CS 3303-01 Data Structures

Instructor: Professor Vandelun Amokaha

Name: Anonymos

Assignment Unit 4

**Introduction**

In this assignment, I implemented a program using the Java language to construct and perform operations on a Binary Search Tree (BST). This program inserts a series of integers entered by the user via standard input into the BST, maintaining the tree structure during insertion and allowing for searching operations within the tree. Additionally, the program verifies whether a specified search value exists within the tree. While final checks were performed in the Jeliot environment, due to its slow performance, basic verification was done using an online Java compiler.

**Algorithm Design**

In this program, the `Node` class was first defined to represent a binary search tree node. This class stores the node's data (integer values), left child, and right child. The `BinarySearchTree` class provides the `insert()` method for inserting new nodes and the `search()` method for searching the tree.

The insertion algorithm leverages the basic properties of a binary search tree, where values smaller than the current node are inserted into the left subtree and values larger than the current node are inserted into the right subtree. This recursive insertion method ensures that the tree structure is consistently maintained. Similarly, the search algorithm uses a recursive approach to check whether a specified value exists within the tree.

**Program Functionality Overview**

The program operates as follows:

1. The user inputs a series of integers. The input ends when the user enters `0`.
2. The entered integers are inserted into the binary search tree, preserving the tree's structure.
3. Afterward, the user enters a search value, and the program checks whether the value exists in the tree, displaying the result.
4. Finally, the tree's contents are displayed using an in-order traversal.

**Time Complexity Analysis**

The average time complexity for insertion and searching in a binary search tree is O(log n). However, in the worst case (when all nodes are skewed to one side), the time complexity can become O(n). In this implementation, since the data is inserted in random order based on user input, the search operations are generally performed with O(log n) efficiency.

**Code Implementation**

The implemented code is submitted as `main.java`. When verifying the implementation in an online compiler, the first line, `import Prog1Tools.IOTools;`, should be commented out, as this is a Jeliot-specific library.

**Results and Considerations**

For the binary search tree, the following input was used:

```

10 5 12 3 1 13 7 2 4 14 9 8 6 11 0

```

The binary search tree constructed from this input is as follows:

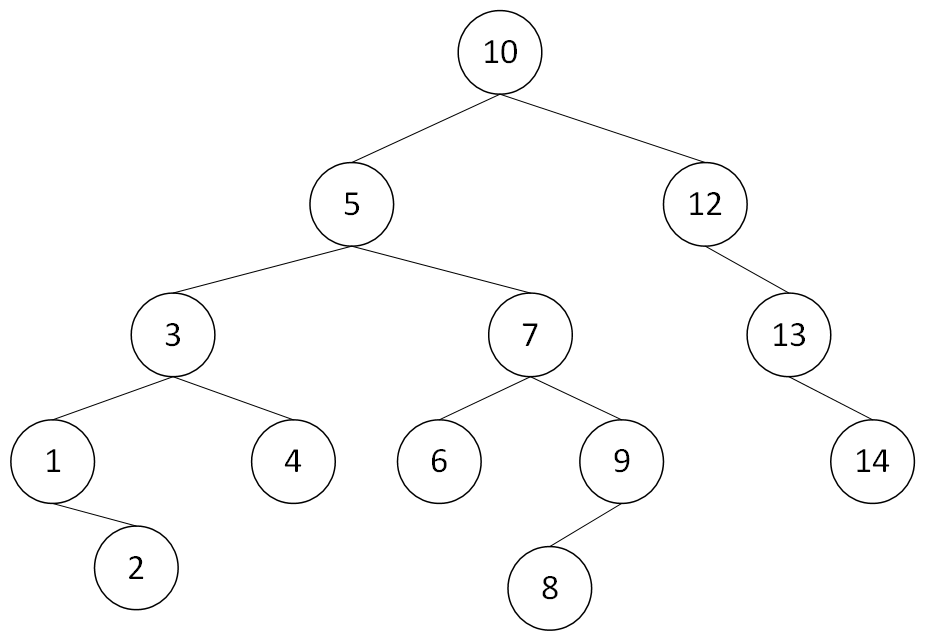


Figure 1: Binary Tree

To organize the search counts for each value in advance, the following search counts were obtained. Search count refers to the number of comparisons made for each node. The counting starts at the root and increments as comparisons are made to determine whether the target node is smaller or larger than the current node.

