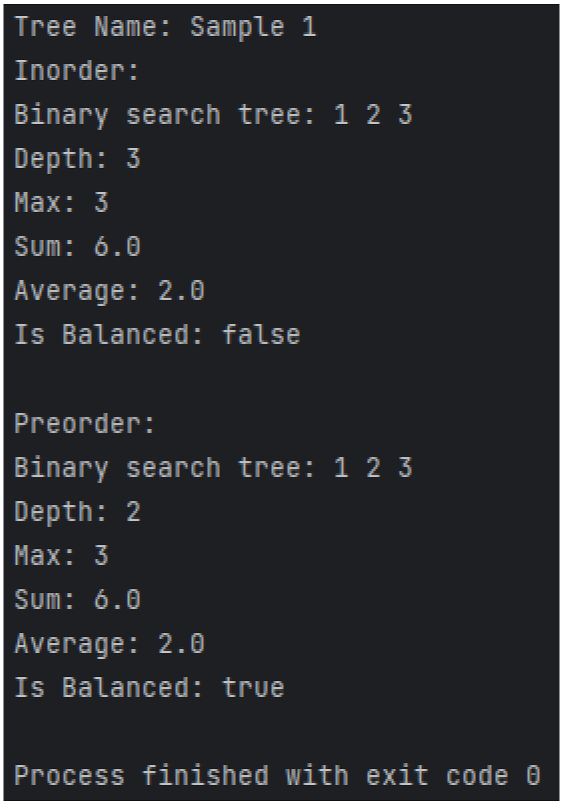
Assignment 8 - BTS

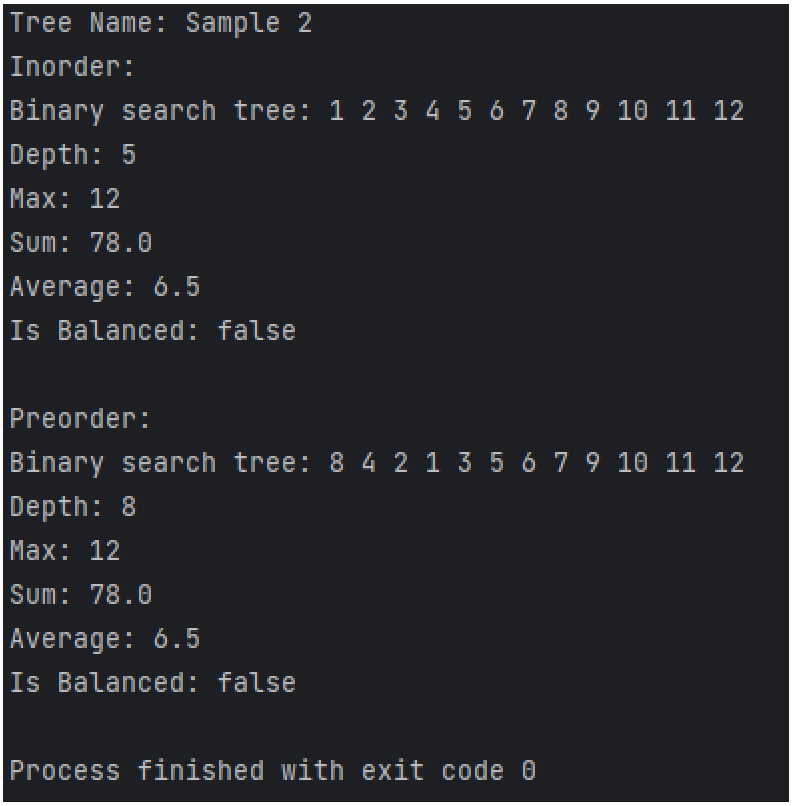
Ryan Brinson

11/2/20203

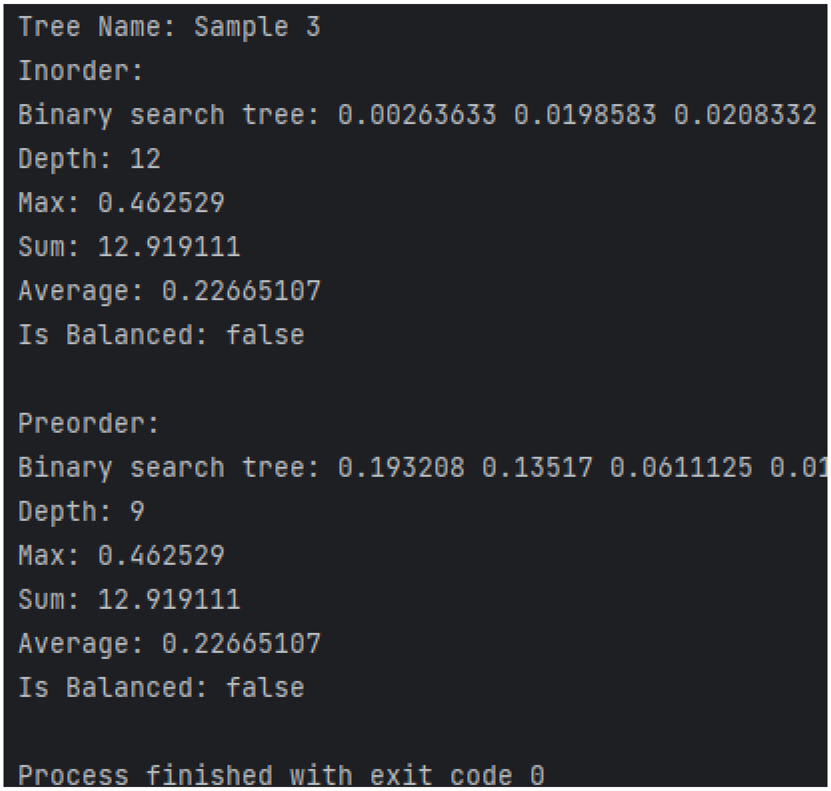
Output:



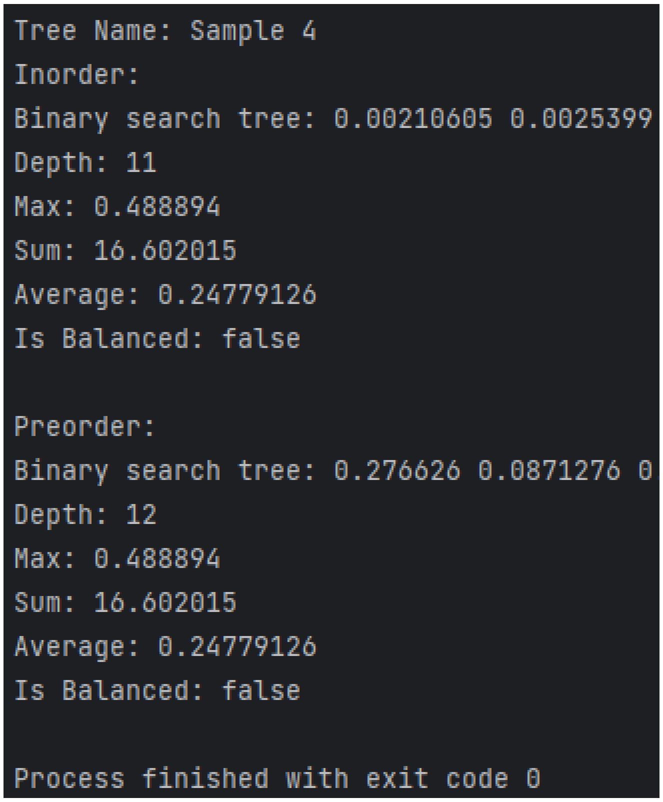
```
Tree Name: Sample 1
Inorder:
Binary search tree: 1 2 3
Depth: 3
Max: 3
Sum: 6.0
Average: 2.0
Is Balanced: false

Preorder:
Binary search tree: 1 2 3
Depth: 2
Max: 3
Sum: 6.0
Average: 2.0
Is Balanced: true

Process finished with exit code 0
```



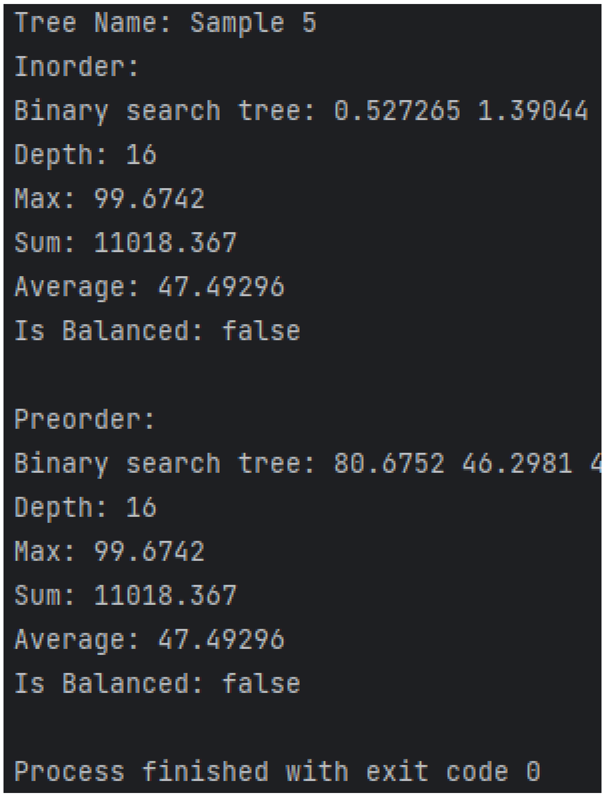
```
Tree Name: Sample 2
Inorder:
Binary search tree: 1 2 3 4 5 6 7 8 9 10 11 12
Depth: 5
Max: 12
Sum: 78.0
Average: 6.5
Is Balanced: false

Preorder:
Binary search tree: 8 4 2 1 3 5 6 7 9 10 11 12
Depth: 8
Max: 12
Sum: 78.0
Average: 6.5
Is Balanced: false

Process finished with exit code 0
```



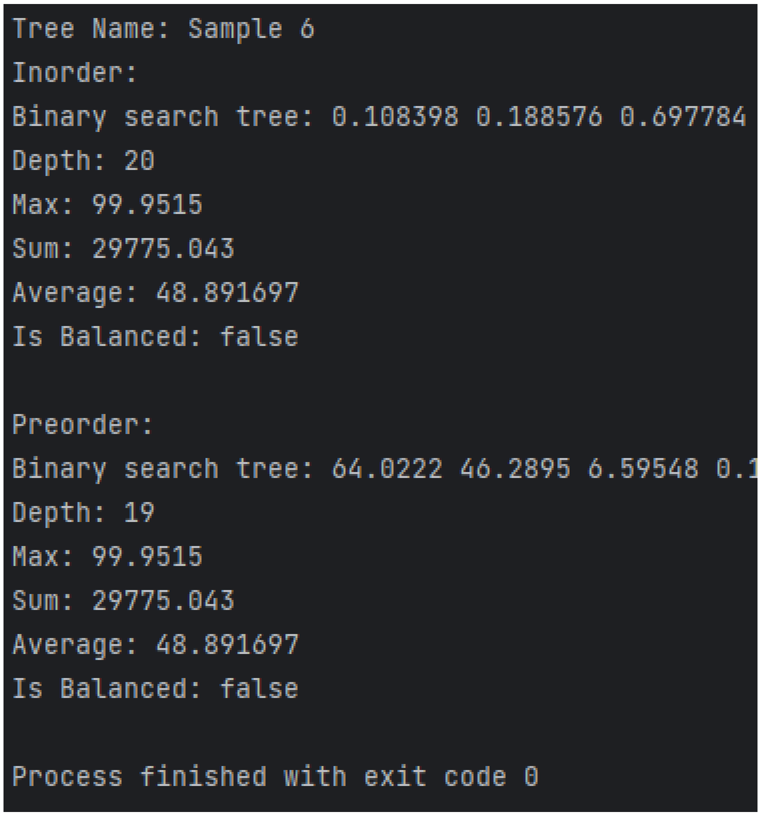
```
Tree Name: Sample 3
Inorder:
Binary search tree: 0.00263633 0.0198583 0.0208332
Depth: 12
Max: 0.462529
Sum: 12.919111
Average: 0.22665107
Is Balanced: false

Preorder:
Binary search tree: 0.193208 0.13517 0.0611125 0.01
Depth: 9
Max: 0.462529
Sum: 12.919111
Average: 0.22665107
Is Balanced: false

Process finished with exit code 0
```



```
Tree Name: Sample 4
Inorder:
Binary search tree: 0.00210605 0.0025399
Depth: 11
Max: 0.488894
Sum: 16.602015
Average: 0.24779126
Is Balanced: false

Preorder:
Binary search tree: 0.276626 0.0871276 0
Depth: 12
Max: 0.488894
