Binary Search Tree Algorithm Analysis

Algorithm Execution Flow

When running the provided algorithm, the program performs the following operations:

1. Creates a root node with value 5
2. Inserts nodes with values 1, 8, 6, 3, and 9 in that order
3. Performs a traversal of the tree and prints each node's value

Let's trace through the execution of this algorithm:

Tree Construction:

- Start with root node = 5
- Insert 1:
  - 1 < 5, so it goes to the left of 5
- Insert 8:
  - 8 > 5, so it goes to the right of 5
- Insert 6:
  - 6 > 5, so we move to the right child (8)
  - 6 < 8, so it goes to the left of 8
- Insert 3:
  - 3 < 5, so we move to the left child (1)
  - 3 > 1, so it goes to the right of 1
- Insert 9:
  - 9 > 5, so we move to the right child (8)
  - 9 > 8, so it goes to the right of 8

Final Tree Structure:

```
      5
    /   \
  1       8
    \   /   \
      3 6     9
```

Traversal Output:

When the printOrder() method is executed, it produces:

```
Traversed 1
Traversed 3
Traversed 5
Traversed 6
Traversed 8
Traversed 9
```

Analysis of Tree Type and Traversal

Tree Type

The algorithm is implementing a **Binary Search Tree (BST)**, which is characterized by the following properties:

- Each node has at most two children (left and right)
- For any node, all values in the left subtree are less than the node's value
- For any node, all values in the right subtree are greater than the node's value
- No duplicate values are allowed (the algorithm doesn't handle equals case)

This can be verified by examining the insert() method, which places smaller values to the left and larger values to the right.

Traversal Type

The traversal being performed is an **in-order traversal**. In an in-order traversal:

1. Visit the left subtree
2. Visit the root node
3. Visit the right subtree

This traversal method always visits nodes in ascending order in a BST, as demonstrated by the output: 1, 3, 5, 6, 8, 9.

This is confirmed by examining the printOrder() method structure:

```java
printOrder(node.left);              // First visit left
System.out.println(node.value);   // Then visit node
printOrder(node.right);             // Finally visit right
```

Asymptotic Analysis

Insert Operation

- **Best Case:** O(1) - When inserting at the root of an empty tree
- **Average Case:** O(log n) - For a balanced BST, each comparison reduces the search space by half
- **Worst Case:** O(n) - When the tree degenerates into a linked list (e.g., inserting already sorted data)

In-Order Traversal

- **Time Complexity:** O(n) - Every node in the tree must be visited exactly once
- **Space Complexity:** O(h) where h is the height of the tree - Due to the recursive calls on the stack

Overall Algorithm

The entire algorithm consists of:

1. Inserting 5 elements
2. Traversing all elements

For the insertion phase:

- 5 insertions at O(log n) on average: O(5 log n) = O(log n)
- Or 5 insertions at O(n) in worst case: O(5n) = O(n)

For the traversal phase:

- Always O(n) regardless of tree structure

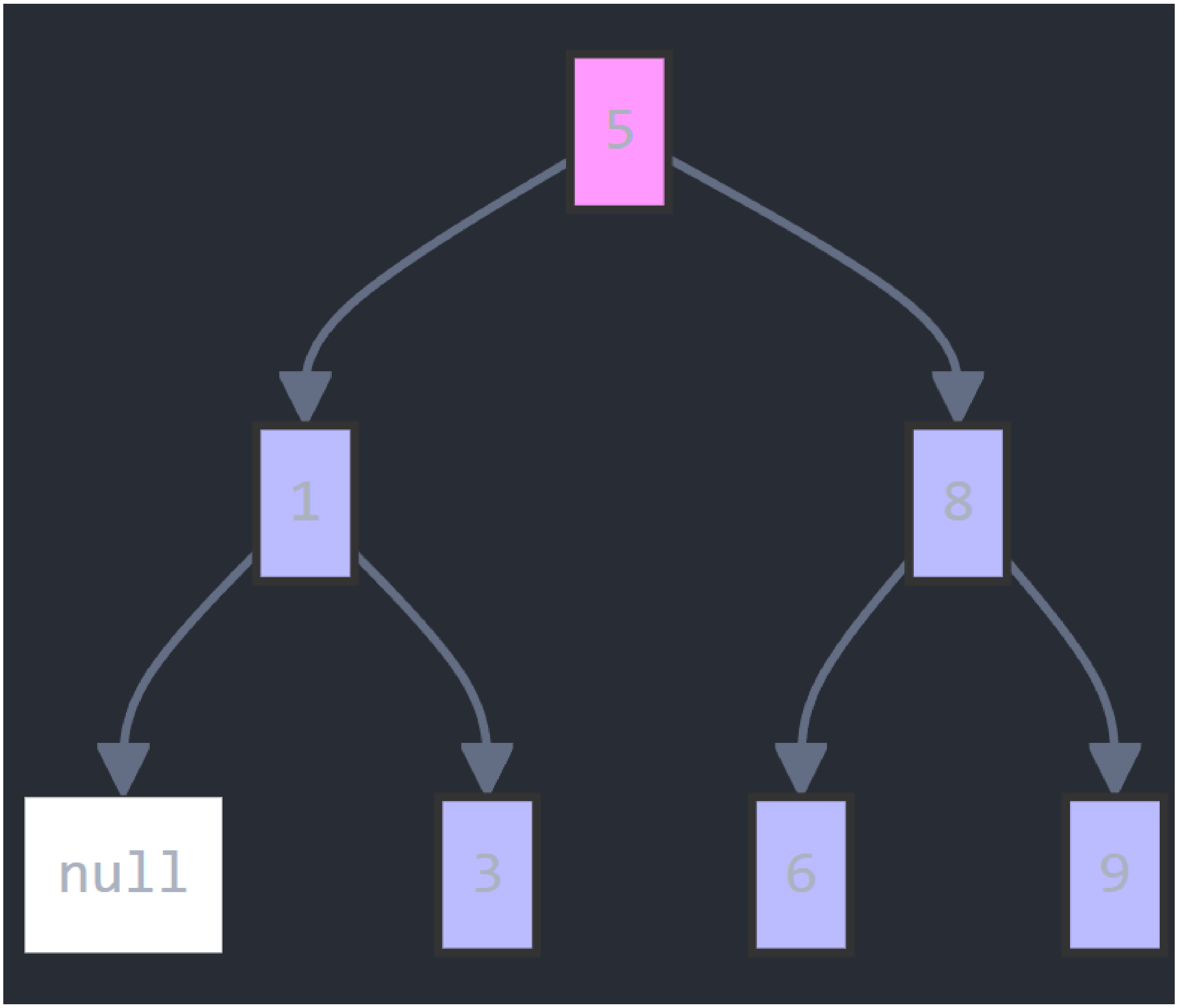
Therefore:

- **Big O (worst case):** O(n) - Dominated by traversal and possible linear insertion time
- **Big Omega (best case):** Ω(n) - Lower bounded by the traversal which is always linear
- **Big Theta (tight bound):** Θ(n) - Since both upper and lower bounds are linear

Conclusion

The provided code implements a Binary Search Tree with standard insertion and in-order traversal operations. The tight bound on the runtime complexity is Θ(n), where n is the number of nodes in the tree.

In practical terms, the efficiency of the BST operations depends on the tree's balance. For this specific example with only 6 nodes, the difference between balanced and unbalanced trees is minimal, but for larger datasets, maintaining balance becomes crucial for performance.


5
1
8
null
3
6
9