```java
package hw06;

import java.io.*;

/**
 * Class for a binary tree that stores type E objects.
 *
 * @author Koffman and Wolfgang
 * @param <E>
 *
 */
public class BinaryTree< E> implements Serializable {

    /**
     * Class to encapsulate a tree node.
     * @param <E>
     */
    protected static class Node< E> implements Serializable {
        // Data Fields
        /**
         * The information stored in this node.
         */
        protected E data;

        /**
         * Reference to the left child.
         */
        protected Node< E> left;

        /**
         * Reference to the right child.
         */
        protected Node< E> right;

        /**
         * Reference to the parent.
         */
        protected Node<E> parent;

        // Constructors
        /**
         * Construct a node with given data and no children.
         *
         * @param data The data to store in this node
         */
        public Node(E data) {
            this.data = data;
            left = null;
            right = null;
            parent = null;
        }

        // Methods
        /**
         * Return a string representation of the node.
         *
         * @return A string representation of the data fields
         */
        @Override
        public String toString() {
            return data.toString();
        }
    }

    // Data Field
    /**
     * The root of the binary tree
     */
    protected Node< E> root;

    public BinaryTree() {
        root = null;
    }

    protected BinaryTree(Node< E> root) {
        this.root = root;
    }

    /**
     * Constructs a new binary tree with data in its root,leftTree as its left
     * subtree and rightTree as its right subtree.
     * @param data
     * @param leftTree
     * @param rightTree
     */
    public BinaryTree(E data, BinaryTree< E> leftTree,
            BinaryTree< E> rightTree) {
        root = new Node<>(data);
        if (leftTree != null) {
            root.left = leftTree.root;
        } else {
            root.left = null;
        }
        if (rightTree != null) {
            root.right = rightTree.root;
        } else {
            root.right = null;
        }
    }

    /**
     * Return the left subtree.
     *
     * @return The left subtree or null if either the root or the left subtree
     * is null
     */
    public BinaryTree< E> getLeftSubtree() {
        if (root != null && root.left != null) {
            return new BinaryTree<>(root.left);
        } else {
            return null;
        }
    }

    /**
     * Return the right sub-tree
     *
     * @return the right sub-tree or null if either the root or the right
     * subtree is null.
     */
    public BinaryTree<E> getRightSubtree() {
        if (root != null && root.right != null) {
            return new BinaryTree<>(root.right);
        } else {
            return null;
        }
    }

    public E getData(){
        if(root != null){
            return root.data;
        }
        else{
            return null;
        }
    }
    /**
     * Determine whether this tree is a leaf.
     *
     * @return true if the root has no children
     */
    public boolean isLeaf() {
        return (root.left == null && root.right == null);
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        preOrderTraverse(root, 1, sb);
        return sb.toString();
    }

    /**
     * Perform a preorder traversal.
     *
     * @param node The local root
     * @param depth The depth
     * @param sb The string buffer to save the output
     */
    private void preOrderTraverse(Node< E> node, int depth,
            StringBuilder sb) {
        for (int i = 1; i < depth; i++) {
            sb.append("  ");
        }
        if (node == null) {
            sb.append("null\n");
        } else {
            sb.append(node.toString());
            sb.append("\n");
            preOrderTraverse(node.left, depth + 1, sb);
            preOrderTraverse(node.right, depth + 1, sb);
        }
    }

    /**
     * Method to read a binary tree. pre: The input consists of a preorder
     * traversal of the binary tree. The line "null" indicates a null tree.
     *
     * @param bR The input file
     * @return The binary tree
     * @throws IOException If there is an input error
     */
    public static BinaryTree< String>
            readBinaryTree(BufferedReader bR) throws IOException {
        // Read a line and trim leading and trailing spaces.
        String data = bR.readLine().trim();
        if (data.equals("null")) {
            return null;
        } else {
            BinaryTree< String> leftTree = readBinaryTree(bR);
            BinaryTree< String> rightTree = readBinaryTree(bR);
            return new BinaryTree<>(data, leftTree, rightTree);
        }
    }

}
```