Sum: 16.602015
Average: 0.24779126
Is Balanced: false

Process finished with exit code 0
```

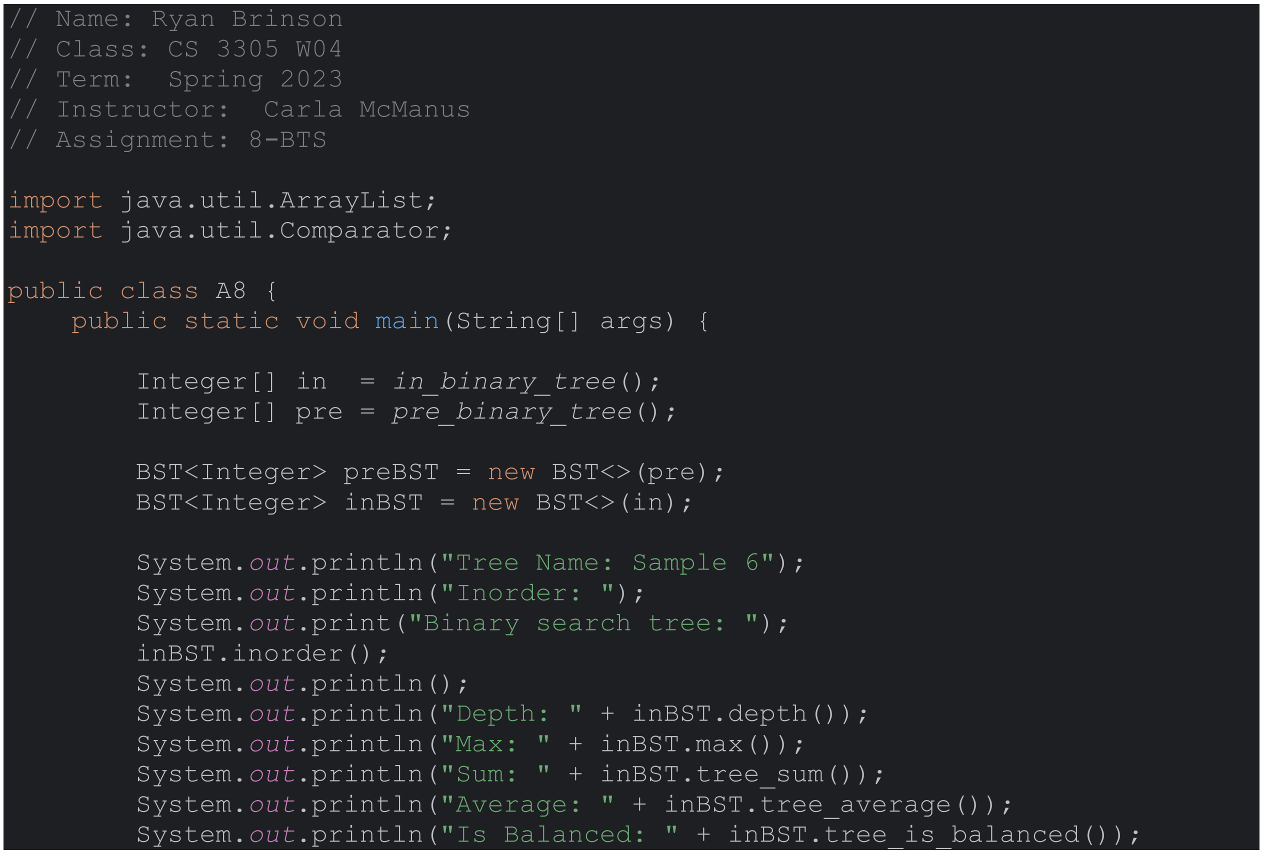


```
Tree Name: Sample 5
Inorder:
Binary search tree: 0.527265 1.39044
Depth: 16
Max: 99.6742
Sum: 11018.367
Average: 47.49296
Is Balanced: false

Preorder:
Binary search tree: 80.6752 46.2981 4
Depth: 16
Max: 99.6742
Sum: 11018.367
Average: 47.49296
Is Balanced: false

Process finished with exit code 0
```

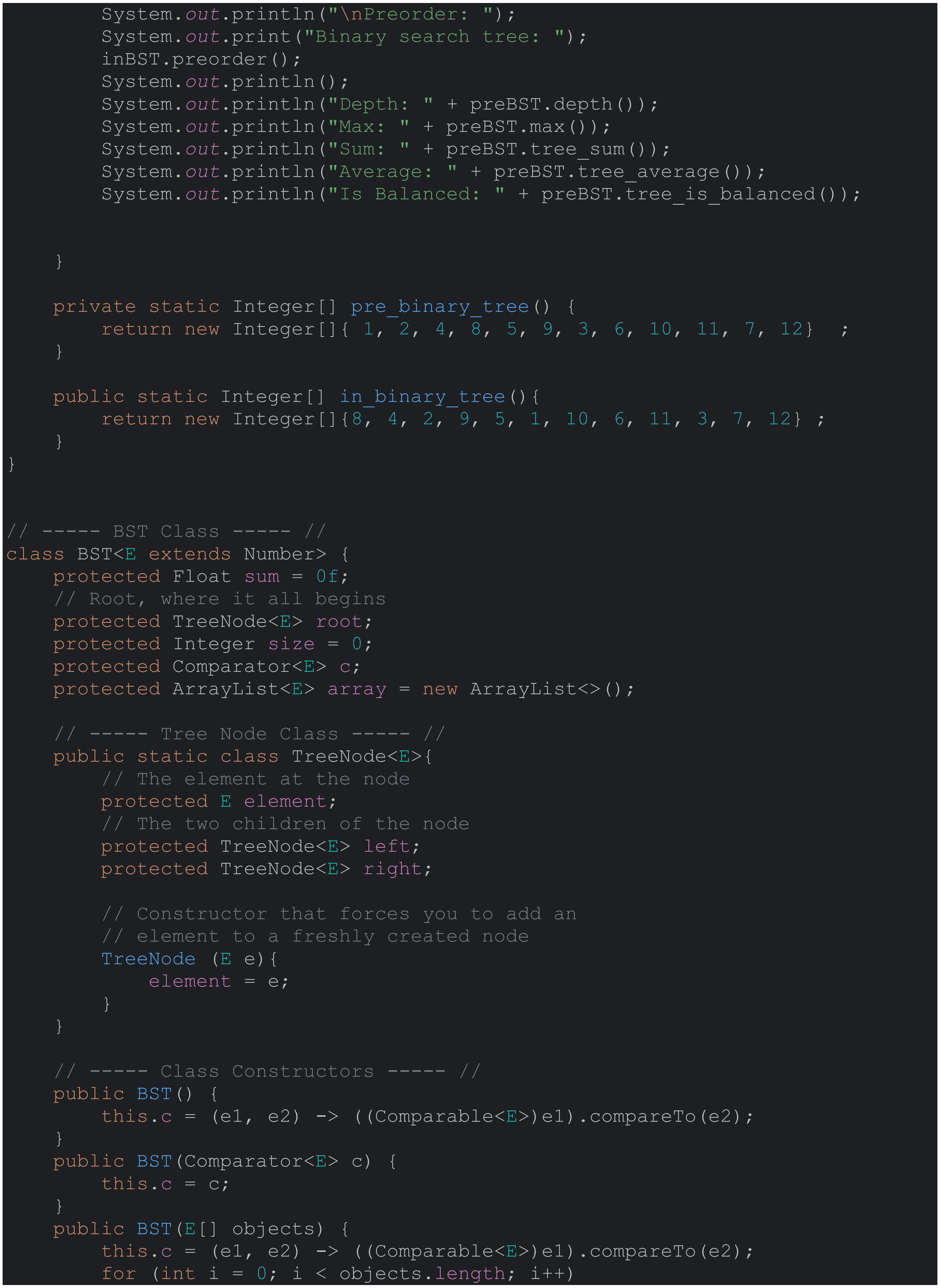


```
Tree Name: Sample 6
Inorder:
Binary search tree: 0.108398 0.188576 0.697784
Depth: 20
Max: 99.9515
Sum: 29775.043
Average: 48.891697
Is Balanced: false

Preorder:
Binary search tree: 64.0222 46.2895 6.59548 0.1
Depth: 19
Max: 99.9515
Sum: 29775.043
Average: 48.891697
Is Balanced: false

Process finished with exit code 0
```

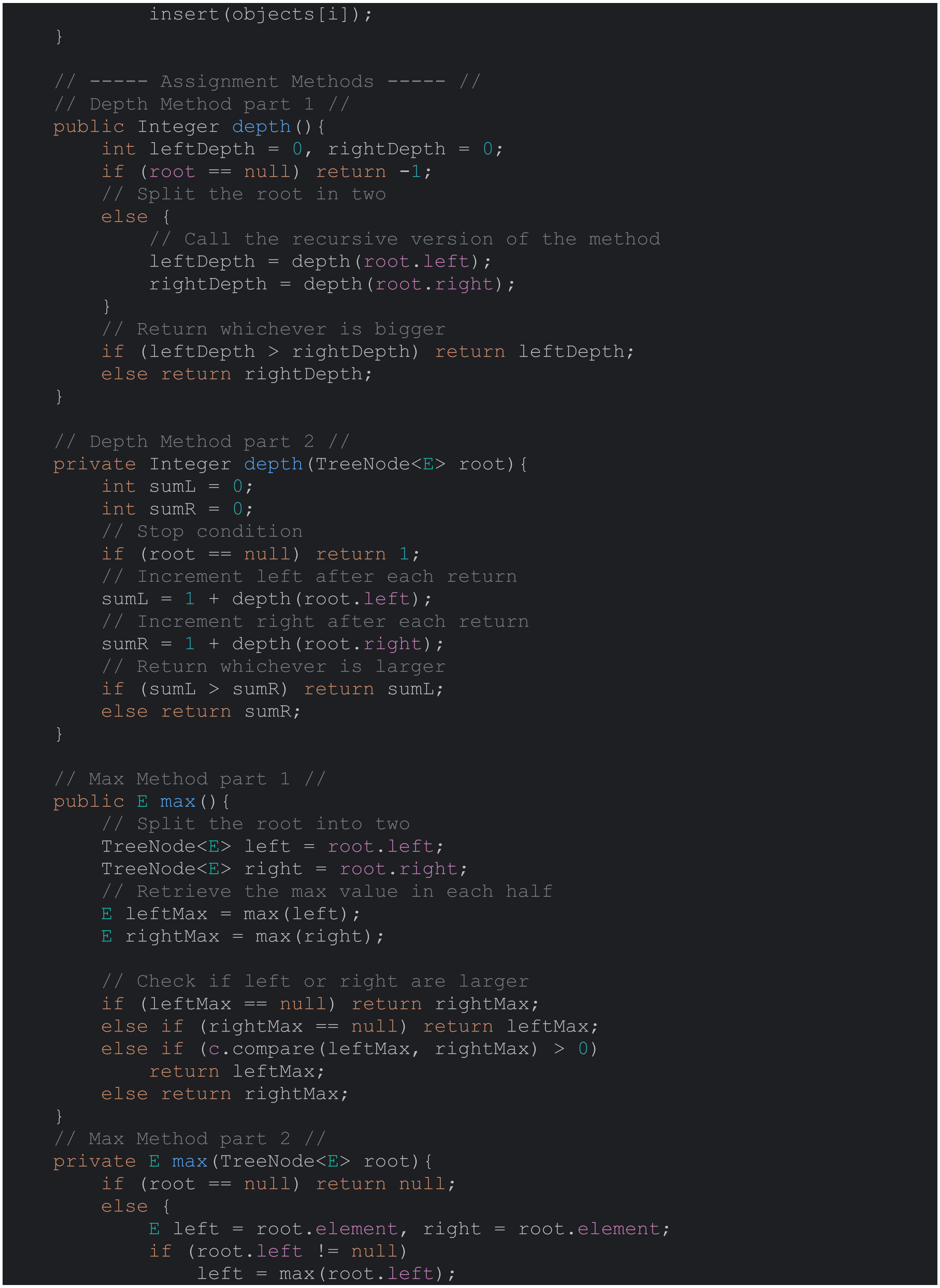
Code:



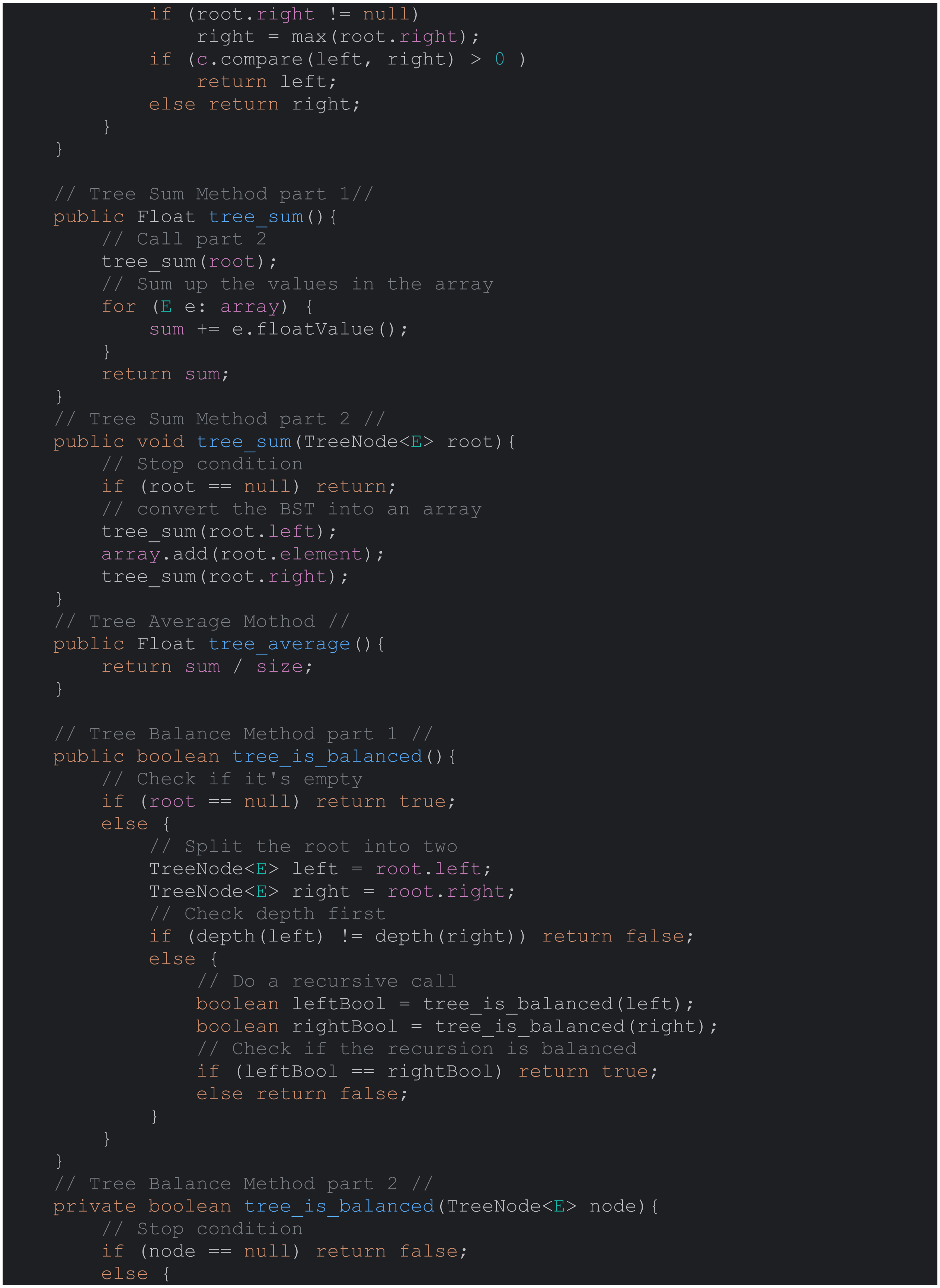
```java
// Name: Ryan Brinson
// Class: CS 3305 W04
// Term:  Spring 2023
// Instructor:  Carla McManus
// Assignment: 8-BTS

import java.util.ArrayList;
import java.util.Comparator;

public class A8 {
    public static void main(String[] args) {

        Integer[] in  = in_binary_tree();
        Integer[] pre = pre_binary_tree();

        BST<Integer> preBST = new BST<>(pre);
        BST<Integer> inBST = new BST<>(in);

        System.out.println("Tree Name: Sample 6");
        System.out.println("Inorder: ");
