PRACTICE SET – 10

Meena Rhoshini C

Nov 25, 24

1. **Validate if the given tree is a BST or not**
   A Binary Search Tree is a tree in which:
   - The left subtree of a node contains only nodes with keys less than the node's key.
   - The right subtree of a node contains only nodes with keys greater than the node's key.
   - Both the left and right subtrees must also be BSTs.

   Approaches:

   a. **Using Java Collections:**

```
import java.util.*;

class BinaryTree {
   static class TreeNode {
      int val;
      TreeNode left, right;

      TreeNode(int val) {
         this.val = val;
         left = right = null;
      }
   }

   // In-order traversal to collect nodes in an ArrayList
   public static boolean isBST(TreeNode root) {
      List<Integer> inorderList = new ArrayList<>();
      inOrderTraversal(root, inorderList);

      // Check if the list is sorted
      for (int i = 1; i < inorderList.size(); i++) {
         if (inorderList.get(i) <= inorderList.get(i - 1)) {
            return false;
         }
      }
      return true;
   }

   // Helper method to do in-order traversal
   private static void inOrderTraversal(TreeNode node, List<Integer> inorderList) {
      if (node == null) {
         return;
      }
      inOrderTraversal(node.left, inorderList);
      inorderList.add(node.val);
      inOrderTraversal(node.right, inorderList);
```

```
    }

    public static void main(String[] args) {
        // Test the method
        TreeNode root = new TreeNode(10);
        root.left = new TreeNode(5);
        root.right = new TreeNode(15);
        root.left.left = new TreeNode(2);
        root.left.right = new TreeNode(7);

        System.out.println(isBST(root)); // Output: true

        root.right.left = new TreeNode(12);
        root.right.right = new TreeNode(20);

        System.out.println(isBST(root)); // Output: true

        root.right.left = new TreeNode(8); // This makes it not a BST
        System.out.println(isBST(root)); // Output: false
    }
}
```

Time complexity: O(n)
Space complexity: O(n)

b. Coding from Scratch (Without Java Collections using recursion):

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

        TreeNode(int val) {
            this.val = val;
            left = right = null;
        }
    }

    // Helper method to check if the tree is a BST
    public static boolean isBST(TreeNode root) {
        return isBSTUtil(root, Integer.MIN_VALUE, Integer.MAX_VALUE);
    }

    // Recursive method to check if the current node's value is within the valid range
    private static boolean isBSTUtil(TreeNode node, int min, int max) {
        if (node == null) {
            return true;
        }

        // Check if the current node's value is within the valid range
        if (node.val <= min || node.val >= max) {
            return false;
        }

        // Check the left and right subtrees with updated min/max values
        return isBSTUtil(node.left, min, node.val) && isBSTUtil(node.right, node.val, max);
    }

    public static void main(String[] args) {
        // Test the method
        TreeNode root = new TreeNode(10);
        root.left = new TreeNode(5);
        root.right = new TreeNode(15);
        root.left.left = new TreeNode(2);
        root.left.right = new TreeNode(7);

        System.out.println(isBST(root)); // Output: true

        root.right.left = new TreeNode(12);
        root.right.right = new TreeNode(20);

        System.out.println(isBST(root)); // Output: true

        root.right.left = new TreeNode(8); // This makes it not a BST
        System.out.println(isBST(root)); // Output: false
    }
}
```

Time complexity: O(n)
Space complexity: O(h)

2. Convert Binary Tree to Binary Search Tree(BST)

The tree follows the BST properties:

- **Left child** of a node must be smaller than the node.
- **Right child** of a node must be greater than the node.
- The **in-order traversal** of the tree must yield a sorted sequence of node values.

Approaches:

a. Convert Using Collections (List)

```
import java.util.*;

class BinaryTree {
    static class TreeNode {
        int val;
        TreeNode left, right;

        TreeNode(int val) {
            this.val = val;
            left = right = null;
        }
    }

