# Binary Tree

```
public class BinaryTreeNode<T> {
    public T data;
    public BinaryTreeNode<T> left;
    public BinaryTreeNode<T> right;

    public BinaryTreeNode(T data) {
        this.data = data;
    }
}

// 1st method
public void print(BinaryTreeNode<Integer> root) {
     if (root == null)
          return;
     System.out.print(root.data + ": ");
     if (root.left != null)
          System.out.print("Left= " + root.left.data + ", ");
     else
          System.out.print("Null, ");
     if (root.right != null)
          System.out.print("Right= " + root.right.data + " ");
     else
          System.out.print("Null ");
     System.out.println();
     print(root.left);
     print(root.right);
}

// 2nd method
public BinaryTreeNode<Integer> takeInput() {
     System.out.println("Enter the root Data: ");
     Scanner scanner = new Scanner(System.in);
     int rootData = scanner.nextInt();
     if (rootData == -1)
          return null;
     BinaryTreeNode<Integer> root = new BinaryTreeNode<>(rootData);
     BinaryTreeNode<Integer> leftChild = takeInput();
     BinaryTreeNode<Integer> rightChild = takeInput();
     root.left = leftChild;
     root.right = rightChild;
     return root;
}

// 3rd method
public BinaryTreeNode<Integer> takeInputBetter(
     boolean isRoot, int parentData, boolean isLeft) {
     if (isRoot)
          System.out.print("Enter the root Data: ");
     else {
          if (isLeft)
               System.out.print("Enter the left child of " +
                                parentData + ": ");
```

```
          else
               System.out.print("Enter the right child of " +
               parentData + ": ");
          }
     Scanner scanner = new Scanner(System.in);
     int rootData = scanner.nextInt();
     if (rootData == -1)
          return null;
     BinaryTreeNode<Integer> root = new BinaryTreeNode<>(rootData);
     BinaryTreeNode<Integer> leftChild = takeInputBetter(false,
     rootData, true);
     BinaryTreeNode<Integer> rightChild = takeInputBetter(false,
          rootData, false);
     root.left = leftChild;
     root.right = rightChild;
     return root;
     }

//4th method
public int numberOfNodes(BinaryTreeNode<Integer> root) {
     if (root == null)
          return 0;
     int leftNodeCount = numberOfNodes(root.left);
     int rightNodeCount = numberOfNodes(root.right);
     return 1 + leftNodeCount + rightNodeCount;
}

// 5th method
public BinaryTreeNode<Integer> takeInputLevelWise() {
     Scanner s = new Scanner(System.in);
     System.out.print("Enter root Data: ");
     int rootData = s.nextInt();
     if (rootData == -1)
          return null;
     BinaryTreeNode<Integer> root = new BinaryTreeNode<>(rootData);
     Queue<BinaryTreeNode<Integer>> pendingChild = new LinkedList<>();
     pendingChild.add(root);
     while (!pendingChild.isEmpty()) {
          BinaryTreeNode<Integer> front = pendingChild.poll();
          System.out.print("Enter the left child of " +
                                   front.data + " : ");
          int left = s.nextInt();
          if (left != -1) {
               BinaryTreeNode<Integer> leftChild =
                                   new BinaryTreeNode<>(left);
               front.left = leftChild;
               pendingChild.add(leftChild);
          }
          System.out.print("Enter the right child of " +
                                   front.data + " : ");
          int right = s.nextInt();
          if (right != -1) {
               BinaryTreeNode<Integer> rightChild =
                         new BinaryTreeNode<>(right);
               front.right = rightChild;
```

```
                pendingChild.add(rightChild);
            }
        }
        return root;
    }

    // 6th method
    public void printLevelWise(BinaryTreeNode<Integer> root) {
        Queue<BinaryTreeNode<Integer>> pendingChild = new LinkedList<>();
        pendingChild.add(root);
        while (!pendingChild.isEmpty()) {
            BinaryTreeNode<Integer> front = pendingChild.poll();
            if (front != null) {
                System.out.print(front.data + ":");
                BinaryTreeNode<Integer> left = front.left;
                pendingChild.add(left);
                if (left != null) {
                    System.out.print("L:" + left.data + ",");
                }
                BinaryTreeNode<Integer> right = front.right;
                pendingChild.add(right);
                if (right != null) {
                    System.out.print("R:" + right.data);
                }
                System.out.println();
            }
        }
    }

    // 7th method
    public int largest(BinaryTreeNode<Integer> root) {
        if (root == null)
            return -1;
        int largestLeft = largest(root.left);
        int largestRight = largest(root.right);
        return Math.max(root.data, Math.max(largestLeft, largestRight));
    }

    // 8th method
    public int height(BinaryTreeNode<Integer> root) {
        if (root == null)
            return 0;
        int leftNodeCount = height(root.left);
        int rightNodeCount = height(root.right);
        int longest = Math.max(leftNodeCount, rightNodeCount);
        return 1 + longest;
    }

    // 9th method
    public int numberOfLeaves(BinaryTreeNode<Integer> root) {
