Tree Comparison Visualizer User Manual

CSE-411 Advanced Programming Techniques by Prof. Corey Montella

Author: Sharmili Rudraraju

Introduction

The Tree Comparison Visualizer is a visualization software tool designed to help in understanding and compare the behavior of Binary Search Trees (BST) and AVL Trees. This manual provides comprehensive guidance on installing, using, and getting the most out of the visualization tool.

This application serves as both a learning aid and a practical tool for understanding tree data structures. By providing side-by-side visualization of BST and AVL trees, along with real-time performance metrics, users can gain deep insights into the differences between these fundamental data structures.

System Requirements and Installation

Before installing the Tree Comparison Visualizer, ensure your system meets the following requirements:

The application requires *Python 3.7* or higher installed on your computer. It works on Windows. Your display should have a minimum resolution of 1400x900 pixels to properly view all elements of the interface. While the application is lightweight, least 4GB of RAM for smooth operation is recommended, especially during stress testing with larger trees.

To install the application:

- First, ensure *Python 3.7* or higher is installed on your system. You can verify this by opening a terminal or command prompt and typing:

```
python --version
```

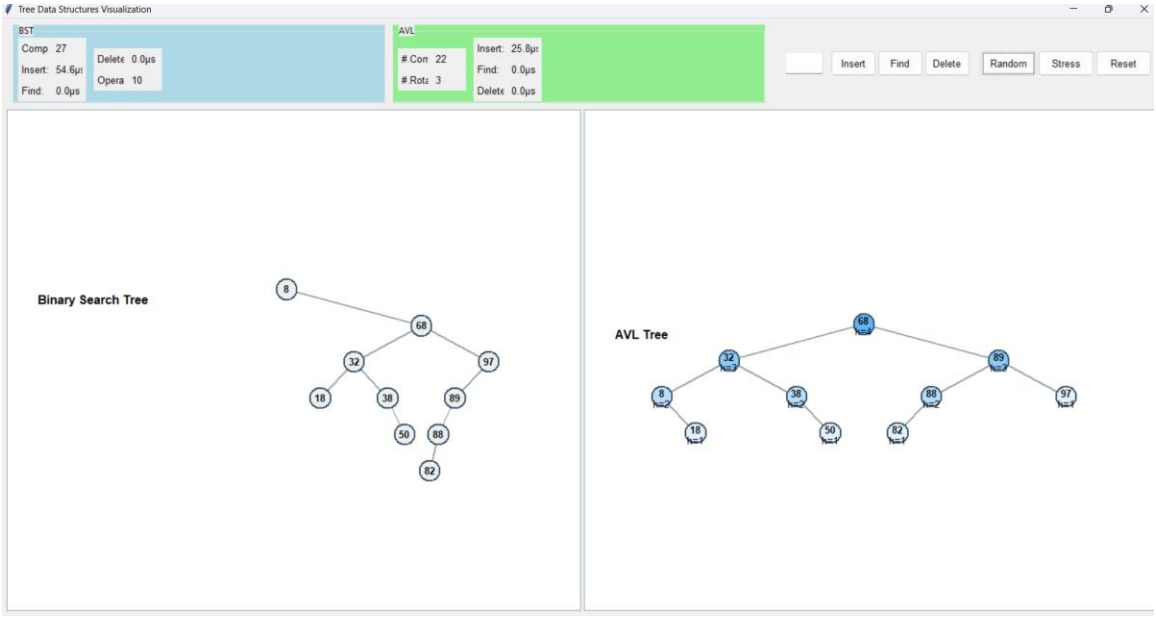
- The application uses 'Tkinter', which typically comes bundled with Python. No additional package installation is required.
- Download the source code from the provided repository and extract it to your preferred location.
- Navigate to the application directory in your terminal or command prompt.
- Launch the application by running:

```
python main.py
```

- I have also provided the windows .exe file which is in the 'dist' folder. So, you can just download and run the application for Windows

Understanding the Interface

Upon launching the Tree Comparison Visualizer, you'll be presented with a clean, intuitive interface divided into several key areas. Let's explore each component in detail.



Main Window Layout

The main window is organized into three primary sections:

- The Control Panel occupies the top portion of the window, providing easy access to all operations and controls. Here you'll find an input field for entering values, along with buttons for various tree operations.
- The Visualization Area takes up most of the window space, split into two equal panels. The left panel displays the Binary Search Tree (BST), while the right panel shows the AVL Tree. This side-by-side arrangement allows for easy comparison of how each tree type handles the same operations.
- The Performance Metrics Panel, located at the top of the window, displays real-time statistics about tree operations, including execution times, comparison counts, and for AVL trees, rotation counts.