Search count: 1 -> 10

Search count: 2 -> 5, 12

Search count: 3 -> 3, 7, 11, 14

Search count: 4 -> 1, 4, 6, 9, 14

Search count: 5 -> 2, 8

The results of the execution are shown below.

In the Jeliot environment, the input provided for testing was as follows:

10 5 12 3 1 13 7 2 4 14 9 8 6 11 0

9

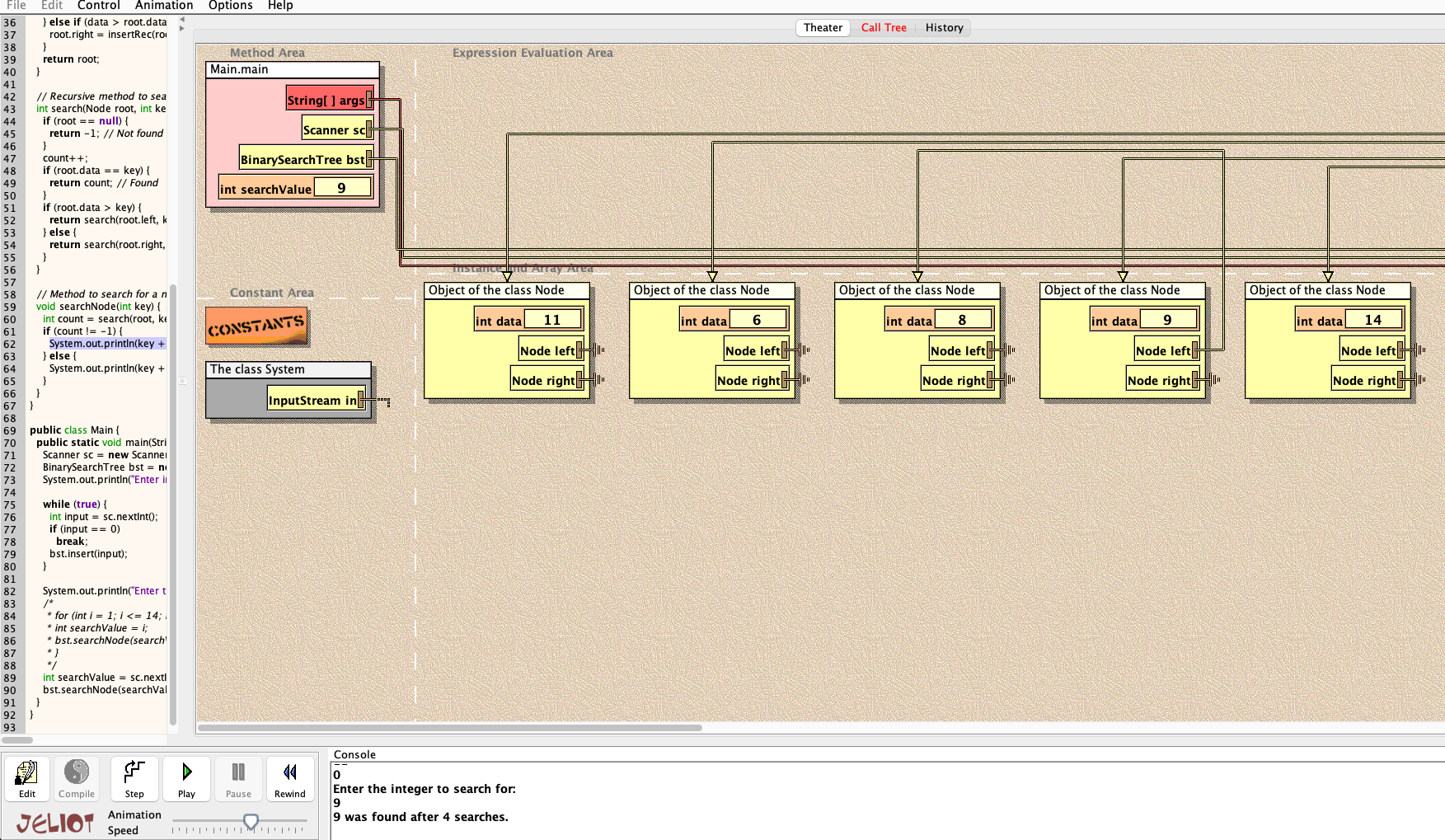
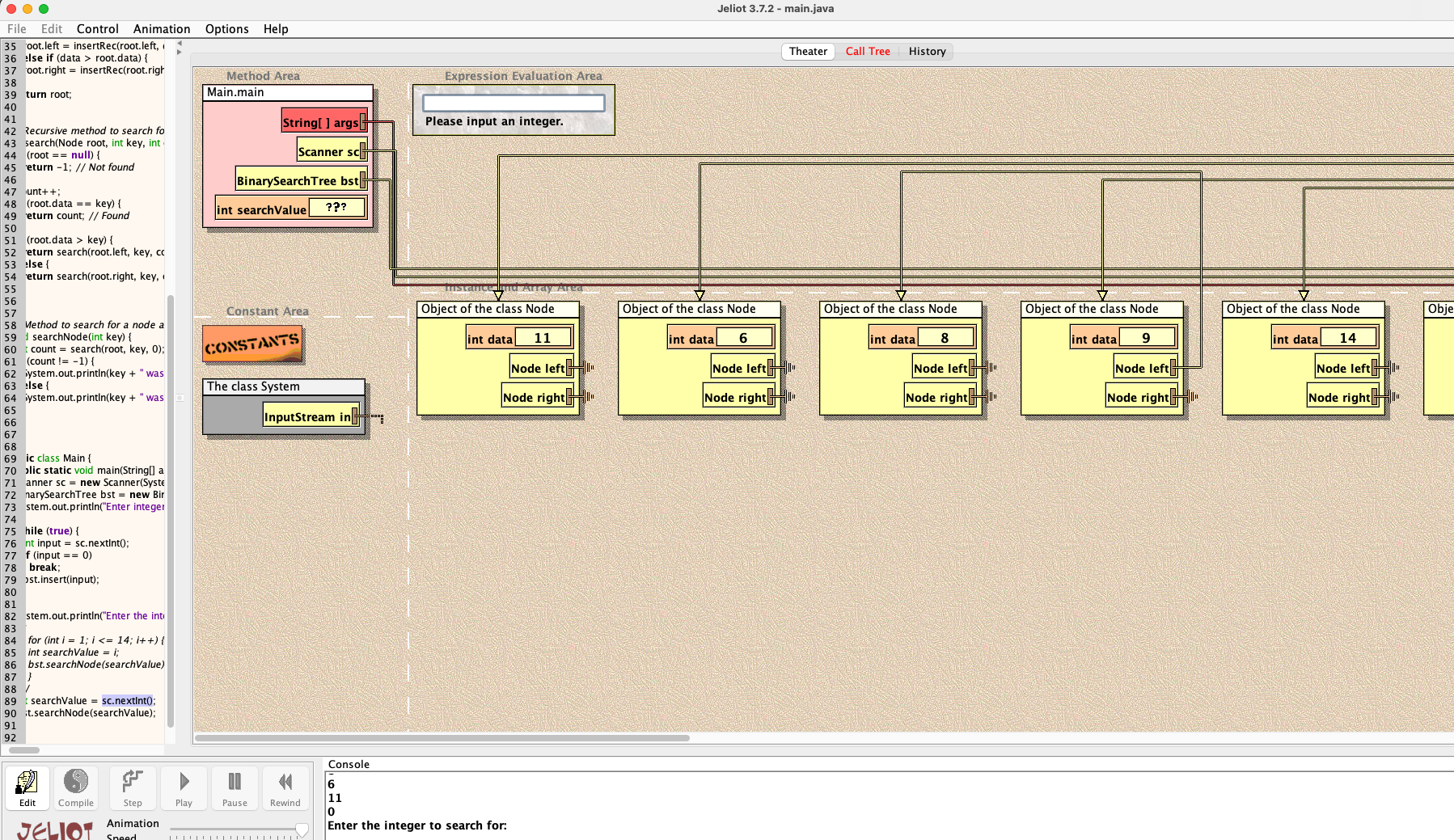


