4. **Aim:** Validate Binary Search Tree- To verify if a given binary tree is a valid Binary Search Tree (BST) where the left subtree nodes are smaller, and the right subtree nodes are larger than the root.

5. **Objective:**
- Implement an in-order traversal to check the BST property.
- Ensure the tree satisfies all conditions of a valid BST.
- Handle edge cases like duplicate values and empty trees.

6. **Code (C++):**

```
//HRITIK_22BET10113
```

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

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

bool isValid(TreeNode* root, long minVal, long maxVal) {
if (!root) return true;
   if (root->val <= minVal || root->val >= maxVal) return false;
   return isValid(root->left, minVal, root->val) && isValid(root->right, root->val, maxVal); }

bool isValidBST(TreeNode* root) {
   return isValid(root, LONG_MIN, LONG_MAX);
}
```

7. **Aim:** Symmetric Tree- To determine if a binary tree is symmetric, meaning it is a mirror image of itself around its center.

8. **Objective:**

- Implement a recursive or iterative solution to check symmetry.
- Compare corresponding left and right subtrees for mirror properties.
- Handle edge cases like empty and single-node trees.

9. **Code (C++):**

```
//HRITIK_22BET10113
#include <iostream>
using namespace std;

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

bool isMirror(TreeNode* t1, TreeNode* t2) {
   if (!t1 && !t2) return true;
   if (!t1 || !t2) return false;
   return (t1->val == t2->val) && isMirror(t1->left, t2->right) && isMirror(t1->right, t2->left); }

bool isSymmetric(TreeNode* root) {
```

```
    return isMirror(root, root);
}
```

10. **Aim:** Binary Tree Level Order Traversal- To perform a level-order traversal of a binary tree, returning nodes level by level from top to bottom.

11. **Objective:**
- Implement breadth-first search (BFS) using a queue.
- Traverse the tree level by level.
- Return a list of nodes at each depth.

12. **Code (C++):**

```
//HRITIK_22BET10113
#include <iostream>
#include <vector> #include
<queue>
using namespace std;

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

vector<vector<int>> levelOrder(TreeNode* root) {
    vector<vector<int>> result;
    if (!root) return result;
    queue<TreeNode*> q;
q.push(root);      while
(!q.empty()) {
vector<int> level;
int size = q.size();
        for (int i = 0; i < size; i++) {
            TreeNode* node = q.front(); q.pop();
level.push_back(node->val);             if
(node->left) q.push(node->left);
            if (node->right) q.push(node->right);
        }
        result.push_back(level);
    }
    return result;
}
```

13. **Aim:** Convert Sorted Array to Binary Search Tree- To convert a sorted array into a height-balanced Binary Search Tree (BST).

14. **Objective:**
- Implement a recursive approach to divide the array.
- Construct a balanced BST from a sorted array.
- Ensure minimal tree height for optimal performance.

15. **Code (C++):**

```
//HRITIK_22BET10113
#include <iostream>
#include <vector> using
namespace std;

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

TreeNode* buildBST(vector<int>& nums, int left, int right) {
if (left > right) return nullptr;     int mid = left + (right - left) /
2;
   TreeNode* root = new TreeNode(nums[mid]);
root->left = buildBST(nums, left, mid - 1);     root-
>right = buildBST(nums, mid + 1, right);     return
root;
}

TreeNode* sortedArrayToBST(vector<int>& nums) {
return buildBST(nums, 0, nums.size() - 1); }
```

16. **Aim:** Binary Tree Inorder Traversal- To perform an in-order traversal of a binary tree and return the node values in left-root-right order.

17. **Objective:**
- Implement recursive and iterative in-order traversal.
- Collect the values of nodes in the correct order.
- Optimize the algorithm for time and space complexity.

18. **Code (C++):**

```
//HRITIK_22BET10113
#include <iostream> #include
<vector>
```

```
using namespace std;

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

void inorder(TreeNode* root, vector<int>& res) {
if (!root) return;     inorder(root->left, res);
res.push_back(root->val);      inorder(root->right,
res);
}

vector<int> inorderTraversal(TreeNode* root) {
vector<int> res;     inorder(root, res);     return
res;
}
```

19. **Aim:** Binary Zigzag Level Order Traversal-To perform a zigzag level order traversal of a binary tree, where nodes are traversed left-to-right on one level and right-to-left on the next.

20. **Objective:**

- Implement a breadth-first search (BFS) approach.
- Alternate the traversal direction on each level.
- Ensure efficient handling of large trees using queues and stacks.

21. **Code (C++):**