        System.out.print("Binary search tree: ");
        inBST.inorder();
        System.out.println();
        System.out.println("Depth: " + inBST.depth());
        System.out.println("Max: " + inBST.max());
        System.out.println("Sum: " + inBST.tree_sum());
        System.out.println("Average: " + inBST.tree_average());
        System.out.println("Is Balanced: " + inBST.tree_is_balanced());
```



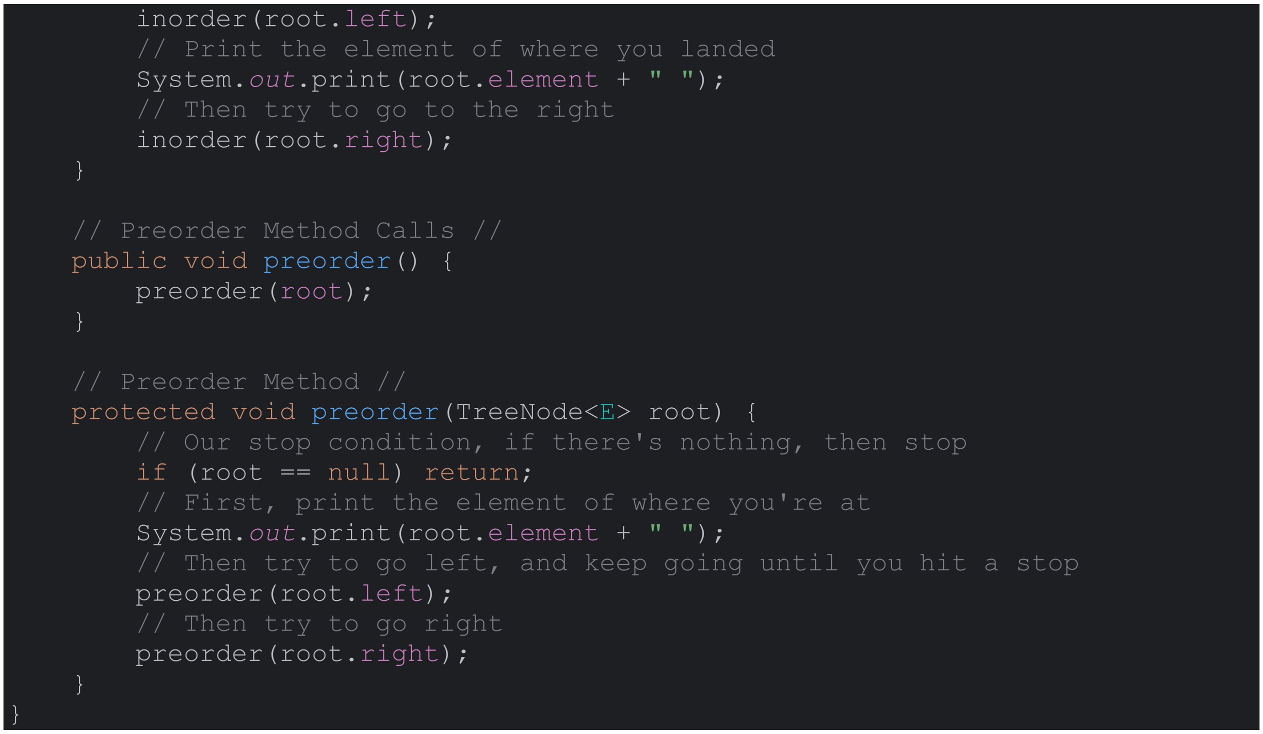
```
        System.out.println("\nPreorder: ");
        System.out.print("Binary search tree: ");
        inBST.preorder();
        System.out.println();
        System.out.println("Depth: " + preBST.depth());
        System.out.println("Max: " + preBST.max());
        System.out.println("Sum: " + preBST.tree_sum());
        System.out.println("Average: " + preBST.tree_average());
        System.out.println("Is Balanced: " + preBST.tree_is_balanced());


    }

    private static Integer[] pre_binary_tree() {
        return new Integer[]{ 1, 2, 4, 8, 5, 9, 3, 6, 10, 11, 7, 12}  ;
    }

    public static Integer[] in_binary_tree(){
        return new Integer[]{8, 4, 2, 9, 5, 1, 10, 6, 11, 3, 7, 12} ;
    }
}


// ----- BST Class ----- //
class BST<E extends Number> {
    protected Float sum = 0f;
    // Root, where it all begins
    protected TreeNode<E> root;
    protected Integer size = 0;
    protected Comparator<E> c;
