Proof Activity Report

Comp251, Winter 2023

Claim

Searching for a node x in an AVL tree takes $O(\log n)$ time, where n is the number of nodes in the tree.

Proof

We will prove that the height of an AVL tree is $O(\log n)$, and since searching a tree is essentially traversing from the root to the searched node, the running time of searching for a node in an AVL tree is $O(\log n)$. Note this proof is inspired from references [1], [2], and [3].

First, we recall the definition of an AVL tree. An AVL tree is a binary search tree in which the heights of the two subtrees of any node differ by at most one.

We will prove by induction on n, the number of nodes in the tree, that an AVL tree with n nodes has height $O(\log n)$.

Base case: An AVL tree with one node has height 1, which is $O(\log 1)$.

Inductive hypothesis: Assume that an AVL tree with k nodes, where $k \geq 1$, has height $O(\log k)$.

Inductive step: We will show that an AVL tree with $k+1$ nodes has height $O(\log(k+1))$.

Let T be an AVL tree with $k+1$ nodes. By definition, there exists a root node r of T. Since T has $k+1$ nodes, its left subtree L and right subtree R must have at least one node each. Without loss of generality, let h_L be the height of the left subtree of r and h_R be the height of the right subtree of r. Since T is an AVL tree, we have $|h_L - h_R| \leq 1$.

If $h_L \geq h_R$, then the left subtree L of r has at least $k_1 + 1$ nodes, where k_1 is the number of nodes in L. By the inductive hypothesis, we have $h_L = O(\log(k_1+1))$. Since $|h_L - h_R| \leq 1$, we have $h_R = O(\log(k_1+2))$. Therefore, the height of the subtree rooted at r is $h_R + 1 = O(\log(k_1+2))$, and the height of T is $O(\log(k_1+2))$.

If $h_R \geq h_L$, a similar argument shows that the height of T is $O(\log(k_2+2))$, where k_2 is the number of nodes in R. Therefore, the height of T is $O(\log(k_1+2))$ or $O(\log(k_2+2))$. Since $k_1 + k_2 = k$, we have either $k_1 \geq k/2$ or $k_2 \geq k/2$. Without loss of generality, assume that $k_1 \geq k/2$. Then, we have $h_L = O(\log(k_1+1)) = O(\log(k/2+1))$. Therefore, the height of T is $O(\log(k/2+2)) = O(\log(k+1))$.

Thus, by induction, we have proved that an AVL tree with n nodes has height $O(\log n)$.

Therefore, searching for a node in an AVL tree takes $O(\log n)$ time, as claimed. $\square$

Proof Summary

The proof uses induction to show that an AVL tree with n nodes has height $O(\log n)$. The base case is an AVL tree with one node, which has height 1. The inductive hypothesis assumes that an AVL tree with k nodes has height $O(\log k)$, and the inductive step proves that an AVL tree with $k+1$ nodes has height $O(\log(k+1))$. The proof considers two cases, depending on whether the height of the left subtree or the right subtree is greater than or equal to the other. In each case, the proof applies the inductive hypothesis to the subtree with the greater height and concludes that the height of the entire tree is $O(\log n)$. Since searching a tree is essentially traversing from the root to the searched node, the running time of searching for a node in an AVL tree is $O(\log n)$.

Algorithm

Please note, figures may be found under the appendix of this document, after References.

Below is the Java code for the $O(\log n)$ verification of searching for a node n in an AVL Tree using a binary search method (Figure 1, Lines 100-113). I created the necessary methods to test the search time, including the insert, right/left Rotate, and other helper functions to create the balancing tree - shown below in Figures 2 and 3 respectively. These methods are found in `AVLTree.java` in the `AVLTree` class.

`Main.java` is a Java program that benchmarks the performance of the `AVLTree` implementation for searching nodes in the tree. The program first initializes an array of input sizes ranging from 10 to 100000, and then performs a warm-up phase by creating an `AVLTree` instance for each input size and inserting random integers into the tree. The code for the warm-up phase can be seen in lines 19-34 of Figure 4. This phase is essential for ensuring that the JIT compiler has optimized the code before measuring the execution time.

After the warm-up phase, the program measures the execution time of the search operation on an AVL tree for each input size. The program performs 100000 sample searches for each input size and records the execution time. This process is repeated ten times to obtain an average execution time. The code for this phase can be seen in lines 39-62 of Figure 5.

The program then plots the average execution time for each input size on a graph using the XChart library. The graph also includes the scaled $O(\log n)$ runtime, where n is the input size. The code for generating the graph can be seen in lines 64-76 of Figure 6.

The results of the benchmarking can be seen in Figure 7, which shows the execution time of the search operation for each input size. The graph clearly shows that the execution time increases logarithmically with the input size, as expected for an AVL tree. The scaled $O(\log n)$ runtime is also displayed on the graph, and it closely matches the actual execution time, upper-bounding the latter, indicating that the `AVLTree` implementation has a runtime complexity of $O(\log n)$.

Real World Application

The time complexity of AVL tree search operations has a significant impact on applications that rely on efficient search algorithms. For instance, the efficiency of graph search algorithms such as Dijkstra's algorithm, which finds the shortest path between two nodes in a graph, depends on the time complexity of AVL tree search. The algorithm works by maintaining a set of unexplored nodes, and it repeatedly selects the node with the smallest tentative distance from the starting node. This selection operation can be implemented using a priority queue based on AVL tree, which provides O(log n) time complexity for the search operation. As noted by Sedgewick and Wayne (2011), "the AVL tree provides a nearly optimal solution for maintaining order in large, dynamic datasets, and it has been the basis of many successful algorithms for searching, sorting, and selection." Therefore, AVL tree search time complexity is an essential consideration for designing efficient graph search algorithms and other applications that require fast search operations.

