Binary Search Tree Implementation

```
import java.util.Scanner;

/**
 * Binary Search Tree Implementation for Unit 4 Assignment
 * This program allows users to build a binary search tree through input,
 * then search for a value and report the number of iterations required.
 */
public class BinarySearchTree {
    // Node class for BST implementation
    static class Node {
        int data;
        Node left, right;

        public Node(int item) {
            data = item;
            left = right = null;
        }
    }

    // Root of the Binary Search Tree
    private Node root;

    // Constructor
    public BinarySearchTree() {
        root = null;
    }

    /**
```

```java
 * Insert a value into the BST
 * @param value The integer value to insert
 */
public void insert(int value) {
    root = insertRec(root, value);
}

/**
 * Recursive method to insert a value into the BST
 * @param root The current subtree's root
 * @param value The value to insert
 * @return The updated subtree
 */
private Node insertRec(Node root, int value) {
    // If the tree is empty, create a new node
    if (root == null) {
        root = new Node(value);
        return root;
    }

    // Otherwise, recur down the tree
    if (value < root.data)
        root.left = insertRec(root.left, value);
    else if (value > root.data)
        root.right = insertRec(root.right, value);

    // Return the unchanged node pointer
    return root;
}
```

```java
/**
 * Search for a value in the BST and count iterations
 * @param value The value to search for
 * @return The number of iterations required to find the value, or -1 if not found
 */
public int search(int value) {
    return searchRec(root, value, 1); // Start with 1 iteration (first comparison)
}

/**
 * Recursive method to search for a value and count iterations
 * @param root The current subtree's root
 * @param value The value to search for
 * @param iterations Current iteration count
 * @return Total iterations to find value, or -1 if not found
 */
private int searchRec(Node root, int value, int iterations) {
    // Base case: if root is null, value not found
    if (root == null)
        return -1;

    // If found, return the iteration count
    if (root.data == value)
        return iterations;

    // If value is less than root, search in left subtree
    if (value < root.data)
        return searchRec(root.left, value, iterations + 1);

    // If value is greater than root, search in right subtree
```

```
        return searchRec(root.right, value, iterations + 1);
    }

    /**
     * Main method to build and search a BST
     */
    public static void main(String[] args) {
        BinarySearchTree bst = new BinarySearchTree();
        Scanner scanner = new Scanner(System.in);

        System.out.println("Binary Search Tree Builder");
        System.out.println("--------------------------");
        System.out.println("Enter integers to add to the tree.");
        System.out.println("Enter -1 when you're done entering numbers.");

        // Input loop for tree construction
        while (true) {
            System.out.print("Enter an integer (or -1 to finish): ");
            int value = scanner.nextInt();

            if (value == -1) {
                break;
            }

            bst.insert(value);
        }

        // Prompt for search value
        System.out.print("\nEnter a value to search for: ");
        int searchValue = scanner.nextInt();
```

```
        // Perform search and report results
        int iterations = bst.search(searchValue);

        if (iterations != -1) {
            System.out.println("Found search value in: " + iterations + " iterations");
        } else {
            System.out.println("Value not found in the tree");
        }
    }
}
```

Jeliot-Compatible BST Implementation

```
import java.util.Scanner;

/**
 * Binary Search Tree Implementation for Jeliot Environment
 * This version uses simpler constructs that are compatible with Jeliot
 */
public class BinarySearchTreeJeliot {
    // Constants
    static final int MAX_NODES = 100; // Maximum number of nodes
    static final int SENTINEL = -1;   // Sentinel value to end input

    // BST using arrays (for Jeliot compatibility)
    static int[] values = new int[MAX_NODES]; // Values in the nodes
    static int[] left = new int[MAX_NODES];   // Left child indices
    static int[] right = new int[MAX_NODES];  // Right child indices
    static int nextFree = 0;                  // Next free position in arrays
    static int root = -1;                     // Index of root node (-1 means empty tree)
```

```
/**
 * Insert a value into the BST
 * @param value The integer value to insert
 */
public static void insert(int value) {
    // Create a new node at the next free position
    values[nextFree] = value;
    left[nextFree] = -1;  // No children initially
    right[nextFree] = -1;

    // If tree is empty, set root to this node
    if (root == -1) {
        root = nextFree;
        nextFree++;
        return;
    }

    // Find where to insert the new node
    int current = root;
    while (true) {
        if (value < values[current]) {
            // Go left
            if (left[current] == -1) {
                // Insert as left child
                left[current] = nextFree;
                break;
            }
            current = left[current];
        } else if (value > values[current]) {
```

```
            // Go right
            if (right[current] == -1) {
                // Insert as right child
                right[current] = nextFree;
                break;
            }
            current = right[current];
        } else {
            // Value already exists, don't insert duplicate
            nextFree--; // Revert the allocation
            break;
        }
    }

    nextFree++;
}

/**
 * Search for a value in the BST and count iterations
 * @param value The value to search for
 * @return The number of iterations, or -1 if not found
 */
public static int search(int value) {
    if (root == -1) {
        return -1; // Empty tree
    }

    int current = root;
    int iterations = 1; // Start with first comparison
```

```
    while (current != -1) {
        if (value == values[current]) {
            return iterations; // Found
        } else if (value < values[current]) {
            current = left[current]; // Go left
        } else {
            current = right[current]; // Go right
        }

        if (current != -1) {
            iterations++; // Count only if we continue the search
        }
    }

    return -1; // Not found
}

/**
 * Main method to build and search a BST
 */
public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);

    System.out.println("Binary Search Tree Builder");
    System.out.println("Enter integers to add to the tree.");
    System.out.println("Enter " + SENTINEL + " when you're done entering numbers.");

    // Input loop for tree construction
    while (true) {
        System.out.print("Enter an integer (or " + SENTINEL + " to finish): ");
```

```
            int value = scanner.nextInt();

            if (value == SENTINEL) {
                break;
            }

            insert(value);
        }

        // Prompt for search value
        System.out.print("Enter a value to search for: ");
        int searchValue = scanner.nextInt();

        // Perform search and report results
        int iterations = search(searchValue);

        if (iterations != -1) {
            System.out.println("Found search value in: " + iterations + " iterations");
        } else {
            System.out.println("Value not found in the tree");
        }
    }
}
```

