1.

Binary Tree - Organizing a Cricket Tournament:

In a cricket tournament, each team is ranked based on their performance. The teams need to be organized in a binary tree structure to quickly manage and track matchups. The binary tree represents the tournament bracket, where each node corresponds to a team, and the tree helps efficiently find opponents and record match results.

As the organizer, you need to insert new teams into this binary tree structure based on their registration order. Your task is to create a system that inserts teams into the binary tree and then displays the teams in a level-order (top-to-bottom) manner, so you can view the bracket.

Input Format

1. First line contains the number of teams to be inserted.
2. Next line contains the team rankings (as integers) to be inserted into the binary tree in sequence. *Italic Text*

Constraints

NA

Output Format

After all teams are inserted, display the team rankings in level-order traversal.

Sample Input 0

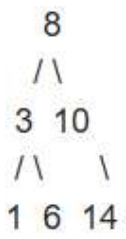
```
6
8 3 10 1 6 14
```

Sample Output 0

```
8 3 10 1 6 14
```

Explanation 0

- The binary tree starts empty.
- The first team with ranking 8 becomes the root.
- The second team with ranking 3 becomes the left child of 8.
- The third team with ranking 10 becomes the right child of 8.
- The fourth team with ranking 1 becomes the left child of 3.
- The fifth team with ranking 6 becomes the right child of 3.
- The sixth team with ranking 14 becomes the right child of 10.



Sample Input 1

```
5
7 4 9 2 5
```

Sample Output 1

```
7 4 9 2 5
```

Program :

```
import java.util.Scanner;

class Node {
    int val;
```

```
    Node left, right;
    Node(int v) { val = v; }
}

class Queue {
    class QNode {
        Node node;
        QNode next;
        QNode(Node n) { node = n; }
    }

    QNode front, rear;

    void add(Node n) {
        QNode q = new QNode(n);
        if (rear == null) front = rear = q;
        else rear = rear.next = q;
    }

    Node remove() {
        if (front == null) return null;
        Node n = front.node;
        front = front.next;
        if (front == null) rear = null;
        return n;
    }

    boolean isEmpty() { return front == null; }
}

public class Main {
    static Node insert(Node root, int val) {
        if (root == null) return new Node(val);
        if (val < root.val) root.left =
insert(root.left, val);
```

```
            else root.right = insert(root.right, val);
            return root;
        }

        static void levelOrder(Node root) {
            Queue q = new Queue();
            q.add(root);
            while (!q.isEmpty()) {
                Node n = q.remove();
                System.out.print(n.val + " ");
                if (n.left != null) q.add(n.left);
                if (n.right != null) q.add(n.right);
            }
        }

        public static void main(String[] args) {
            Scanner sc = new Scanner(System.in);
            int n = sc.nextInt();
            Node root = null;
            for (int i = 0; i < n; i++)
                root = insert(root, sc.nextInt());
            levelOrder(root);
        }
}
```

2.

In a cricket tournament, teams are ranked based on their performance, and the organizer uses a binary tree to structure the teams' rankings. The rankings are represented in such a way that:

- **Each node in the binary tree represents a team's ranking.**
- **The inorder traversal of the binary tree gives the rankings of the teams based on the number of matches they had won.**

Given a list of rankings for teams, you need to perform an inorder traversal of the binary tree to obtain the correct sequence of team rankings.

Input Format

1. First line contains the number of nodes (teams).
2. Next line contains the node values (team rankings) in level order for constructing the binary tree.

Constraints

NA

Output Format

A single line with the result of the inorder traversal of the binary tree

Sample Input 0

```
4
1 null 2 3
```

Sample Output 0

```
1 3 2
```

Explanation 0

- The first value 1 is the root of the binary tree.
- The second value null means the root 1 has no left child.
- The third value 2 is the right child of the root 1.
- The fourth value 3 is the left child of node 2.

The structure of the binary tree will look like this:

The inorder traversal visits nodes in the order:

- **Left subtree.**
- **Root.**
- **Right subtree.**

For the given binary tree:

- **First, it visits 1 (root).**
- **Then, it visits the left subtree of node 2, which is 3.**
- **Finally, it visits node 2.**

Thus, the inorder traversal output is 1 3 2.