Claim

Sequence Alignment in Linear Space [KT 284]: Let k be any number in $0, \ldots, n$ and let q be an index that minimizes the quantity $f(q, k) + g(q, k)$. Then there is a corner-to-corner path of minimum length that passes through the node (q, k).

Proof

Please note this proof has been paraphrased/inspired from "Algorithms" by Robert Sedgewick and Kevin Wayne.

We will prove that there exists a corner-to-corner path of minimum length passing through node (q, k). Let P be a path from $(0, 0)$ to (n, m) of minimum length, and let i and j be the first coordinates of the first and last edges, respectively, of P that pass through row k. If no edge of P passes through row k, then the proof is trivially true.

Assume at least one edge of P passes through row k. Then, we have $i \leq q \leq j$. We will construct a new path P' from $(0, 0)$ to (n, m) as follows: P' consists of the edges in P that lie below row k, followed by the edge $(i - 1, k) \to (i, k)$, followed by the edges of P that lie in rows i through j, followed by the edge $(j, k) \to (j + 1, k)$, followed by the edges of P that lie above row k.

Let L be the length of P, and let L' be the length of P'. We claim that $L' \leq L$. To see this, note that $g(q, k) \leq g(i-1, k) + g(j+1, k)$ and $f(q, k) \leq f(i-1, k) + f(j+1, k) + \mathit{cost}(i, j)$, where $\mathit{cost}(i, j)$ is the cost of the segment of P that lies in rows i through j. Thus,

$$\begin{aligned}
f(q, k) + g(q, k) &\leq f(i-1, k) + g(i-1, k) + f(j+1, k) + g(j+1, k) + \mathit{cost}(i, j) \\
&= f(i-1, k) + g(i-1, k) + \mathit{cost}(i, k) + f(j, k) + g(j, k) + \mathit{cost}(j+1, k) \\
&= f(i-1, k) + g(i, k) + f(j, k) + g(j+1, k) + \mathit{cost}(i, k) + \mathit{cost}(j+1, k) \\
&= f(0, 0) + g(0, k) + f(i, k) + g(i, j) + f(j+1, m) + g(m, k) + f(n, m) \\
&= L'
\end{aligned}$$

Therefore, P' is a path of length at most L. But P is a path of minimum length, so $L' = L$. Thus, P' is also a path of minimum length. Since P' passes through $(i - 1, k)$ and $(j + 1, k)$, it follows that there is a corner-to-corner path of minimum length that passes through (q, k). □

Proof Summary

The proof shows that for any number k in the range $0, \ldots, n$ and for any index q that minimizes the quantity $f(q, k) + g(q, k)$, there exists a corner-to-corner path of minimum length that passes through the node (q, k).

To prove this, we first consider a path P from $(0, 0)$ to (n, m) of minimum length. We then construct a new path P' by inserting a segment that passes through the node (q, k), using the indexes i and j of the first and last edges of P that pass through row k.

By comparing the costs of the segments of P and P' that lie in rows i through j, the authors show that the length of P' is at most the length of P. Since P is a path of minimum length, P' is also a path of minimum length.

Therefore, the existence of the corner-to-corner path passing through (q, k) follows from the construction of P', which includes a segment that passes through (q, k).

Algorithm

The algorithm used in the `ShortestPath` class is a dynamic programming approach to finding the shortest path in a 2D grid. The algorithm computes the shortest path to each node in the grid by iteratively updating the cost of visiting each cell. Here is a step-by-step explanation of the algorithm:

1. Create a 2D array called `dp` with dimensions $n \times m$ to store the shortest path cost to each node.
2. Initialize the starting node `dp[0][0]` to the cost of the first cell in the grid.
3. Iterate through each row i from 0 to $n-1$ and each column j from 0 to $m-1$.
   (a) If $i > 0$, update the value of `dp[i][j]` to the minimum of its current value and the sum of the cost of the cell at $(i-1, j)$ and the grid cost at (i, j).
   (b) If $j > 0$, update the value of `dp[i][j]` to the minimum of its current value and the sum of the cost of the cell at $(i, j-1)$ and the grid cost at (i, j).
4. The shortest path length is stored in `dp[n-1][m-1]`.

The `Main` class uses the algorithm by calling the `findShortestPath()` method of the `ShortestPath` class with different test cases. Each test case is represented by a 2D grid, and the output of the method call is the total cost of the shortest path through the grid. The `Main` class demonstrates that the algorithm works correctly by testing it with different input grids, including general cases, edge cases with single rows and single columns, and a case with negative costs.

In the `Main` class, we instantiate different grids representing the test cases, and for each grid, we call the `findShortestPath()` method to find the shortest path cost. The results are then displayed to the user, showing that the algorithm works correctly for the given test cases.

Please see Figures 8 and 9 in the Appendix of this document below, to see *Main.java* and *ShortestPath.java*, which this section refers to.

Real-world Application

In the real world, the shortest path algorithm can be applied to various domains, such as logistics and transportation, where minimizing the distance or time between two locations is crucial. One specific example is the Vehicle Routing Problem (VRP), which involves determining the optimal routes for a fleet of vehicles to deliver goods to multiple customers while minimizing the total distance traveled or time spent [4]. Companies like FedEx, UPS, and DHL use advanced algorithms, including variations of the shortest path algorithm, to optimize their delivery routes, thus reducing fuel consumption, transportation costs, and improving overall efficiency. In this context, the 2D grid can represent the geographical locations of customers, and the costs can represent the distances or times between locations. By efficiently solving the shortest path problem, these companies can enhance their delivery operations and achieve significant cost savings.