Control Panel Features

The control panel has been designed for intuitive operation. The main input field accepts numerical values for tree operations. Adjacent to this, you'll find several operation buttons:

- The 'Insert' button adds new values to both trees simultaneously. When clicked, the value in the input field is inserted into both the BST and AVL tree, allowing you to observe how each tree handles the insertion.
- The 'Find' button initiates a search operation in both trees. When used, the application highlights the search path in each tree, making it easy to compare how different tree structures affect search efficiency.

- The 'Delete' button removes values from both trees. This operation clearly demonstrates the different approaches to maintaining tree structure after node removal.

Additional controls include the 'Random' button for inserting random values, a 'Stress' button for performance testing, and a 'Reset' button to clear both trees.

Understanding Tree Visualization:

The visualization area uses several visual cues to help understand tree structure and operations:

- Nodes in the BST are displayed in a consistent color scheme, while AVL tree nodes use a gradient coloring system based on node height. This makes it easy to understand the balance characteristics of the AVL tree briefly.
- When operations are performed, the affected paths are highlighted. Search paths are shown in yellow, with the target node (if found) highlighted in green. This makes it easy to follow the operation's progress through the tree.
- Lines connecting nodes (edges) are drawn with different thicknesses and colors depending on whether they're part of an operation's path. This helps visualize the traversal path during operations.

Basic Operations

Inserting Values

To insert a value into the trees:

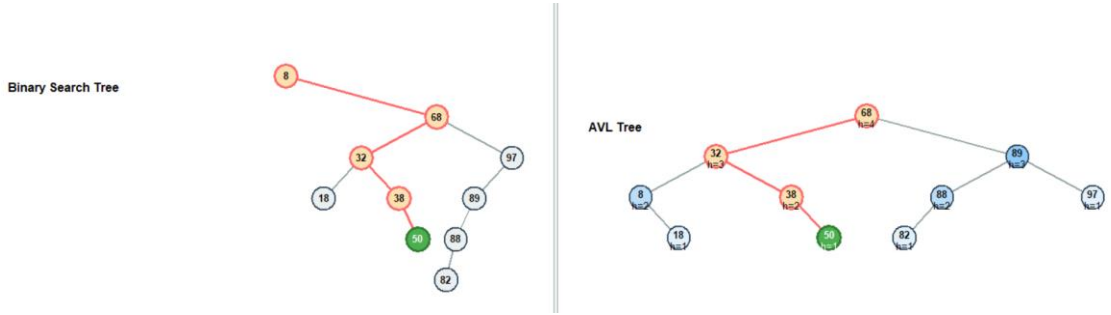
- Type a numeric value into the input field at the top of the window. The application accepts integer values only.
- Click the 'Insert' button or press Enter. The value will be inserted into both trees simultaneously.
- Observe how each tree handles the insertion. In the BST, the value will be placed according to the binary search tree property - lesser values to the left, greater values to the right. In the AVL tree, you may observe additional rotations to maintain balance.
- For example, if you insert the value 50, it becomes the root node in both trees. Following this with 30 and 70 creates a balanced structure in both trees. However, inserting several ascending values (like 80, 90, 100) will demonstrate how the AVL tree maintains balance while the BST becomes increasingly skewed.

Finding Values

The find operation demonstrates how each tree structure affects search efficiency:

- Enter the value you want to find in the input field.
- Click the 'Find' button or press ‘Ctrl+F’.

- The application will highlight the search path in both trees. You'll see the path traced from the root to either the found node or the point where the search determines the value doesn't exist.



The performance metrics panel will update to show the number of comparisons required by each tree, helping you understand the efficiency differences between balanced and potentially unbalanced trees.

Deleting Values

To remove a value from the trees:

- Enter the value to delete in the input field.
- Click the 'Delete' button or press the Delete key.
- Observe how each tree handles the removal and subsequent restructuring.

The delete operation is particularly interesting as it shows different approaches to maintaining tree structure. The BST performs a simple deletion, while the AVL tree may require additional rotations to maintain balance.

Advanced Features

Stress Testing:

The stress test feature helps understand how trees perform with larger datasets:

- Click the 'Stress' button or press ‘Ctrl+T’ to begin the test.
- The application will insert 20 random values into both trees.
- A progress window shows the test's progress.
- Watch how each tree grows and handles the insertions differently.

During the stress test, pay attention to:

- The growing height difference between the trees
- The number of rotations performed by the AVL tree
- The comparison counts for each tree
- The execution times for operations

Performance Metrics:

The performance metrics panel provides valuable insights into tree operations:

For BST:

- Number of comparisons performed
- Time taken for insertions
- Time taken for deletions
- Time taken for searches
- Total operation count

For AVL Trees:

- Number of comparisons
- Number of rotations performed
- Operation timing
- Balance statistics

All timing measurements are displayed in microseconds for precision.

