Experiment 6(A)

Student Name: Tanmaya Kumar Pani **UID: 22BCS12986**
Branch: BE-CSE **Section/Group: IOT-613B**
Semester: 5 **Date of Performance: 10/9/24**
Subject Name: Advanced Programming Lab-1 **Subject Code: 22CSP-314**

1. Title: Tree: Top View

2. Objective:

Given a pointer to the root of a binary tree, print the top view of the binary tree.

The tree as seen from the top the nodes, is called the top view of the tree.

3. Algorithm:

a) Initialization:

- Create an empty map topNodes to store the first node encountered at each horizontal distance (HD).
- Create an empty queue q to facilitate level order traversal. The queue will store pairs of nodes and their corresponding HD values.
- Start by pushing the root node into the queue with an HD of 0.

b) Level Order Traversal:

- While the queue is not empty:
    1. Dequeue: Retrieve the front element of the queue, which contains a node and its HD.
    2. Node Processing:
        - If the current HD is not already present in the topNodes map, add it to the map with the node's data.
    3. Enqueue Children:
        - If the current node has a left child, enqueue the left child with an HD of current HD - 1.
        - If the current node has a right child, enqueue the right child with an HD of current HD + 1.

c) Output Top View:

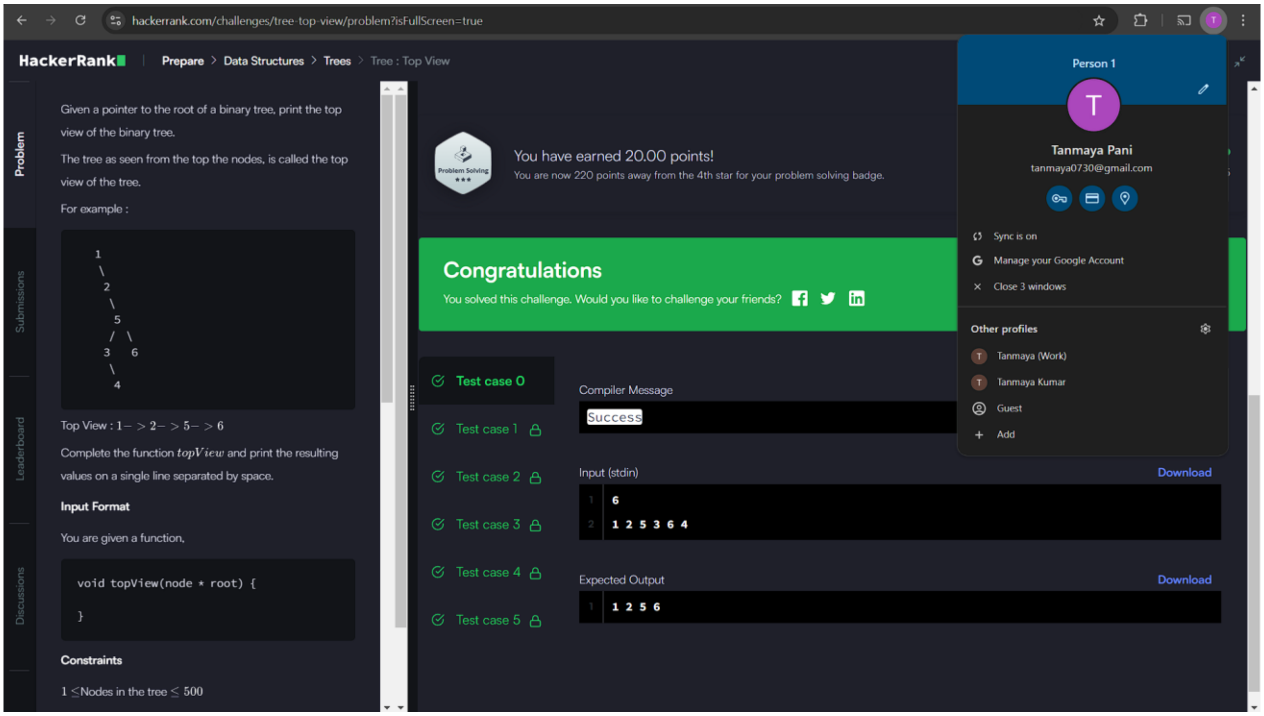
- Iterate through the topNodes map (sorted by HD) and print the values stored in the map. This gives the top view of the binary tree.

4. Implementation/Code:

```
#include<bits/stdc++.h>

using namespace std;

class Node {
    public:
        int data;
        Node *left;
        Node *right;
        Node(int d) {
            data = d;
            left = NULL;
            right = NULL;
        }
};

class Solution {
    public:
          Node* insertion (Node* root, int data) {
            if (root == NULL) {
                return new Node(data);
            } else {
                Node* cur;
                if(data <= root->data) {
                    cur = insertion(root->left, data);
                    root->left = cur;
                } else {
                    cur = insertion(root->right, data);
                    root->right = cur;
               }

               return root;
           }
        }

        void TOPVIEW(Node *root) {
    if (root == nullptr) {
        return;
    }

    // Step 1: Create map and queue
    map<int, int> topNodes;
    queue<pair<Node*, int>> q;
```

```
    // Step 2: Start with the root node
    q.push({root, 0});

    // Step 3: Level Order Traversal
    while (!q.empty()) {
        pair<Node*, int> p = q.front();
        q.pop();

        Node* current = p.first;
        int hd = p.second;

        // Step 3.1: Process current node
        if (topNodes.find(hd) == topNodes.end()) {
            topNodes[hd] = current->data;
        }

        // Step 3.2: Enqueue children
        if (current->left != nullptr) {
            q.push({current->left, hd - 1});
        }
        if (current->right != nullptr) {
            q.push({current->right, hd + 1});
        }
    }

    // Step 4: Print the top view
    for (auto it : topNodes) {
        cout << it.second << " ";
    }
}};

int main() {
    Solution myTree;
    Node* root = NULL;

    int t;
    int data;
    cin >> t;

    while(t-- > 0) {
        cin >> data;
        root = myTree.insertion(root, data);
    }

    myTree.TOPVIEW(root);
    return 0;
}
```

5. Output:



6. Learning Outcomes:

- Tree Traversal Techniques: Understand how to use level order traversal (breadth-first search) with a queue to visit nodes in a binary tree.
- Horizontal Distance Concept: Learn the concept of horizontal distance (HD) to determine the top view of the binary tree.
- Data Structures: Gain experience using a map to store the first node encountered at each HD and a queue for managing nodes during traversal.
- Algorithm Efficiency: Recognize the time and space complexity of the algorithm, which is O(N) for both, where N is the number of nodes.
- Practical Problem-Solving: Apply these techniques to solve the top view problem and enhance problem-solving skills with binary trees.

7. Time Complexity: O(n)

8. Space Complexity: O(n)