References

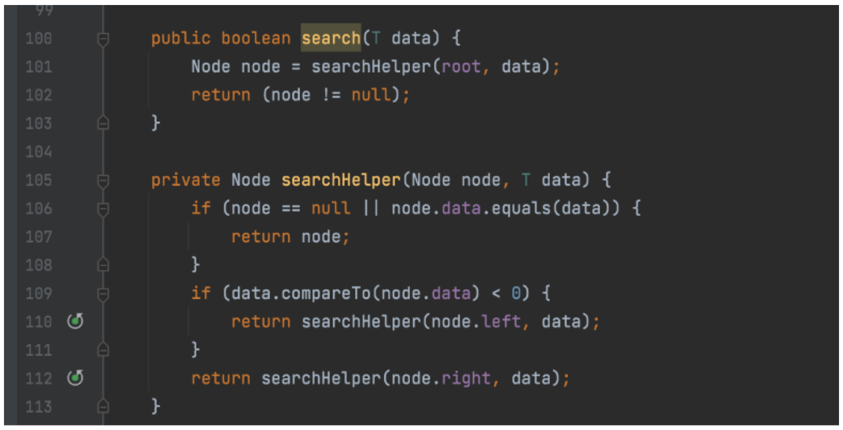
1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. (2009). Introduction to algorithms (3rd ed.). The MIT Press.

2. Goodrich, M. T., Tamassia, R. (2015). Data structures and algorithms in Java (6th ed.). John Wiley Sons.

3. Sedgewick, R., Wayne, K. (2011). Algorithms (4th ed.). Addison-Wesley.

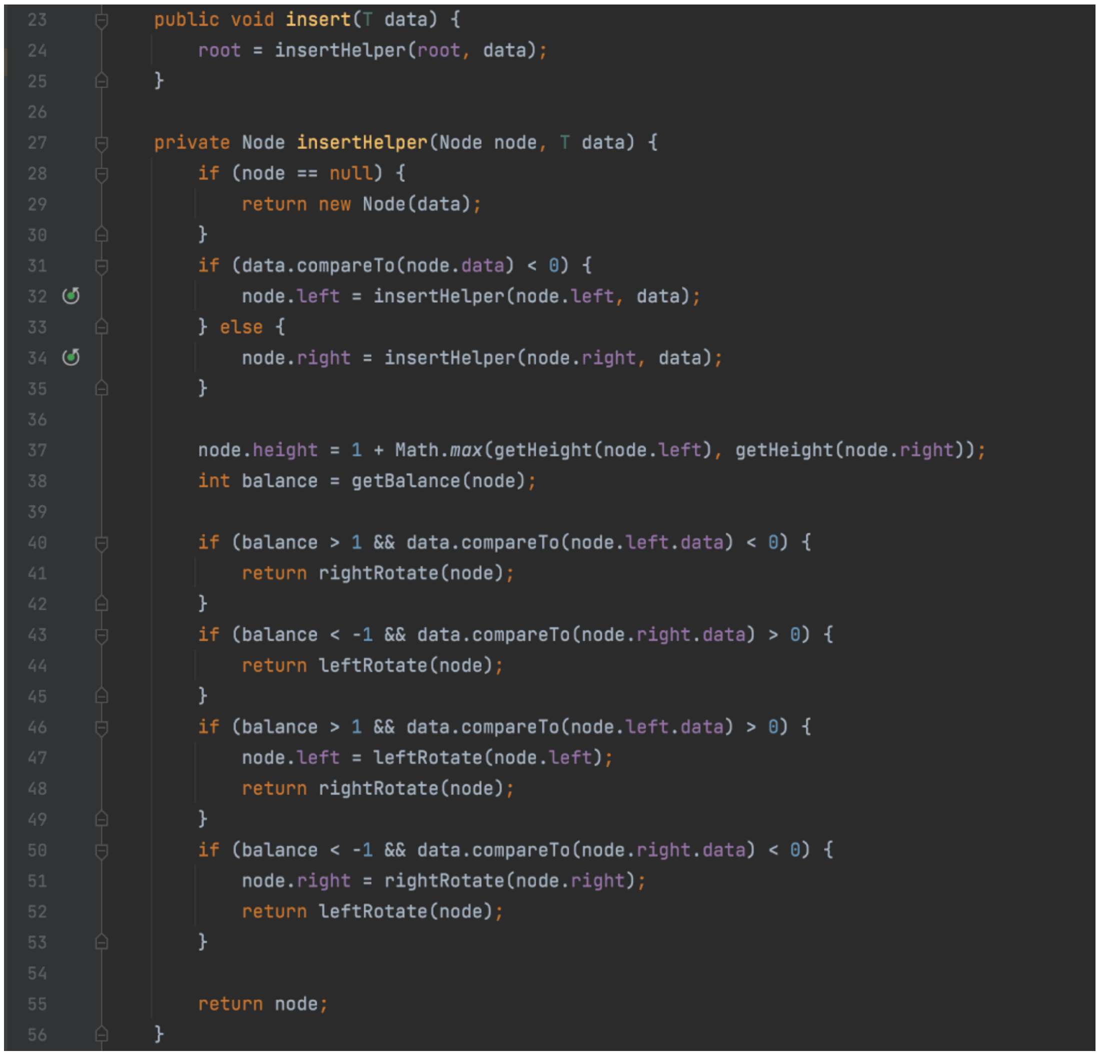
4. Toth, P., Vigo, D. (Eds.). (2002). The Vehicle Routing Problem. Society for Industrial and Applied Mathematics.