    protected ArrayList<E> array = new ArrayList<>();

    // ----- Tree Node Class ----- //
    public static class TreeNode<E>{
        // The element at the node
        protected E element;
        // The two children of the node
        protected TreeNode<E> left;
        protected TreeNode<E> right;

        // Constructor that forces you to add an
        // element to a freshly created node
        TreeNode (E e){
            element = e;
        }
    }

    // ----- Class Constructors ----- //
    public BST() {
        this.c = (e1, e2) -> ((Comparable<E>)e1).compareTo(e2);
    }
    public BST(Comparator<E> c) {
        this.c = c;
    }
    public BST(E[] objects) {
        this.c = (e1, e2) -> ((Comparable<E>)e1).compareTo(e2);
        for (int i = 0; i < objects.length; i++)
```



```
            insert(objects[i]);
    }

    // ----- Assignment Methods ----- //
    // Depth Method part 1 //
    public Integer depth(){
        int leftDepth = 0, rightDepth = 0;
        if (root == null) return -1;
        // Split the root in two
        else {
            // Call the recursive version of the method
            leftDepth = depth(root.left);
            rightDepth = depth(root.right);
        }
        // Return whichever is bigger
        if (leftDepth > rightDepth) return leftDepth;
        else return rightDepth;
    }

    // Depth Method part 2 //
    private Integer depth(TreeNode<E> root){