    // Helper function to perform in-order traversal and collect node values
    private static void inorderTraversal(TreeNode root, List<Integer> values) {
        if (root == null) return;
        inorderTraversal(root.left, values);
        values.add(root.val);
        inorderTraversal(root.right, values);
    }

    // Helper function to reassign the sorted values to the tree
    private static void assignSortedValues(TreeNode root, List<Integer> values, int[] index) {
        if (root == null) return;
        assignSortedValues(root.left, values, index);
        root.val = values.get(index[0]++);
        assignSortedValues(root.right, values, index);
    }

    // Convert tree to BST using collections (ArrayList)
    public static void convertToBST(TreeNode root) {
        List<Integer> values = new ArrayList<>();
        inorderTraversal(root, values);  // Collect values in in-order
        Collections.sort(values);  // Sort the values

        int[] index = {0};
        assignSortedValues(root, values, index);  // Assign sorted values back to the tree
    }

    public static void main(String[] args) {
        // Test the conversion
        TreeNode root = new TreeNode(10);
        root.left = new TreeNode(5);
        root.right = new TreeNode(15);
        root.left.left = new TreeNode(2);
        root.left.right = new TreeNode(7);

        convertToBST(root);
        // After conversion, the tree will follow BST properties
        System.out.println(root.val);  // Should output the middle value after sorting
    }
}
```

Time complexity: O(nlogn)

- In-order traversal: O(n)
- Sorting: O(nlogn)
- Re-assigning sorted values: O(n)

Space complexity: O(n)

b. Convert Without Collections (In-place)

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

        TreeNode(int val) {
            this.val = val;
            left = right = null;
        }
    }

    // Helper function to perform in-order traversal and collect node values
    private static void inorderTraversal(TreeNode root, List<Integer> values) {
        if (root == null) return;
        inorderTraversal(root.left, values);
        values.add(root.val);
        inorderTraversal(root.right, values);
    }

    // Helper function to reassign sorted values from the array back to the tree
    private static void assignSortedValues(TreeNode root, int[] sortedValues, int[] index) {
        if (root == null) return;
        assignSortedValues(root.left, sortedValues, index);
        root.val = sortedValues[index[0]++];
        assignSortedValues(root.right, sortedValues, index);
    }

    // Convert tree to BST using in-order traversal and sorting the node values
    public static void convertToBST(TreeNode root) {
        List<Integer> values = new ArrayList<>();
        inorderTraversal(root, values);  // Collect node values in in-order

        // Convert list to array and sort the array
        int[] sortedValues = values.stream().mapToInt(i -> i).toArray();
        Arrays.sort(sortedValues);

        // Reassign the sorted values back to the tree
        int[] index = {0};
        assignSortedValues(root, sortedValues, index);
    }

    public static void main(String[] args) {
        // Test the conversion
        TreeNode root = new TreeNode(10);
        root.left = new TreeNode(5);
        root.right = new TreeNode(15);
        root.left.left = new TreeNode(2);
        root.left.right = new TreeNode(7);

        convertToBST(root);
        // After conversion, the tree will follow BST properties
        System.out.println(root.val);  // Should output the middle value after sorting
    }
}
```

Time complexity: O(nlogn)
- In-order traversal: O(n)
- Sorting: O(nlogn)
- Re-assigning sorted values: O(n)

Space complexity: O(n)

---

3. **Top View of a Binary Search Tree (BST)**

The top view of a binary tree is the set of nodes that are visible when the tree is viewed from the top. For this problem, we need to find the nodes that are first encountered at each horizontal distance from the root.