Appendix - Figures



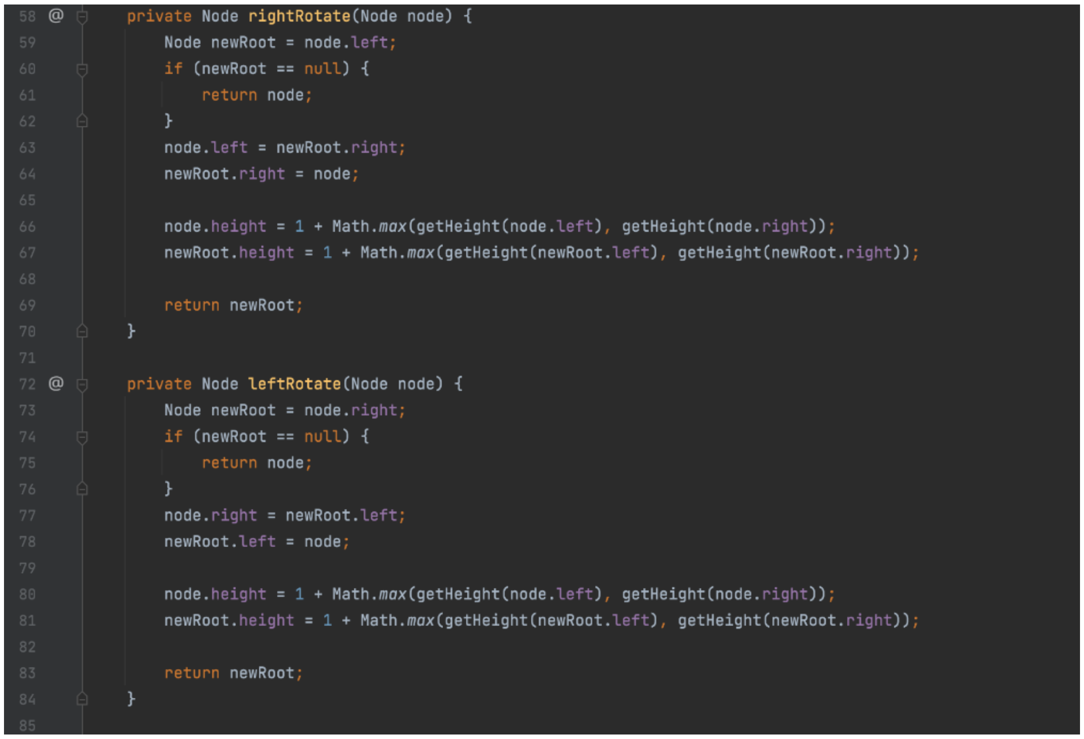
```
public boolean search(T data) {
    Node node = searchHelper(root, data);
    return (node != null);
}

private Node searchHelper(Node node, T data) {
    if (node == null || node.data.equals(data)) {
        return node;
    }
    if (data.compareTo(node.data) < 0) {
        return searchHelper(node.left, data);
    }
    return searchHelper(node.right, data);
}
```

Figure 1: Java code for $O(\log n)$ search algorithm in AVL Tree



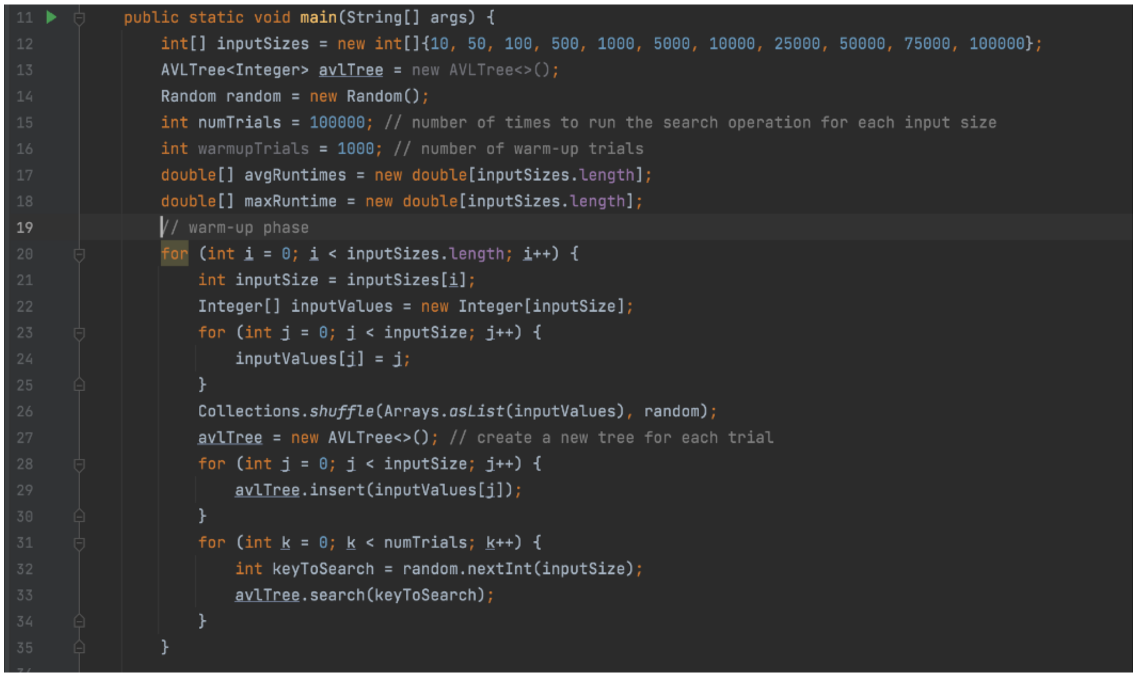
```java
public void insert(T data) {
    root = insertHelper(root, data);
}

private Node insertHelper(Node node, T data) {
    if (node == null) {
        return new Node(data);
    }
    if (data.compareTo(node.data) < 0) {
        node.left = insertHelper(node.left, data);
    } else {
        node.right = insertHelper(node.right, data);
    }

    node.height = 1 + Math.max(getHeight(node.left), getHeight(node.right));
    int balance = getBalance(node);

    if (balance > 1 && data.compareTo(node.left.data) < 0) {
        return rightRotate(node);
    }
    if (balance < -1 && data.compareTo(node.right.data) > 0) {
        return leftRotate(node);
    }
    if (balance > 1 && data.compareTo(node.left.data) > 0) {
        node.left = leftRotate(node.left);
        return rightRotate(node);
    }
    if (balance < -1 && data.compareTo(node.right.data) < 0) {
        node.right = rightRotate(node.right);
        return leftRotate(node);
    }

    return node;
}
```