        if (root == null)
            return 0;
        if (root.left == null && root.right == null)
            return 1;
        return numberOfLeaves(root.left) + numberOfLeaves(root.right);
```

```
}

// 10th method
public void printAtDepthK(BinaryTreeNode<Integer> root, int k) {
     if (root == null)
          return;
     if (k == 0) {
          System.out.print(root.data + " ");
          return;
     }
     printAtDepthK(root.left, k - 1);
     printAtDepthK(root.right, k - 1);
//   System.out.println();
}

// 11th method
public BinaryTreeNode<Integer> removeLeaves(BinaryTreeNode<Integer> root) {
     if (root == null)
          return null;
     if (root.left == null & root.right == null)
          return null;
     root.left = removeLeaves(root.left);
     root.right = removeLeaves(root.right);
     return root;
}

// 12th method
public boolean isBalanced(BinaryTreeNode<Integer> root) {
     if (root == null)
          return true;
     int leftHeight = height(root.left);
     int rightHeight = height(root.right);
     if (Math.abs(leftHeight - rightHeight) > 1)
          return false;
     boolean isLeftBalanced = isBalanced(root.left);
     boolean isRightBalanced = isBalanced(root.right);
     return isLeftBalanced && isRightBalanced;
}

public class BalancedTreeReturn {
     int height;
     boolean isBalanced;
}

// 13th method
public BalancedTreeReturn isBalancedBetter(BinaryTreeNode<Integer> root) {
     if (root == null) {
          int height = 0;
          boolean isBalance = true;
          BalancedTreeReturn ans = new BalancedTreeReturn();
          ans.height = height;
          ans.isBalanced = isBalance;
          return ans;
     }
     BalancedTreeReturn leftSide = isBalancedBetter(root.left);
```

```
     BalancedTreeReturn rightSide = isBalancedBetter(root.right);
     boolean isBalance = true;
     int height = 1 + Math.max(leftSide.height, rightSide.height);
     if (Math.abs(leftSide.height - rightSide.height) > 1)
          isBalance = false;
     if (!leftSide.isBalanced || !rightSide.isBalanced)
          isBalance = false;
     BalancedTreeReturn ans = new BalancedTreeReturn();
     ans.height = height;
     ans.isBalanced = isBalance;
     return ans;
}

// 14th method
public int diameter(BinaryTreeNode<Integer> root) {
     if (root == null)
          return 0;
     int diameterThroughNode = height(root.left) + height(root.right);
     int diameterInLeft = diameter(root.left);
     int diameterInRight = diameter(root.right);
     return Math.max(diameterThroughNode, Math.max(diameterInLeft,
                    diameterInRight));
}

// 15th method
public void mirrorBinaryTree(BinaryTreeNode<Integer> root) {
     if (root == null)
          return;
     System.out.print(root.data + " ");
     mirrorBinaryTree(root.right);
     mirrorBinaryTree(root.left);
     System.out.println();
}

// 16th method
public  static  BinaryTreeNode<Integer>  buildTreeFromPreIn(int[]  preorder,
int[] inorder) {
    BinaryTreeNode<Integer> root = buildTreeFromPreInHelper(preorder,
                          inorder, 0, preorder.length, 0,inorder.length);
    return root;
}
public static BinaryTreeNode<Integer> buildTreeFromPreInHelper(
       int[] preorder, int[] inorder, int startIndexOfPreorder, int
     endIndexOfPreorder, int startIndexOfInorder, int endIndexOfInorder) {
    if (startIndexOfPreorder > endIndexOfPreorder)
       return null;
    int rootData = preorder[startIndexOfPreorder];
    BinaryTreeNode<Integer> root = new BinaryTreeNode<>(rootData);
    // finding the root index to get start and end in inOrder and preOrder
    int rootIndex = -1;
    for (int i = startIndexOfInorder; i <= endIndexOfInorder; i++) {
       if (inorder[i] == rootData) {
          rootIndex = i;
          break;
       }
```

```
    }
   int startIndexOfPreorderLeft = startIndexOfPreorder + 1;
   int startIndexOfInorderLeft = startIndexOfInorder;
   int endIndexOfInorderLeft = rootIndex - 1;
   int startIndexOfInorderRight = rootIndex + 1;
   int endIndexOfPreorderRight = endIndexOfPreorder;
   int endIndexOfInorderRight = endIndexOfInorder;
// finding length of left subtree
    int leftSubTreeLength= endIndexOfInorderLeft-startIndexOfInorderLeft+1;
   int endIndexOfPreorderLeft = startIndexOfPreorderLeft+leftSubTreeLength-1;
   int startIndexOfPreorderRight = endIndexOfPreorderLeft + 1;

    BinaryTreeNode<Integer> left = buildTreeFromPreInHelper(
                                          preorder,
                                          inorder,
                                          startIndexOfPreorderLeft,
                                          endIndexOfPreorderLeft,
                                          startIndexOfInorderLeft,
                                          endIndexOfInorderLeft);

    BinaryTreeNode<Integer> right = buildTreeFromPreInHelper(
                                          preorder,
                                          inorder,
                                          startIndexOfPreorderRight,
                                          endIndexOfPreorderRight,
                                          startIndexOfInorderRight,
                                          endIndexOfInorderRight);
    root.left = left;
    root.right = right;
    return root;
}
```