Technical Details

Tree Implementation:

The application implements both tree types using a node-based structure:

The Node class serves as the fundamental building block:

```
6   class Node:
7       value: int
8       x: float = 0
9       y: float = 0
10      height: int = 1
11      left: Optional['Node'] = None
12      right: Optional['Node'] = None
```

The AVL implementation includes additional balance management:

```
def _update_height(self, node: Optional[Node]) -> None:
    if not node:
        return
    node.height = max(self._get_height(node.left),
                      self._get_height(node.right)) + 1
```

Node Positioning Algorithm

The application employs a sophisticated node positioning algorithm to ensure clear visualization of tree structures. This algorithm handles both sparse and dense trees

efficiently.
The horizontal spacing between nodes is dynamically calculated based on the tree's depth and width. Each level of the tree maintains a list of occupied positions to prevent node overlap. When positioning a new node, the algorithm first attempts to place it at an ideal position (centered below its parent), then adjusts if needed to avoid conflicts with existing nodes.
For example, when you have a balanced tree with seven nodes, the positioning ensures that:

- The root node is centered in its display area
- Level 1 nodes (children of root) are evenly spaced on either side
- Level 2 nodes maintain proper spacing without overlap
- All parent-child relationships are clearly visible through connecting lines

Performance Metrics System

The performance metrics system provides real-time insights into tree operations. The system tracks:

Time Measurements

Each operation is precisely timed using high-resolution performance counters:

```
    def start_operation(self):
        return time.perf_counter()

    Tabnine | Edit | Test | Explain | Document | Ask
    def end_operation(self, operation_name, start_time):
        end_time = time.perf_counter()
        duration = end_time - start_time
        self.operations[operation_name] += 1
        self.execution_times[operation_name].append(duration)

    Tabnine | Edit | Test | Explain | Document | Ask
    def get_avg_time(self, operation_name):
        times = self.execution_times.get(operation_name, [])
        if not times:
            return 0.0
        # Filter out any extreme outliers
        filtered_times = [t for t in times if t < 1.0]  # Remove times > 1 second
        return statistics.mean(filtered_times) if filtered_times else 0.0
```

The metrics panel displays:

- Average operation times in microseconds
- Number of comparisons performed
- Number of rotations (for AVL trees)
- Total operation counts

These measurements help users understand the performance characteristics of each tree type under different scenarios.

Common Issues and Solutions

Application Performance

If you experience slow performance:

1. Large Trees
   - Symptom: Slow updates during operations
   - Solution: Reset trees periodically when testing
   - Prevention: Use smaller datasets for demonstrations

2. Visual Glitches
   - Symptom: Overlapping nodes or unclear connections
   - Solution: Resize window to trigger layout update
   - Prevention: Maintain reasonable tree sizes

Operation Errors

1. Invalid Input
   - Symptom: Operation buttons not responding
   - Check: Ensure input is a valid integer
   - Solution: Clear input field and retry

2. Deletion Problems
   - Symptom: Node not removed as expected
   - Check: Verify value exists in tree
   - Solution: Use find operation to confirm node presence

Best Practices

For Educational Use:

1. Start Simple
   - Begin with small trees (5-7 nodes)
   - Demonstrate basic properties before complex operations
   - Use step-by-step insertion sequences
2. Demonstrate Contrasts
   - Show balanced vs unbalanced scenarios
   - Compare search paths in different tree shapes
   - Highlight performance differences

For Performance Testing:

1. Systematic Testing
   - Use consistent test patterns
   - Record and compare metrics
   - Test edge cases and typical scenarios
2. Data Management
   - Regular tree resets
   - Structured test sequences
   - Documented test cases

Keyboard Shortcuts

The application supports various keyboard shortcuts for efficient operation:

Shortcut	Action	Description
Enter	Insert	Adds current value to trees
Delete	Remove	Deletes current value
Ctrl+F	Find	Searches for current value
Ctrl+R	Random	Inserts random value
Ctrl+T	Stress Test	Runs performance test
Escape	Reset	Clears both trees

Performance Specifications

Optimal operating parameters:
- Maximum recommended nodes: 100 per tree
- Minimum node spacing: 50 pixels
- Recommended window size: 1400x900 pixels
- Maximum stress test size: 20 nodes

References:

https://docs.python.org/3/

https://tkdocs.com/

https://www.geeksforgeeks.org/complexity-different-operations-binary-tree-binary-search-tree-avl-tree/

https://github.com/kousheekc/Interactive-Visual-Binary-Search-Tree

https://github.com/nahrens007/BinarySearchTreeGui