Name of a Student: Bhatraju Kejiya

Name of the Section: 9P179

Registration Number: 12323094

Roll Number: R9P179A55

Project Title : Binary Tree View System(Tree--Level Order)

Problem Statement: Dynamic Skyline View of a Binary Tree

A city skyline where buildings are arranged in a binary tree structure (each building can have at most two sub-buildings).
From a specific side view, only some buildings are visible because others are blocked.

You are given a binary tree. Each node represents a building with a height value.

Your task is to compute the Right Skyline View of the city:

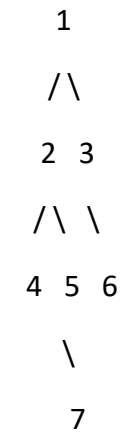
The Right Skyline View contains exactly one building per level:
the rightmost visible building when the city is viewed from the right side.

You are also given a list of queries:

- Each query removes (demolishes) one subtree.
- After each removal, you must output the new Right Skyline View of the tree.

Example 1:

Tree 1:



Input 1:

Queries[] = {5, 3, 2}

Output 1:

After removing subtree rooted at 5: [1, 3, 6]

After removing subtree rooted at 3: [1, 2, 4]

After removing subtree rooted at 2: [1, 3, 6]

Example 2:

Tree 2:

1

\

2

\

3

\

4

Input 2 :

Queries = {3, 2}

Output 2:

After removing subtree rooted at 3: [1, 2]

After removing subtree rooted at 2: [1]

Constraints :

- 1 ≤ N ≤ 10^5 (number of nodes)
- 1 ≤ Q ≤ 10^5 (number of queries)
- 1 ≤ Node.val ≤ 10^9 (unique values)

Topic Explanation: Binary Tree View System using Level Order Traversal

01) Introduction

In this project, we work with a Binary Tree data structure and design a system to compute the Right View of the tree using Level Order Traversal.

Additionally, the system supports dynamic queries, where for each query, a subtree is temporarily removed, and the Right View of the remaining tree is recalculated.

This type of problem is inspired by real-world scenarios such as:

- Visibility of buildings in a skyline
- Hierarchical systems where parts can be removed
- Network or organizational structures with dynamic changes

02) Binary Tree

A Binary Tree is a hierarchical data structure in which:

- Each node contains a value (data)
- Each node can have at most two children:
    - Left child
    - Right child

In this project:

- Each node is represented using a TreeNode class
- The tree is built using references (left and right)
- The root node represents the top of the hierarchy

03) Tree View Concept (Right View)

The Right View of a binary tree is defined as:

The set of nodes that are visible when the tree is viewed from the right side.

From each level of the tree:

- Only one node is visible
- That node is the rightmost node of that level

So, for every level in the tree, we select:

- The last node encountered in that level during traversal

04) Level Order Traversal (Breadth First Search - BFS)

Level Order Traversal is a method of traversing a tree level by level, from top to bottom and from left to right.

It is implemented using a Queue data structure:

Steps:

1. Insert the root node into the queue
2. While the queue is not empty:
    - Count the number of nodes at the current level
    - Process all nodes of that level one by one

    - Add their left and right children to the queue
3. Repeat until all levels are processed

Why BFS is used here?

- Because we need to process the tree level by level
- The Right View depends on identifying the last node of each level
- BFS naturally groups nodes by levels, making it ideal for this problem

05) Computing the Right View Using Level Order Traversal

During Level Order Traversal:

- For each level, we know how many nodes are present (size)
- While processing nodes of that level:
    - The first node is at index 0
    - The last node is at index size - 1
- The last node processed at each level is added to the Right View list

Thus:

- We get exactly one node per level
- This forms the Right View of the tree

06) Dynamic Subtree Removal Using Queries

In this project, we extend the basic Right View problem by introducing queries.

Each query:

- Specifies a node value
- The entire subtree rooted at that node is temporarily removed from the tree
- After removal:
    - The Right View of the remaining tree is recalculated using Level Order Traversal
- The subtree is then restored for the next query

This simulates dynamic changes in the tree structure.

07) Use of HashMap for Fast Access

To efficiently process queries:

- A HashMap is used to map:
    - node value → TreeNode reference
    - node value → parent node reference

This allows:

- **O(1)** access to any node
- Quick disconnection and restoration of subtrees
- Efficient handling of multiple queries

08) Data Structures Used

1. **Binary Tree**
    - To represent hierarchical data
2. **Queue (LinkedList)**
    - Used for Level Order Traversal (BFS)
3. **ArrayList**
    - To store the Right View result for each query
4. **HashMap**
    - For fast lookup of nodes and their parents during subtree removal and restoration

09) Algorithm Workflow

1. Build the binary tree
2. Create HashMaps to store:
    - Node → reference
    - Node → parent
3. For each query:
    - Temporarily remove the subtree rooted at the given node
    - Perform Level Order Traversal to compute the Right View
    - Store the result

- Restore the removed subtree

4. Print the result for all queries

10) **Time and Space Complexity**

Let:

- N = number of nodes in the tree
- Q = number of queries

For each query:

- Level Order Traversal takes **O(N)** time in worst case

So:

- **Time Complexity:** O(Q × N) (for this basic implementation)
- **Space Complexity:** O(N) (queue + maps + result storage)

Advantages:

1. Efficient Level-wise Processing
   The algorithm uses Level Order Traversal (BFS), which processes the tree level by level. This makes it very suitable for problems related to tree views, since each level is handled separately.
2. Clear and Simple Logic
   The Right View is obtained by selecting the last node of each level, which makes the logic easy to understand, implement, and debug.
3. Supports Dynamic Queries
   The system allows temporary removal of subtrees and recomputation of the view. This makes it useful for dynamic scenarios where the tree structure changes frequently.
4. Fast Node Access Using HashMap
   By using HashMap to store node and parent references, the program can locate and modify any subtree in O(1) time, which improves efficiency for query handling.
5. Scalable to Large Trees

   The approach works even for large trees (up to 10^5 nodes), as BFS and hash-based access are well-suited for large data sizes.

6. Real-World Applicability
   The concept can be applied to:
   - Visibility problems (skyline view)
   - Hierarchical systems (organization charts)
   - Network structures
   - Tree-based simulations and analysis systems
7. Uses Standard Data Structures
   The solution uses commonly known structures like Queue, ArrayList, HashMap, and Binary Tree, making it easy to understand for DSA learners and evaluators.

Disadvantages:

1. High Time Complexity for Many Queries
   For each query, the algorithm performs a full Level Order Traversal of the tree.
   So the time complexity becomes:

   $$O(Q \times N)$$

This can be slow when both N (nodes) and Q (queries) are very large.

2. Extra Memory Usage
   The use of:

- Queue for BFS
- HashMaps for node and parent storage
- Lists for storing results
  increases the space complexity to O(N), which may be memory-heavy for very large trees**.**

Code:

```
import java.util.*;


class TreeNode {
   int val;
   TreeNode left, right;
```

```
    TreeNode(int v) {
        val = v;
        left = right = null;
    }
}

public class Main {

    // Maps value -> node
    static Map<Integer, TreeNode> nodeMap = new HashMap<>();
    // Maps value -> parent
    static Map<Integer, TreeNode> parentMap = new HashMap<>();

    // Build maps for fast access
    static void buildMaps(TreeNode root, TreeNode parent) {
        if (root == null) return;

        nodeMap.put(root.val, root);
        parentMap.put(root.val, parent);

        buildMaps(root.left, root);
        buildMaps(root.right, root);
    }

    // Compute right view using level order traversal
    static List<Integer> rightView(TreeNode root) {
        List<Integer> res = new ArrayList<>();
        if (root == null) return res;

        Queue<TreeNode> q = new LinkedList<>();
```

```
    q.add(root);

    while (!q.isEmpty()) {
        int size = q.size();
        TreeNode last = null;

        for (int i = 0; i < size; i++) {
            TreeNode curr = q.poll();
            last = curr;

            if (curr.left != null) q.add(curr.left);
            if (curr.right != null) q.add(curr.right);
        }

        // last node of this level = right view
        res.add(last.val);
    }

    return res;
}

// Remove subtree rooted at val, return info needed to restore
static TreeNode[] removeSubtree(int val) {
    TreeNode node = nodeMap.get(val);
    TreeNode parent = parentMap.get(val);

    if (node == null || parent == null) {
        // Removing root or invalid
        return new TreeNode[]{null, null, null};
    }
```

```
    // Check if it's left or right child
    if (parent.left == node) {
        parent.left = null;
        return new TreeNode[]{parent, node, new TreeNode(0)}; // marker left
    } else {
        parent.right = null;
        return new TreeNode[]{parent, node, new TreeNode(1)}; // marker right
    }
}

// Restore subtree
static void restoreSubtree(TreeNode[] info) {
    if (info[0] == null) return;

    TreeNode parent = info[0];
    TreeNode node = info[1];
    int side = info[2].val; // 0 = left, 1 = right

    if (side == 0) parent.left = node;
    else parent.right = node;
}

// Main function for queries
static List<List<Integer>> processQueries(TreeNode root, int[] queries) {
    List<List<Integer>> answer = new ArrayList<>();

    for (int q : queries) {
        TreeNode[] info = removeSubtree(q);
```

```
        List<Integer> view = rightView(root);
        answer.add(view);

        restoreSubtree(info);
    }

    return answer;
}

public static void main(String[] args) {

    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);
    root.left.right.right = new TreeNode(7);

    // Build maps
    buildMaps(root, null);

    int[] queries = {5, 3, 2};

    List<List<Integer>> ans = processQueries(root, queries);

    for (int i = 0; i < ans.size(); i++) {
        System.out.println("After removing subtree rooted at " + queries[i] + ": " + ans.get(i));
    }
}
```

}

Output :

After removing subtree rooted at 5: [1, 3, 6]

After removing subtree rooted at 3: [1, 2, 5, 7]

After removing subtree rooted at 2: [1, 3, 6]

GDP link: https://onlinegdb.com/HlEsXIqxN

Conclusion :

This project demonstrates:

- Practical usage of Binary Trees
- Application of Level Order Traversal (BFS)
- Concept of Tree Views (Right View)
- Handling dynamic queries on tree structures
- Efficient use of HashMaps and Queues