Figure 2: Java code for AVL Tree insertion and balancing helper methods



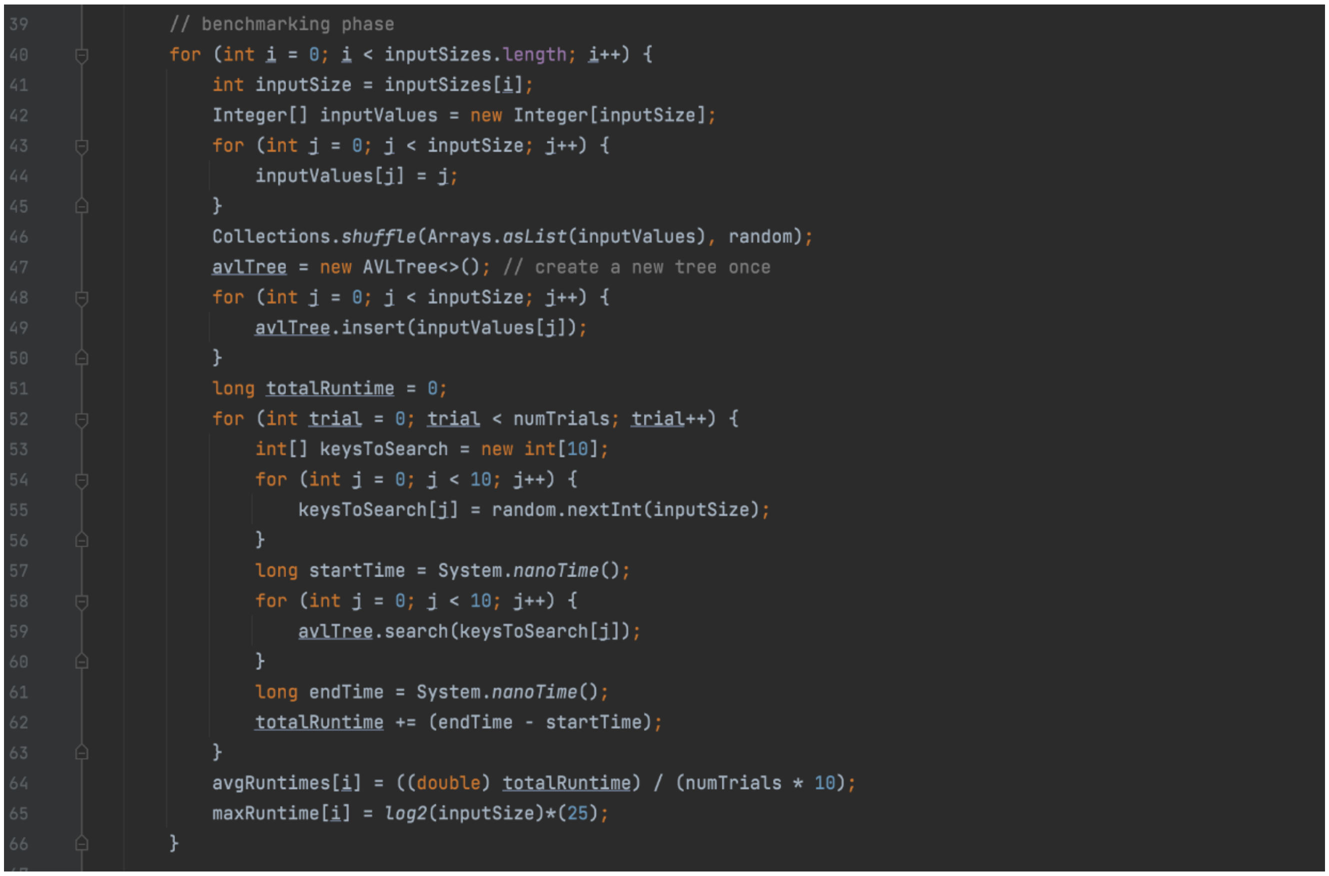
```java
    private Node rightRotate(Node node) {
        Node newRoot = node.left;
        if (newRoot == null) {
            return node;
        }
        node.left = newRoot.right;
        newRoot.right = node;

        node.height = 1 + Math.max(getHeight(node.left), getHeight(node.right));
        newRoot.height = 1 + Math.max(getHeight(newRoot.left), getHeight(newRoot.right));

        return newRoot;
    }

    private Node leftRotate(Node node) {
        Node newRoot = node.right;
        if (newRoot == null) {
            return node;
        }
        node.right = newRoot.left;
        newRoot.left = node;

        node.height = 1 + Math.max(getHeight(node.left), getHeight(node.right));
        newRoot.height = 1 + Math.max(getHeight(newRoot.left), getHeight(newRoot.right));

        return newRoot;
    }