```
//HRITIK_22BET10113
#include <iostream>
#include <vector>
#include <queue> #include
<deque>
using namespace std;

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

vector<vector<int>> zigzagLevelOrder(TreeNode* root) {
vector<vector<int>> result;
   if (!root) return result;
```

```
    queue<TreeNode*> q;
q.push(root);
    bool leftToRight = true;

    while (!q.empty()) {
int size = q.size();
        deque<int> level;

        for (int i = 0; i < size; ++i) {
            TreeNode* node = q.front(); q.pop();
if (leftToRight) level.push_back(node->val);
else level.push_front(node->val);             if (node-
>left) q.push(node->left);
            if (node->right) q.push(node->right);
        }

        result.push_back(vector<int>(level.begin(), level.end()));
leftToRight = !leftToRight;
    }

    return result;
}
```

22. **Aim:** Construct Binary Tree from Inorder and Postorder Traversal- To construct a binary tree from given inorder and postorder traversal sequences.

23. **Objective:**

- Reconstruct the tree by identifying root nodes from postorder traversal.
- Use inorder traversal to segment left and right subtrees.
- Ensure correct tree structure and handle all input sizes.

24. **Code (C++):**

```
//HRITIK_22BET10113
#include <iostream>
#include <vector> #include
<unordered_map>
using namespace std;

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

```
TreeNode* buildTreeHelper(vector<int>& inorder, vector<int>& postorder, int& postIndex, int inLeft,
int inRight, unordered_map<int, int>& inMap) {     if (inLeft > inRight) return nullptr;

    int rootVal = postorder[postIndex--];
TreeNode* root = new TreeNode(rootVal);

    int inIndex = inMap[rootVal];     root->right = buildTreeHelper(inorder, postorder,
postIndex, inIndex + 1, inRight, inMap);     root->left = buildTreeHelper(inorder, postorder,
postIndex, inLeft, inIndex - 1, inMap);

    return root;
}

TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {
unordered_map<int, int> inMap;     for (int i = 0; i < inorder.size();
++i) {         inMap[inorder[i]] = i;
    }
    int postIndex = postorder.size() - 1;
    return buildTreeHelper(inorder, postorder, postIndex, 0, inorder.size() - 1, inMap); }
```

25. **Aim:** Kth Smallest Element in a BST- To find the kth smallest element in a Binary Search Tree (BST).

26. **Objective:**

- Implement in-order traversal to retrieve elements in sorted order.
- Efficiently find the kth element during traversal.
- Handle large trees while maintaining optimal performance.

27. **Code (C++):**

```
//HRITIK_22BET10113
#include <iostream>
using namespace std;

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

void inorder(TreeNode* root, int& k, int& result) {
if (!root) return;
    inorder(root->left, k, result);
if (--k == 0) {         result =
root->val;
        return;
```

```
    }
    inorder(root->right, k, result);
}

int kthSmallest(TreeNode* root, int k) {
int result = -1;     inorder(root, k,
result);     return result;
}
```

28. **Aim:** Populating Next Right Pointers in Each Node-To populate each node's next pointer to its right neighbor in a perfect binary tree.

29. **Objective:**

- Use level-order traversal to connect nodes on the same level.
- Optimize space by linking nodes without using extra storage.
- Handle all levels and edge cases efficiently.

30. **Code (C++):**

```
//HRITIK_22BET10113
#include <iostream> #include
<queue>
using namespace std;

struct Node {
    int val;
    Node* left;
    Node* right;
    Node* next;
    Node(int x) : val(x), left(NULL), right(NULL), next(NULL) {} };

Node* connect(Node* root) {
if (!root) return nullptr;
queue<Node*> q;
    q.push(root);

    while (!q.empty()) {
int size = q.size();
Node* prev = nullptr;

        for (int i = 0; i < size; ++i) {
Node* node = q.front();
q.pop();             if (prev) prev-
>next = node;             prev = node;

            if (node->left) q.push(node->left);
```

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

    return root;
}
```