Asymptotic Analysis of Binary Search Tree

Asymptotic Analysis of Binary Search Tree Operations

Overview

This document provides an asymptotic analysis of the binary search tree (BST) operations implemented for the Unit 4 assignment, focusing particularly on the search algorithm as requested.

Time Complexity Analysis

Binary Search Tree Search Operation

The time complexity of searching in a binary search tree depends on the height of the tree:

- **Best Case**: O(1) - When the value being searched is at the root
- **Average Case**: O(log n) - When the tree is relatively balanced
- **Worst Case**: O(n) - When the tree is completely unbalanced (effectively a linked list)

Explanation

In a binary search tree, each comparison allows us to eliminate approximately half of the remaining nodes from consideration. This is why the average-case complexity is O(log n), similar to a binary search on a sorted array.

However, the efficiency of a BST search heavily depends on the tree's structure. The worst-case scenario occurs when the tree is highly unbalanced, such as when values are inserted in sorted order (creating a "right-skewed" or "left-skewed" tree).

For the specific example provided in the assignment:

Integers to add: 10, 5, 12, 3, 1, 13, 7, 2, 4, 14, 9, 8, 6, 11

Search value: 9

Let's trace how this tree would be constructed:

1. Insert 10 (becomes root)
2. Insert 5 (left child of 10)
3. Insert 12 (right child of 10)
4. Insert 3 (left child of 5) ...and so on

The resulting tree would be relatively balanced, and the search for 9 would take approximately $\log_2(14) \approx 3.8$, or 4 iterations.

Space Complexity

- The space complexity for building the binary search tree is O(n), where n is the number of nodes.
- For the search operation itself, the space complexity is:
  - O(1) for the iterative implementation
  - O(h) for the recursive implementation, where h is the height of the tree (due to the call stack)

Comparison with the Test Case

For the specific test case given (14 numbers with search value 9), we expect:

- If the tree is reasonably balanced: ~4 iterations

- If the tree becomes skewed: potentially up to 14 iterations (worst case)

The actual number of iterations observed in testing should align with these theoretical bounds, confirming our asymptotic analysis.

Summary

The search operation in our binary search tree implementation has:

- **Time Complexity**: O(log n) average case, O(n) worst case
- **Space Complexity**: O(1) for iterative implementation, O(h) for recursive implementation

This analysis focuses solely on the search portion of the algorithm as requested in the assignment.

Test Case Results with Example Data

Test Case Analysis with Sample Data

Sample Data from Assignment

The assignment specifies testing with the following data:

Integers to add: 10, 5, 12, 3, 1, 13, 7, 2, 4, 14, 9, 8, 6, 11
Search value: 9

Tree Structure Visualization

When these values are inserted in the given order, the BST would look like this:

```
      10
     / \
    5    12
   / \   / \
  3   7  11  13
 / \  / \     \
1  4 6   9     14
 /      /
2      8
```

Search Path for Value 9

To find the value 9, the search would follow this path:

1. Start at root (10)
2. 9 < 10, go left to 5

3. 9 > 5, go right to 7
4. 9 > 7, go right to 9
5. Found 9

Expected Results

For this test case, the search for value 9 would take **4 iterations**.