        int sumL = 0;
        int sumR = 0;
        // Stop condition
        if (root == null) return 1;
        // Increment left after each return
        sumL = 1 + depth(root.left);
        // Increment right after each return
        sumR = 1 + depth(root.right);
        // Return whichever is larger
        if (sumL > sumR) return sumL;
        else return sumR;
    }

    // Max Method part 1 //
    public E max(){
        // Split the root into two
        TreeNode<E> left = root.left;
        TreeNode<E> right = root.right;
        // Retrieve the max value in each half
        E leftMax = max(left);
        E rightMax = max(right);

        // Check if left or right are larger
        if (leftMax == null) return rightMax;
        else if (rightMax == null) return leftMax;
        else if (c.compare(leftMax, rightMax) > 0)
            return leftMax;
        else return rightMax;
    }
    // Max Method part 2 //
    private E max(TreeNode<E> root){
        if (root == null) return null;
        else {
            E left = root.element, right = root.element;
            if (root.left != null)
                left = max(root.left);
```



```
            if (root.right != null)
                right = max(root.right);
            if (c.compare(left, right) > 0 )
                return left;
            else return right;
        }
    }

    // Tree Sum Method part 1//
    public Float tree_sum(){
        // Call part 2
        tree_sum(root);
        // Sum up the values in the array
        for (E e: array) {
            sum += e.floatValue();
        }
        return sum;
    }
    // Tree Sum Method part 2 //
    public void tree_sum(TreeNode<E> root){
        // Stop condition
        if (root == null) return;
        // convert the BST into an array
        tree_sum(root.left);
        array.add(root.element);
        tree_sum(root.right);
    }
    // Tree Average Mothod //
    public Float tree_average(){
        return sum / size;
    }

    // Tree Balance Method part 1 //
    public boolean tree_is_balanced(){
        // Check if it's empty
        if (root == null) return true;
        else {
            // Split the root into two
            TreeNode<E> left = root.left;
            TreeNode<E> right = root.right;
            // Check depth first
            if (depth(left) != depth(right)) return false;
            else {
                // Do a recursive call
                boolean leftBool = tree_is_balanced(left);
                boolean rightBool = tree_is_balanced(right);
                // Check if the recursion is balanced
                if (leftBool == rightBool) return true;
                else return false;
            }
        }
    }
    // Tree Balance Method part 2 //
    private boolean tree_is_balanced(TreeNode<E> node){
        // Stop condition