Sample Input 1

```
5
10 5 15 null null 12 18
```

Sample Output 1

```
5 10 12 15 18
```

```java
import java.util.Scanner;

class TreeNode {
    int val;
    TreeNode left, right;
    TreeNode(int v) {
        val = v;
    }
}

class ListNode {
    int val;
    ListNode next;
    ListNode(int v) {
        val = v;
    }
}

public class Main {
    // Builds a binary tree from level-order (null
for absent) using a manual array of references
    static TreeNode buildTree(String[] vals) {
        if (vals.length == 0 ||
vals[0].equals("null")) {
            return null;
        }
        TreeNode root = new
TreeNode(Integer.parseInt(vals[0]));
        TreeNode[] nodes = new
TreeNode[vals.length];
        nodes[0] = root;
        int i = 1;
        int j = 0;
        while (i < vals.length) {
            TreeNode parent = nodes[j++];
```

```
            if (parent != null) {
                if (i < vals.length &&
!vals[i].equals("null")) {
                    parent.left = nodes[i] = new
TreeNode(Integer.parseInt(vals[i]));
                }
                i++;
                if (i < vals.length &&
!vals[i].equals("null")) {
                    parent.right = nodes[i] = new
TreeNode(Integer.parseInt(vals[i]));
                }
                i++;
            }
        }
        return root;
    }

    // Recursively build a linked list from
inorder traversal
    static ListNode inorderToList(TreeNode root) {
        return inorderHelper(root)[0];
    }

    // Helper returns [head, tail] of the produced
list for easy connection
    static ListNode[] inorderHelper(TreeNode root)
{
        if (root == null) {
            return new ListNode[] {null, null};
        }
        ListNode[] left =
inorderHelper(root.left);
        ListNode mid = new ListNode(root.val);
```

```
        ListNode[] right =
inorderHelper(root.right);
        // Attach left tail to current node
        if (left[1] != null) {
            left[1].next = mid;
        }
        // Attach current node to right head
        mid.next = right[0];
        ListNode head = (left[0] != null) ?
left[0] : mid;
        ListNode tail = (right[1] != null) ?
right[1] : mid;
        return new ListNode[] {head, tail};
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = Integer.parseInt(sc.nextLine());
        String[] vals =
sc.nextLine().trim().split(" ");
        TreeNode root = buildTree(vals);
        ListNode head = inorderToList(root);
        while (head != null) {
            System.out.print(head.val);
            head = head.next;
            if (head != null) {
                System.out.print(" ");
            }
        }
    }
}
```

3.

In a cricket tournament, every team has several players, and among them are left-handed players. Each player is represented by a node in a binary tree, where the left and right children represent players in different roles. The tournament organizers want to calculate the total contribution (score) made by all left-handed players who are not further managing other players, i.e., left-handed players who are "leaves" (players with no further responsibilities).

The binary tree structure can help represent the hierarchy of the players, and our goal is to find the sum of the scores of all left-handed players who are at the end of the hierarchy (leaf players).

Input Format

1. **First line contains the number of nodes (players).**
2. **The next line contains the values of nodes (scores of players) in level-order (use "null" for no player at that posit**

Constraints

NA

Output Format

A single integer representing the sum of the scores of all left-handed (left child) leaf players.

Sample Input 0

```
7
3 9 20 null null 15 7
```

Sample Output 0

24

Explanation 0

- **The first value 3 is the root of the binary tree (captain).**
- **The second and third values 9 and 20 represent the left and right players of 3.**
- **The values null indicate that player 9 has no children.**
- **The next two values 15 and 7 represent the left and right players of player 20.**

The structure of the binary tree looks like this:

```
 3
/ \
9  20
  / \
 15 7
```

The left leaf in this binary tree is 9, and the left leaf of node 20 is 15. Therefore, the sum of the left leaf nodes is 9 + 15 = 24.

Sample Input 1

```
7
5 3 8 null null 6 10
```

Sample Output 1

```
9
```

Program:

```
import java.util.Scanner;

class TreeNode {
    int val;
    TreeNode left, right;
    TreeNode(int v) { val = v; }
}

public class Main {
    static TreeNode buildTree(String[] vals) {
        if (vals.length == 0 ||
vals[0].equals("null")) return null;
        TreeNode[] nodes = new
TreeNode[vals.length];
        TreeNode root = nodes[0] = new
TreeNode(Integer.parseInt(vals[0]));
        for (int i = 1, j = 0; i < vals.length;) {
            TreeNode parent = nodes[j++];
            if (parent != null) {
                if (!vals[i].equals("null"))
parent.left = nodes[i] = new
TreeNode(Integer.parseInt(vals[i]));
                i++;
                if (i < vals.length &&
!vals[i].equals("null")) parent.right = nodes[i] =
new TreeNode(Integer.parseInt(vals[i]));
                i++;
            }
        }
        return root;
    }

    static boolean isLeaf(TreeNode node) {
```

```
        return node != null && node.left == null
&& node.right == null;
    }

    static int leftLeafSum(TreeNode root) {
        if (root == null) return 0;
        int sum = 0;
        if (isLeaf(root.left)) sum +=
root.left.val;
        return sum + leftLeafSum(root.left) +
leftLeafSum(root.right);
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        sc.nextLine(); // read n but ignore since
not used
        String[] vals = sc.nextLine().split(" ");

System.out.println(leftLeafSum(buildTree(vals)));
    }
}
```