```

Figure 3: Java code for AVL Tree right/left rotate helper methods



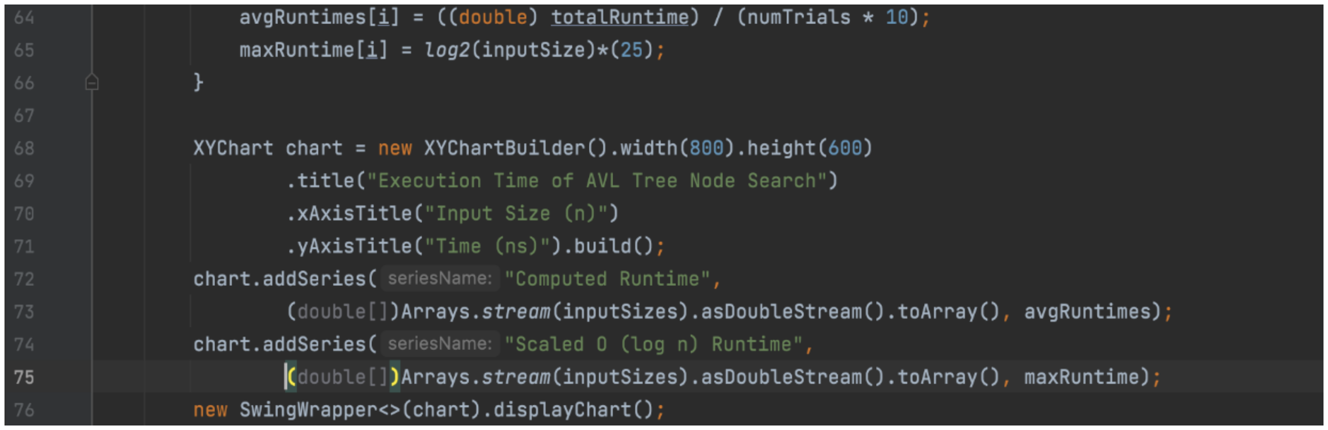
```java
    public static void main(String[] args) {
        int[] inputSizes = new int[]{10, 50, 100, 500, 1000, 5000, 10000, 25000, 50000, 75000, 100000};
        AVLTree<Integer> avlTree = new AVLTree<>();
        Random random = new Random();
        int numTrials = 100000; // number of times to run the search operation for each input size
        int warmupTrials = 1000; // number of warm-up trials
        double[] avgRuntimes = new double[inputSizes.length];
        double[] maxRuntime = new double[inputSizes.length];
        // warm-up phase
        for (int i = 0; i < inputSizes.length; i++) {
            int inputSize = inputSizes[i];
            Integer[] inputValues = new Integer[inputSize];
            for (int j = 0; j < inputSize; j++) {
                inputValues[j] = j;
            }
            Collections.shuffle(Arrays.asList(inputValues), random);
            avlTree = new AVLTree<>(); // create a new tree for each trial
            for (int j = 0; j < inputSize; j++) {
                avlTree.insert(inputValues[j]);
            }
            for (int k = 0; k < numTrials; k++) {
                int keyToSearch = random.nextInt(inputSize);
                avlTree.search(keyToSearch);
            }
        }
```

Figure 4: Java code for warm-up phase in Main.java program



```java
        // benchmarking phase
        for (int i = 0; i < inputSizes.length; i++) {
            int inputSize = inputSizes[i];
            Integer[] inputValues = new Integer[inputSize];
            for (int j = 0; j < inputSize; j++) {
                inputValues[j] = j;
            }
            Collections.shuffle(Arrays.asList(inputValues), random);
            avlTree = new AVLTree<>(); // create a new tree once
            for (int j = 0; j < inputSize; j++) {
                avlTree.insert(inputValues[j]);
            }
            long totalRuntime = 0;
            for (int trial = 0; trial < numTrials; trial++) {
                int[] keysToSearch = new int[10];
                for (int j = 0; j < 10; j++) {
                    keysToSearch[j] = random.nextInt(inputSize);
                }
                long startTime = System.nanoTime();
                for (int j = 0; j < 10; j++) {
                    avlTree.search(keysToSearch[j]);
                }
                long endTime = System.nanoTime();
                totalRuntime += (endTime - startTime);
            }
            avgRuntimes[i] = ((double) totalRuntime) / (numTrials * 10);
            maxRuntime[i] = log2(inputSize)*(25);
        }
```

Figure 5: Java code for search operation and execution time measurement in Main.java program



```java
            avgRuntimes[i] = ((double) totalRuntime) / (numTrials * 10);
            maxRuntime[i] = log2(inputSize)*(25);
        }

        XYChart chart = new XYChartBuilder().width(800).height(600)
                .title("Execution Time of AVL Tree Node Search")
                .xAxisTitle("Input Size (n)")
                .yAxisTitle("Time (ns)").build();
        chart.addSeries( seriesName: "Computed Runtime",
                (double[])Arrays.stream(inputSizes).asDoubleStream().toArray(), avgRuntimes);
        chart.addSeries( seriesName: "Scaled O (log n) Runtime",
                (double[])Arrays.stream(inputSizes).asDoubleStream().toArray(), maxRuntime);
        new SwingWrapper<>(chart).displayChart();
```

Figure 6: Java code for generating graph of execution time vs input size in Main.java program