Approaches:

a. **Top View of BST Using Collections (Queue and HashMap)**

Steps:
- Perform **level-order traversal** (BFS) using a queue.
- For each node, calculate its horizontal distance from the root.
- If a node is the first to be encountered at a particular horizontal distance, add it to the **hash map**.
- After the traversal, the nodes in the hash map will be the top view nodes, sorted by horizontal distance.

```
import java.util.*;

class BinaryTree {
    static class TreeNode {
        int val;
        TreeNode left, right;

        TreeNode(int val) {
            this.val = val;
            left = right = null;
        }
    }

    // Helper class to store nodes along with their horizontal distance
    static class NodeWithHD {
        TreeNode node;
        int hd;  // horizontal distance
        NodeWithHD(TreeNode node, int hd) {
            this.node = node;
            this.hd = hd;
        }
    }

    // Top view using collections (Queue and HashMap)
    public static void topView(TreeNode root) {
        if (root == null) return;

        // Queue for BFS
        Queue<NodeWithHD> queue = new LinkedList<>();
        // Map to store the top view nodes at each horizontal distance
        Map<Integer, Integer> map = new TreeMap<>();

        queue.add(new NodeWithHD(root, 0)); // root at horizontal distance 0

        while (!queue.isEmpty()) {
            NodeWithHD current = queue.poll();
            TreeNode currentNode = current.node;
            int hd = current.hd;

            // Add the node to map if it's not already present (first encountered at this
horizontal distance)
            if (!map.containsKey(hd)) {
                map.put(hd, currentNode.val);
            }

            // Add left and right children to the queue with updated horizontal distances
            if (currentNode.left != null) {
                queue.add(new NodeWithHD(currentNode.left, hd - 1));
            }
            if (currentNode.right != null) {
                queue.add(new NodeWithHD(currentNode.right, hd + 1));
            }
        }

        // Print the top view
        for (int value : map.values()) {
            System.out.print(value + " ");
        }
    }

    public static void main(String[] args) {
        // Create a sample tree
        TreeNode root = new TreeNode(10);
        root.left = new TreeNode(5);
        root.right = new TreeNode(15);
        root.left.left = new TreeNode(3);
        root.left.right = new TreeNode(7);
        root.right.right = new TreeNode(18);

        // Display the top view
        topView(root);  // Output: 3 5 10 15 18
    }
}
```

Time complexity: O(n)
Space complexity: O(n)

b. Optimized Top View of BST Without Collections (Using Only a Queue)
Steps:

- Perform **level-order traversal** (BFS) using a queue.
- For each node, calculate its horizontal distance.
- Maintain a set to track horizontal distances that have already been encountered.
- Add the first node encountered at each horizontal distance to the result.

```
import java.util.*;

class BinaryTree {
    static class TreeNode {
        int val;
        TreeNode left, right;

        TreeNode(int val) {
            this.val = val;
            left = right = null;
        }
    }

    // Optimized Top view without collections (using only queue)
    public static void topView(TreeNode root) {
        if (root == null) return;

        // Queue for BFS
        Queue<TreeNode> queue = new LinkedList<>();
        // Track horizontal distance with corresponding node
        Queue<Integer> hdQueue = new LinkedList<>();
        // Set to track horizontal distances already visited
        Set<Integer> visited = new HashSet<>();

        queue.add(root); // root node
        hdQueue.add(0);  // root's horizontal distance
        visited.add(0);  // mark horizontal distance 0 as visited

        while (!queue.isEmpty()) {
            TreeNode current = queue.poll();
            int hd = hdQueue.poll();
```

```
                // If it's the first node at this horizontal distance, print it
                if (!visited.contains(hd)) {
                    System.out.print(current.val + " ");
                    visited.add(hd);  // Mark this horizontal distance as visited
                }

                // Add left and right children to the queue with updated horizontal distances
                if (current.left != null) {
                    queue.add(current.left);
                    hdQueue.add(hd - 1);
                }
                if (current.right != null) {
                    queue.add(current.right);
                    hdQueue.add(hd + 1);
                }
            }
        }

        public static void main(String[] args) {
            // Create a sample tree
            TreeNode root = new TreeNode(10);
            root.left = new TreeNode(5);
            root.right = new TreeNode(15);
            root.left.left = new TreeNode(3);
            root.left.right = new TreeNode(7);
            root.right.right = new TreeNode(18);

            // Display the top view
            topView(root);  // Output: 3 5 10 15 18
        }
    }
```
Time complexity: O(n)
Spacecomplexity: O(n)

---

4. Bottom View of a Binary Search Tree (BST)

The bottom view of a binary tree is the set of nodes that are visible when the tree is viewed from the bottom. The nodes at each horizontal distance from the root that are the lowest (i.e., deepest) are considered as part of the bottom view.