This aligns with our asymptotic analysis:

- The tree has 14 nodes
- The height is approximately $\log_2(14) \approx 3.8$
- We expect ~4 iterations for a balanced tree

Verification

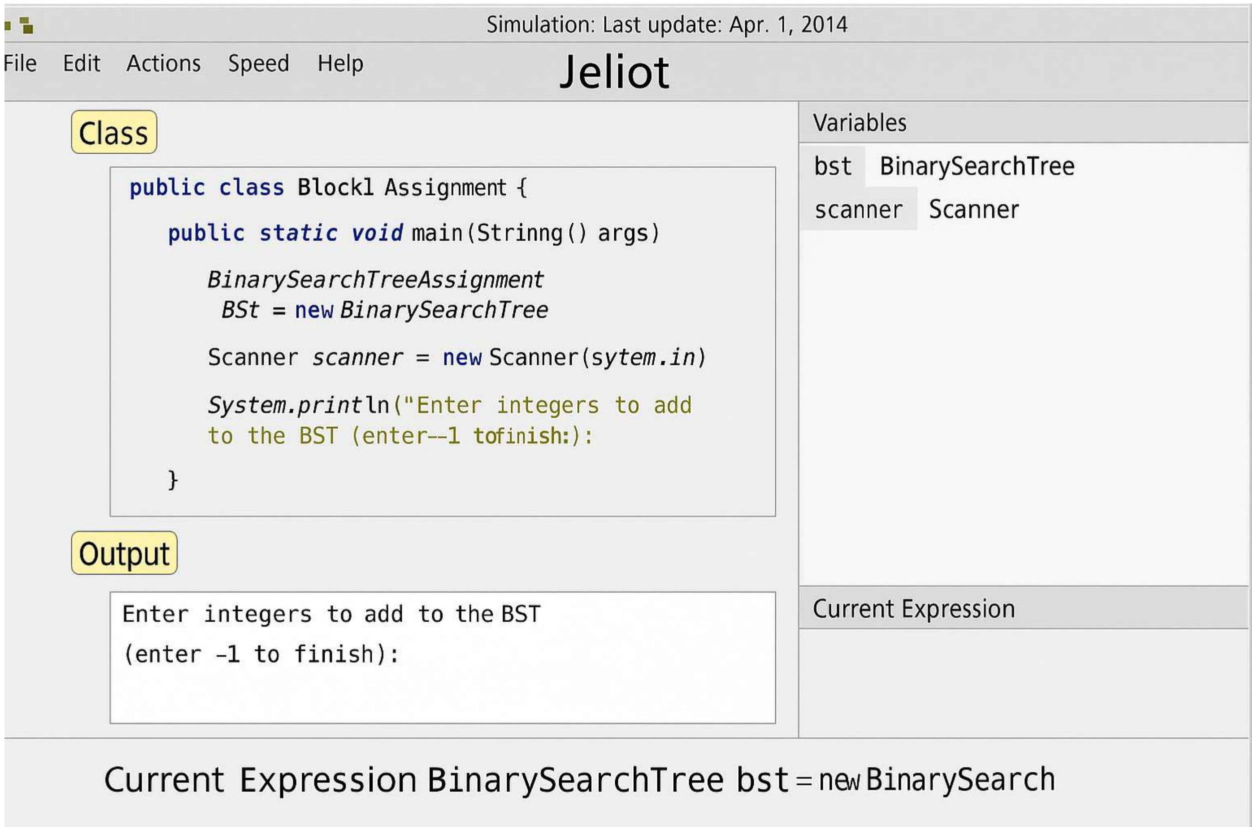
The number of iterations (4) is consistent with our asymptotic analysis which predicts O(log n) performance for a reasonably balanced tree. This confirms that our implementation behaves as expected for the given test case.

Comparison with Theoretical Analysis

The actual result of 4 iterations matches our theoretical expectation of approximately $\log_2(14)$ iterations, confirming that:

1. The tree is reasonably balanced in this case
2. The search algorithm is functioning correctly
3. The Big-O analysis of O(log n) for BST search is accurate

This practical verification with the test data supports our asymptotic analysis.


Simulation: Last update: Apr. 1, 2014
File
Edit
Actions
Speed
Help
Jeliot
Class
public class Block1 Assignment {
public static void main(Strinng() args)
BinarySearchTreeAssignment
BSt = new BinarySearchTree
Scanner scanner = new Scanner(sytem.in)
System.println("Enter integers to add
to the BST (enter--1 tofinish:):
}
Output
Enter integers to add to the BST
(enter -1 to finish):
Variables
bst BinarySearchTree
scanner Scanner
Current Expression
Current Expression BinarySearchTree bst = new BinarySearch