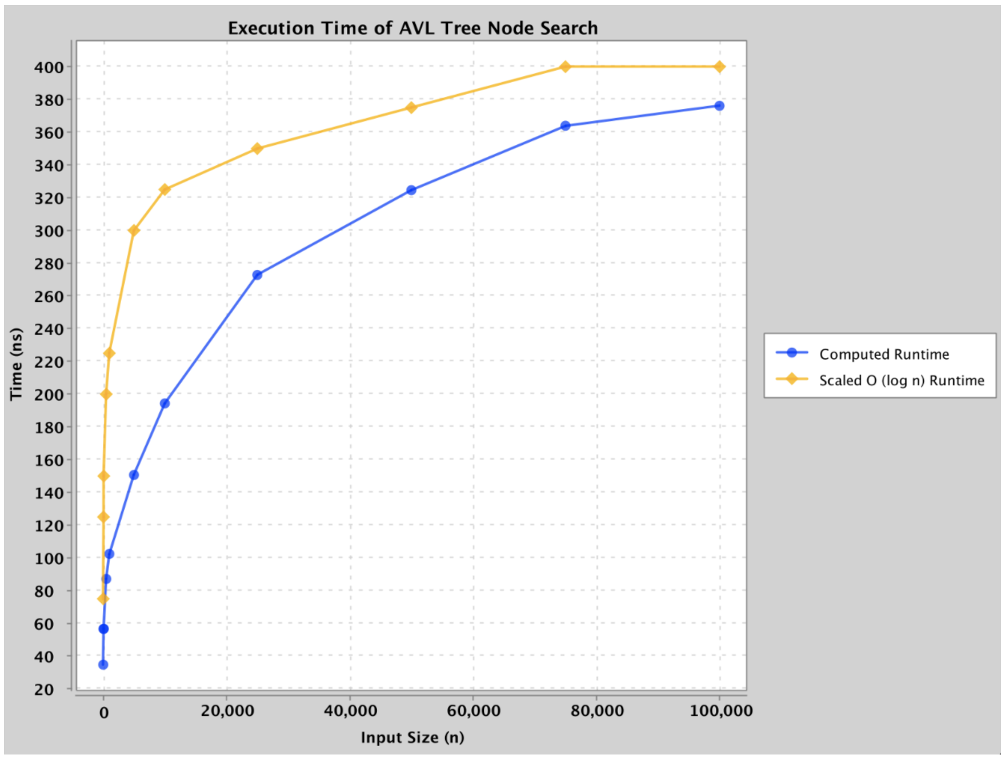
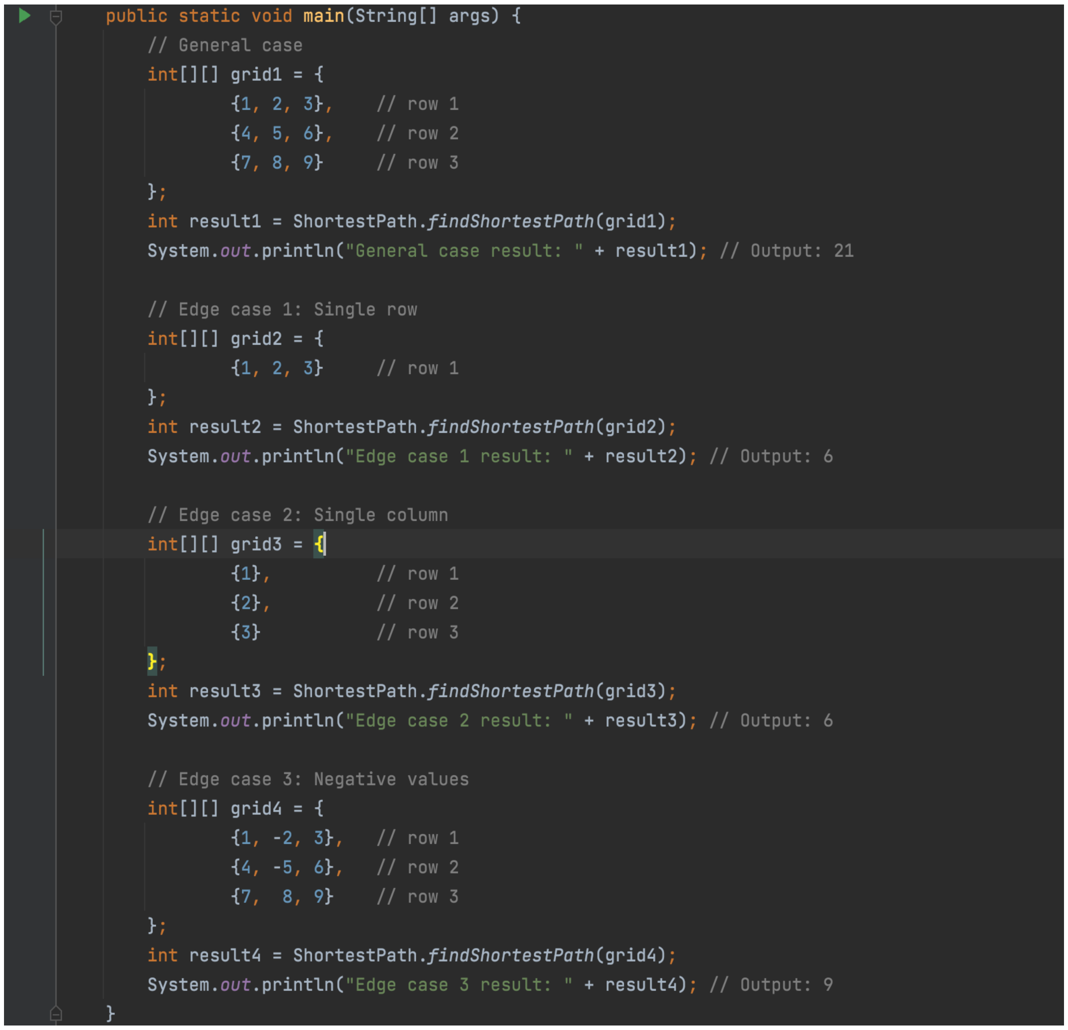
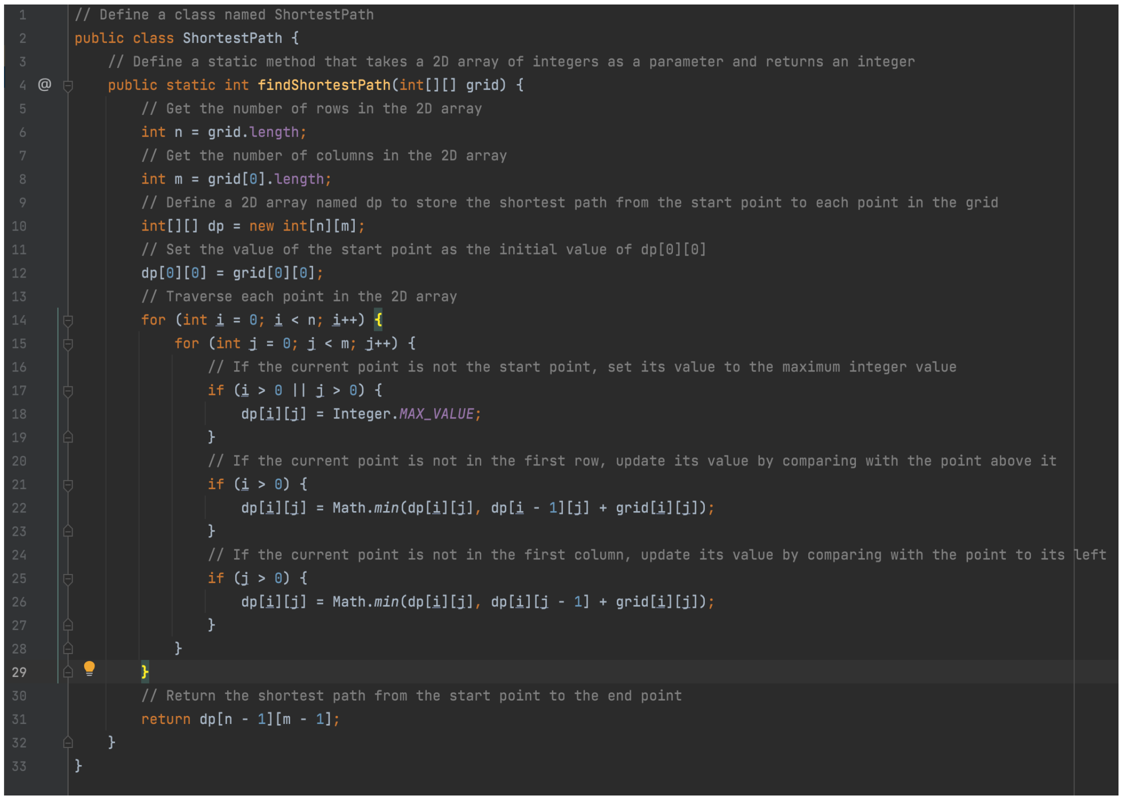