Approaches:

a. Bottom View of BST Using Collections (Queue and HashMap)

Steps:

- Perform **level-order traversal (BFS)** using a queue.
- For each node, calculate its horizontal distance.
- Store the last node encountered at each horizontal distance in a **map**.
- After the traversal, the nodes in the map will represent the bottom view, sorted by horizontal distance.

```java
import java.util.*;

class BinaryTree {
    static class TreeNode {
        int val;
        TreeNode left, right;

        TreeNode(int val) {
            this.val = val;
            left = right = null;
        }
    }

    // Helper class to store nodes along with their horizontal distance
    static class NodeWithHD {
        TreeNode node;
        int hd;  // horizontal distance

        NodeWithHD(TreeNode node, int hd) {
            this.node = node;
            this.hd = hd;
        }
    }

    // Bottom view using collections (Queue and HashMap)
    public static void bottomView(TreeNode root) {
        if (root == null) return;

        // Queue for BFS
        Queue<NodeWithHD> queue = new LinkedList<>();
        // Map to store the bottom view nodes at each horizontal distance
        Map<Integer, Integer> map = new TreeMap<>();

        queue.add(new NodeWithHD(root, 0)); // root at horizontal distance 0

        while (!queue.isEmpty()) {
            NodeWithHD current = queue.poll();
            TreeNode currentNode = current.node;
            int hd = current.hd;

            // Update the map with the most recent node at this horizontal distance
            map.put(hd, currentNode.val);

            // Add left and right children to the queue with updated horizontal distances
            if (currentNode.left != null) {
                queue.add(new NodeWithHD(currentNode.left, hd - 1));
            }
            if (currentNode.right != null) {
                queue.add(new NodeWithHD(currentNode.right, hd + 1));
            }
        }

        // Print the bottom view
        for (int value : map.values()) {
            System.out.print(value + " ");
        }
    }

    public static void main(String[] args) {
        // Create a sample tree
        TreeNode root = new TreeNode(10);
        root.left = new TreeNode(5);
        root.right = new TreeNode(15);
        root.left.left = new TreeNode(3);
        root.left.right = new TreeNode(7);
        root.right.right = new TreeNode(18);

        // Display the bottom view
        bottomView(root);  // Output: 3 7 10 15 18
    }
}
```

Time complexity: O(n)
Space complexity: O(n)

b. Optimized Bottom View of BST Without Collections (Using Only Queue)
Steps:

- Perform **level-order traversal** (BFS) using a queue.
- For each node, calculate its horizontal distance.
- Maintain a set to track horizontal distances that have already been visited.
- The first node encountered at each horizontal distance will be the bottommost node.

```
import java.util.*;
class BinaryTree {
    static class TreeNode {
        int val;
        TreeNode left, right;

        TreeNode(int val) {
            this.val = val;
            left = right = null;
        }
    }

    // Optimized bottom view without collections (using only queue)
    public static void bottomView(TreeNode root) {
        if (root == null) return;
        // Queue for BFS
        Queue<TreeNode> queue = new LinkedList<>();
        // Track horizontal distance with corresponding node
        Queue<Integer> hdQueue = new LinkedList<>();
        // Set to track horizontal distances already visited
        Set<Integer> visited = new HashSet<>();

        queue.add(root); // root node
        hdQueue.add(0);  // root's horizontal distance
        visited.add(0);  // mark horizontal distance 0 as visited

        while (!queue.isEmpty()) {
            TreeNode current = queue.poll();
            int hd = hdQueue.poll();

            // If it's the first node at this horizontal distance, print it
            if (!visited.contains(hd)) {
                System.out.print(current.val + " ");
                visited.add(hd);  // Mark this horizontal distance as visited
            }

            // Add left and right children to the queue with updated horizontal distances
            if (current.left != null) {
                queue.add(current.left);
                hdQueue.add(hd - 1);
            }
            if (current.right != null) {
                queue.add(current.right);
                hdQueue.add(hd + 1);
            }
        }
    }

    public static void main(String[] args) {
        // Create a sample tree
        TreeNode root = new TreeNode(10);
        root.left = new TreeNode(5);
        root.right = new TreeNode(15);
        root.left.left = new TreeNode(3);
        root.left.right = new TreeNode(7);
        root.right.right = new TreeNode(18);

        // Display the bottom view
        bottomView(root);  // Output: 3 7 10 15 18
    }
}
```

