**Title: Efficient Deletion in Binary Trees Using Inorder Traversal and Level-Order Rebuild**

**Overview:** While binary tree deletion typically involves node-level manipulation (especially replacing the node with the deepest-rightmost child), this document presents an alternative strategy suited for scenarios involving bulk deletions or value-based filtering: using inorder traversal to collect values, filtering them, and rebuilding the tree using a level-order insertion strategy.

This approach trades structural exactness for simplicity and scalability, making it useful in specific applications such as visual transformations, filtering, or data sanitization.

**Approach Breakdown:**

**Step 1: Perform Inorder Traversal** Traverse the binary tree in inorder (Left → Root → Right) and collect the node values in a list.

void inorder(TreeNode root, List<Integer> values) {  
 if (root == null) return;  
 inorder(root.left, values);  
 values.add(root.data);  
 inorder(root.right, values);  
}

**Step 2: Filter Values** Remove the target values from the list as needed (e.g., deleting all values that match a condition or a fixed set).

values.removeIf(x -> keysToDelete.contains(x));

**Step 3: Rebuild the Tree (Level-Order)** Construct a new binary tree from the filtered list using level-order (breadth-first) insertion.

TreeNode buildLevelOrderTree(List<Integer> values) {  
 if (values.isEmpty()) return null;  
  
 TreeNode root = new TreeNode(values.get(0));  
 Queue<TreeNode> q = new LinkedList<>();  
 q.add(root);  
 int i = 1;  
  
 while (i < values.size()) {  
 TreeNode current = q.poll();  
  
 if (i < values.size()) {  
 current.left = new TreeNode(values.get(i++));  
 q.add(current.left);  
 }  
  
 if (i < values.size()) {  
 current.right = new TreeNode(values.get(i++));  
 q.add(current.right);  
 }  
 }  
 return root;  
}

**Use Cases:**

* Bulk deletion or data cleanup
* Transformation pipelines where node identity is unimportant
* Educational tools to demonstrate value-level updates

**Not Suitable When:**

* Tree structure and parent-child relationships must be preserved
* Node identity (memory references) matters
* In-place memory efficiency is required

**Performance Notes:**

* Time Complexity: O(n) for traversal + O(n) for rebuild
* Space Complexity: O(n) for storing values + queue during rebuild

For large datasets, this approach is competitive and can outperform traditional deletion if many values are to be removed at once.

**Conclusion:** This strategy is a practical, scalable alternative to classic node-level deletion in binary trees. It emphasizes clarity, simplicity, and is especially effective when structural fidelity is not critical. Though not standard for in-place deletion, it can serve well in controlled or value-driven tree manipulation scenarios.