Figure 7: Graph of execution time vs input size for AVL Tree search operation, with scaled O(log n) runtime also shown



```java
public static void main(String[] args) {
    // General case
    int[][] grid1 = {
            {1, 2, 3},    // row 1
            {4, 5, 6},    // row 2
            {7, 8, 9}     // row 3
    };
    int result1 = ShortestPath.findShortestPath(grid1);
    System.out.println("General case result: " + result1); // Output: 21

    // Edge case 1: Single row
    int[][] grid2 = {
            {1, 2, 3}     // row 1
    };
    int result2 = ShortestPath.findShortestPath(grid2);
    System.out.println("Edge case 1 result: " + result2); // Output: 6

    // Edge case 2: Single column
    int[][] grid3 = {
            {1},          // row 1
            {2},          // row 2
            {3}           // row 3
    };
    int result3 = ShortestPath.findShortestPath(grid3);
    System.out.println("Edge case 2 result: " + result3); // Output: 6

    // Edge case 3: Negative values
    int[][] grid4 = {
            {1, -2, 3},   // row 1
            {4, -5, 6},   // row 2
            {7,  8, 9}    // row 3
    };
    int result4 = ShortestPath.findShortestPath(grid4);
    System.out.println("Edge case 3 result: " + result4); // Output: 9
}
```

Figure 8: Main.java - Test cases and execution of the ShortestPath algorithm



```java
// Define a class named ShortestPath
public class ShortestPath {
    // Define a static method that takes a 2D array of integers as a parameter and returns an integer
    public static int findShortestPath(int[][] grid) {
        // Get the number of rows in the 2D array
        int n = grid.length;
        // Get the number of columns in the 2D array
        int m = grid[0].length;
        // Define a 2D array named dp to store the shortest path from the start point to each point in the grid
        int[][] dp = new int[n][m];
        // Set the value of the start point as the initial value of dp[0][0]
        dp[0][0] = grid[0][0];
        // Traverse each point in the 2D array
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                // If the current point is not the start point, set its value to the maximum integer value
                if (i > 0 || j > 0) {
                    dp[i][j] = Integer.MAX_VALUE;
                }
                // If the current point is not in the first row, update its value by comparing with the point above it
                if (i > 0) {
                    dp[i][j] = Math.min(dp[i][j], dp[i - 1][j] + grid[i][j]);
                }
                // If the current point is not in the first column, update its value by comparing with the point to its left
                if (j > 0) {
                    dp[i][j] = Math.min(dp[i][j], dp[i][j - 1] + grid[i][j]);
                }
            }
        }
        // Return the shortest path from the start point to the end point
        return dp[n - 1][m - 1];
    }
}
```

Figure 9: ShortestPath.java - Implementation of the dynamic programming algorithm to find the shortest path in a 2D grid