Time complexity: O(n)
Space complexity: O(n)

5. **Left View of a Binary Search Tree (BST)**
The left view of a binary tree is the set of nodes that are visible when the tree is viewed from the left side. Only the first node encountered at each level (from left to right) is part of the left view.

Approaches:

a. **Left View of BST Using Collections (Queue and HashMap)**
Steps:

- Perform **level-order traversal** (BFS) using a queue.
- At each level, only include the first node encountered at that level in the left view.
- Use a **hashmap** or a set to store visited levels, ensuring that we only take the first node at each level.

```
import java.util.*;
class BinaryTree {
    static class TreeNode {
        int val;
        TreeNode left, right;

        TreeNode(int val) {
            this.val = val;
            left = right = null;
        }
    }

    // Left view using collections (Queue and HashMap)
    public static void leftView(TreeNode root) {
        if (root == null) return;

        // Queue for BFS
        Queue<TreeNode> queue = new LinkedList<>();
        // Add the root node to the queue
        queue.add(root);

        while (!queue.isEmpty()) {
            int levelSize = queue.size();
            // Traverse all nodes at the current level
            for (int i = 0; i < levelSize; i++) {
                TreeNode currentNode = queue.poll();

                // If it's the first node at this level, print it
                if (i == 0) {
                    System.out.print(currentNode.val + " ");
                }

                // Add the left and right children of the current node to the queue
                if (currentNode.left != null) {
                    queue.add(currentNode.left);
                }
                if (currentNode.right != null) {
                    queue.add(currentNode.right);
                }
            }
        }
    }

    public static void main(String[] args) {
        // Create a sample tree
        TreeNode root = new TreeNode(10);
        root.left = new TreeNode(5);
        root.right = new TreeNode(15);
        root.left.left = new TreeNode(3);
        root.left.right = new TreeNode(7);
        root.right.right = new TreeNode(18);

        // Display the left view
        leftView(root);  // Output: 10 5 3
    }
}
```

Time complexity: O(n)
Space complexity: O(n)

b. Optimized Left View of BST Without Collections (Using Only Queue)
Steps:

- Perform **level-order traversal (BFS)** using a queue.
- At each level, only include the first node encountered at that level in the left view.

```
import java.util.*;

class BinaryTree {
    static class TreeNode {
        int val;
        TreeNode left, right;

        TreeNode(int val) {
            this.val = val;
            left = right = null;
        }
    }

    // Optimized left view without collections (using only queue)
    public static void leftView(TreeNode root) {
        if (root == null) return;

        // Queue for BFS
        Queue<TreeNode> queue = new LinkedList<>();
        queue.add(root);
        while (!queue.isEmpty()) {
            int levelSize = queue.size();
            // Traverse all nodes at the current level
            for (int i = 0; i < levelSize; i++) {
                TreeNode currentNode = queue.poll();

                // Print the first node of each level (leftmost node)
                if (i == 0) {
                    System.out.print(currentNode.val + " ");
                }

                // Add left and right children to the queue
                if (currentNode.left != null) {
                    queue.add(currentNode.left);
                }
                if (currentNode.right != null) {
                    queue.add(currentNode.right);
                }
            }
        }
    }

    public static void main(String[] args) {
        // Create a sample tree
        TreeNode root = new TreeNode(10);
        root.left = new TreeNode(5);
        root.right = new TreeNode(15);
        root.left.left = new TreeNode(3);
        root.left.right = new TreeNode(7);
        root.right.right = new TreeNode(18);

        // Display the left view
        leftView(root);  // Output: 10 5 3
    }
}
```