        if (node == null) return false;
        else {
            // Other wise
            boolean left = tree_is_balanced(node.left);
            boolean right = tree_is_balanced(node.right);
            if (left == right) return true;
            else return false;
        }
    }

    // ----- Class Methods ----- //
    // Insert new element //
    public boolean insert(E e) {
        // If there is no root, create it
        if (root == null)
            root = new TreeNode<>(e); // Create a new root
        else {
            TreeNode<E> parent = null;
            TreeNode<E> current = root;

            // Let's find a place for e
            while (current != null){
                // If e < current e then we travel left
                if (c.compare(e, current.element) < 0) {
                    parent = current;
                    current = current.left;
                }
                // Else, if e > current e, we travel to the right
                else if (c.compare(e, current.element) > 0) {
                    parent = current;
                    current = current.right;
                }
                // If it's equal a value we've found, then we can disregard
it
                else if (c.compare(e, current.element) == 0)
                    return false; // Failed to insert element
            }

            // Now that we have the right parent for our node
            // If the e < parent e it goes to the left
            if (c.compare(e, parent.element) < 0)
                parent.left = new TreeNode<>(e);
            // Otherwise, if e > parent, it goes to the right
            else
                parent.right = new TreeNode<>(e);
        }
        // The tree just grew
        size++;
        return true; // Element inserted successfully
    }

    // Inorder Method Calls //
    public void inorder() {
        inorder(root);
    }
    protected void inorder(TreeNode<E> root) {
        // Our stop condition, if there's nothing, then stop
        if (root == null) return;
        // First go all the way down the left as far as possible
```



```
        inorder(root.left);
        // Print the element of where you landed
        System.out.print(root.element + " ");
        // Then try to go to the right
        inorder(root.right);
    }

    // Preorder Method Calls //
    public void preorder() {
        preorder(root);
    }

    // Preorder Method //
    protected void preorder(TreeNode<E> root) {
        // Our stop condition, if there's nothing, then stop
        if (root == null) return;
        // First, print the element of where you're at
        System.out.print(root.element + " ");
        // Then try to go left, and keep going until you hit a stop
        preorder(root.left);
        // Then try to go right
        preorder(root.right);
    }
}
```