Figure 2: Result from Jeliot execution

Here, the expected search count for the value `9` is `4 search count.`, and the result matches this expectation.

In the online Java compiler, I output the search count for all values entered into the binary search tree.

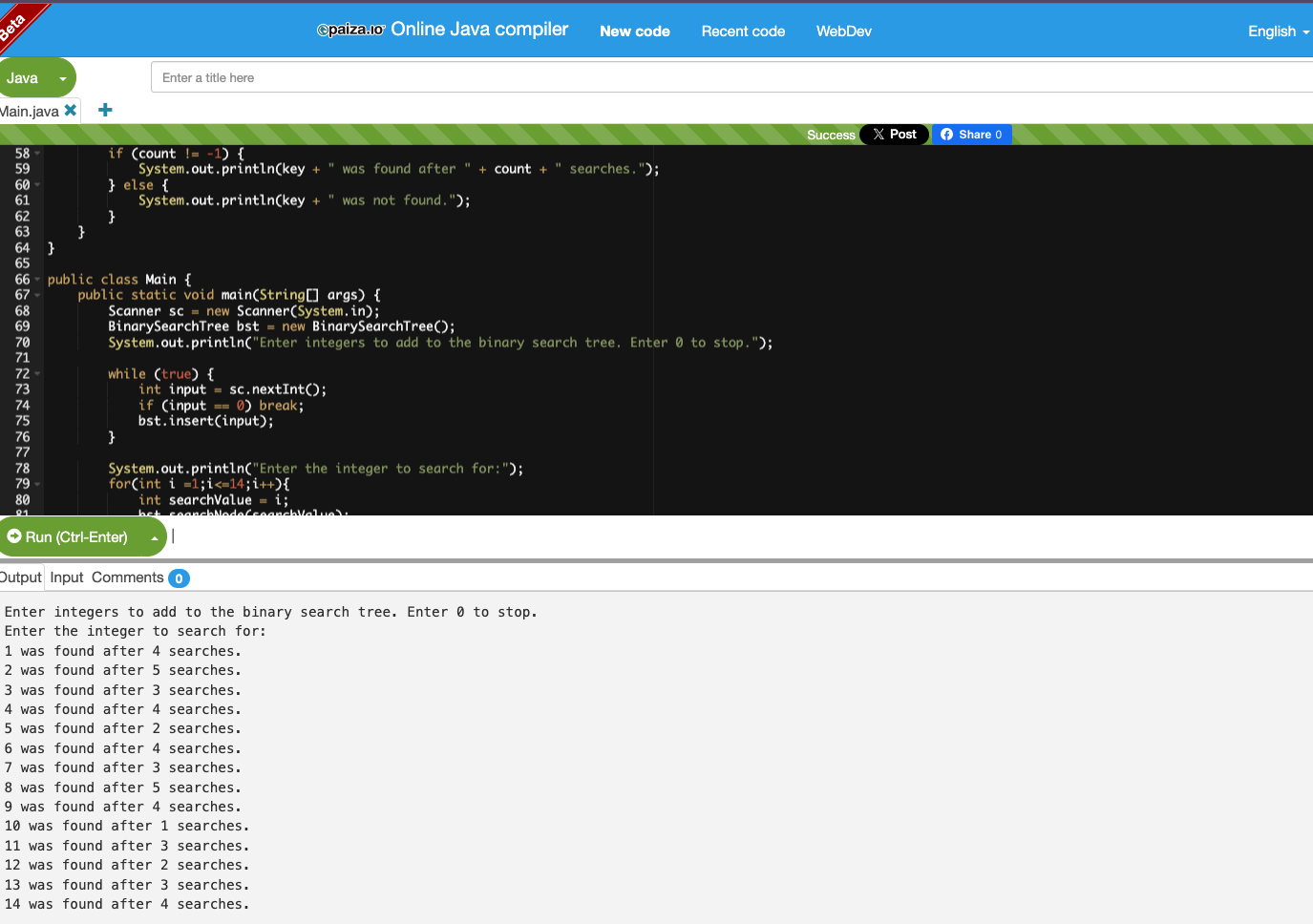


Figure 3: Result from online Java compiler execution

This result matches the pre-organized search count, confirming that the expected number of searches was conducted for each value in the tree.

**Conclusion**

In this assignment, I implemented an insertion and search algorithm that takes advantage of the essential properties of binary search trees. By utilizing both the Jeliot environment and an online Java environment, I was able to construct the tree structure appropriately and verify the algorithm's efficiency. Through this implementation, I gained an understanding of the efficient search capabilities of binary search trees, as well as their advantages and limitations. In the future, I plan to study tree-balancing algorithms (e.g., AVL trees, red-black trees) to design even more efficient data structures.

Word Count: 707

References

1. paiza. (n.d.). *Online Java compiler and coding environment.* paiza.io. Retrieved October 1, 2024, from https://paiza.io/