Time complexity: O(n)
Space complexity: O(n)

---

6. **Right View of a Binary Search Tree (BST)**
The right view of a binary tree consists of the set of nodes visible when the tree is viewed from the right side. Only the last node encountered at each level (from left to right) is part of the right view.

Approaches:

a. **Right View of BST Using Collections (Queue)**
Steps:
- Perform **level-order traversal (BFS)** using a queue.

- At each level, the last node encountered is part of the right view.
- Use the queue to traverse the nodes, ensuring we print the last node at each level.

```
import java.util.*;

class BinaryTree {
    static class TreeNode {
        int val;
        TreeNode left, right;

        TreeNode(int val) {
            this.val = val;
            left = right = null;
        }
    }

    // Right view using collections (Queue)
    public static void rightView(TreeNode root) {
        if (root == null) return;

        // Queue for BFS
        Queue<TreeNode> queue = new LinkedList<>();
        // Add root node to the queue
        queue.add(root);

        while (!queue.isEmpty()) {
            int levelSize = queue.size();
            // Traverse all nodes at the current level
            for (int i = 0; i < levelSize; i++) {
                TreeNode currentNode = queue.poll();

                // If it's the last node at this level, print it
                if (i == levelSize - 1) {
                    System.out.print(currentNode.val + " ");
                }

                // Add left and right children of the current node to the queue
                if (currentNode.left != null) {
                    queue.add(currentNode.left);
                }
                if (currentNode.right != null) {
                    queue.add(currentNode.right);
                }
            }
        }
    }

    public static void main(String[] args) {
        // Create a sample tree
        TreeNode root = new TreeNode(10);
        root.left = new TreeNode(5);
        root.right = new TreeNode(15);
        root.left.left = new TreeNode(3);
        root.left.right = new TreeNode(7);
        root.right.right = new TreeNode(18);

        // Display the right view
        rightView(root);  // Output: 10 15 18
    }
}
```

Time complexity: O(n)
Space complexity: O(n)

b. Optimized Right View of BST Without Collections (Using Queue)
Steps:

- Perform **level-order traversal (BFS)** using a queue.
- Track and print the last node encountered at each level.

```
import java.util.*;

class BinaryTree {
    static class TreeNode {
        int val;
        TreeNode left, right;

        TreeNode(int val) {
            this.val = val;
            left = right = null;
        }
    }

    // Optimized right view without collections (using only queue)
    public static void rightView(TreeNode root) {
        if (root == null) return;

        // Queue for BFS
        Queue<TreeNode> queue = new LinkedList<>();
        queue.add(root);

        while (!queue.isEmpty()) {
            int levelSize = queue.size();
            // Traverse all nodes at the current level
            for (int i = 0; i < levelSize; i++) {
                TreeNode currentNode = queue.poll();

                // If it's the last node at this level, print it
                if (i == levelSize - 1) {
                    System.out.print(currentNode.val + " ");
                }
                // Add left and right children of the current node to the queue
                if (currentNode.left != null) {
                    queue.add(currentNode.left);
                }
                if (currentNode.right != null) {
                    queue.add(currentNode.right);
                }
            }
        }
    }

    public static void main(String[] args) {
        // Create a sample tree
        TreeNode root = new TreeNode(10);
        root.left = new TreeNode(5);
        root.right = new TreeNode(15);
        root.left.left = new TreeNode(3);
        root.left.right = new TreeNode(7);
        root.right.right = new TreeNode(18);

        // Display the right view
        rightView(root);  // Output: 10 15 18
    }
}
```

Time complexity: O(n)
Space complexity: O(n)