# Project I:

# Efficiency of basic operations based on BST, AVL and Splay Trees

Group 1: Lin Zhiyu, Hu Junyu, Huang Zhenhua

2025-09-XX

## 1 Introduction

In this project, we implemented three types of binary search trees: BST, AVL Tree, and Splay Tree as well as some basic operations on them including insertion, deletion and find. We then conducted experiments to compare the efficiency of a series of these operations on the three types of trees.

## 2 Algorithm Specification

This section details the implementation of the three tree structures: Binary Search Tree (BST), AVL Tree, and Splay Tree.

### 2.1 Binary Search Tree (BST)

BST is a binary tree data structure which has the following properties: - The left subtree of a node contains only nodes with keys lesser than the node's key. - The right subtree of a node contains only nodes with keys greater than the node's key. - The left and right subtree each must also be a binary search tree. - There must be no duplicate nodes.

Our implementation is a straightforward recursive approach.

**2.1.1 BST Insertion** The insertion operation recursively finds the correct position for the new key while maintaining the BST property.

```
function BST_INSERT(node, key):
  // If the current node is null, we've found the insertion point.
  if node is null:
    return CREATE_NODE(key)
  // Recur down the tree.
  if key < node.key:
    node.left = BST_INSERT(node.left, key)
  else if key > node.key:
    node.right = BST_INSERT(node.right, key)
  // Return the (unchanged) node pointer.
  return node
```

**2.1.2 BST Deletion** Deletion handles three cases for the node to be deleted: no children, one child, or two children. For a node with two children, it is replaced by its in-order successor (the smallest key in its right subtree).

```
function BST_DELETE(node, key):
  if node is null: return null
  // Find the node to be deleted.
  if key < node.key:
    node.left = BST_DELETE(node.left, key)
  else if key > node.key:
    node.right = BST_DELETE(node.right, key)
  else: // Key found, this is the node to be deleted.
    // Case 1 & 2: Node with one or no child.
    if node.left is null:
      temp = node.right
      free(node)
      return temp
    else if node.right is null:
      temp = node.left
      free(node)
      return temp
    // Case 3: Node with two children.
    // Find the in-order successor (smallest in the right subtree).
    temp = FIND_MIN(node.right)
    // Copy the successor's key to this node.
    node.key = temp.key
    // Delete the in-order successor from the right subtree.
    node.right = BST_DELETE(node.right, temp.key)
  return node
```

### 2.2 AVL Tree

An AVL tree is a self-balancing Binary Search Tree. The heights of the two child subtrees of any node differ by at most one. If at any time they differ by more than one, rebalancing is done to restore this property.

**2.2.1 AVL Insertion** Insertion begins like a standard BST insertion. After inserting the new node, we traverse back up the tree, updating the height of each ancestor node. At each node, we check the balance factor and perform rotations if the tree has become unbalanced.

```
function AVL_INSERT(node, key):
  // 1. Perform standard BST insertion.
  if node is null:
    return CREATE_NODE(key)
  if key < node.key:
    node.left = AVL_INSERT(node.left, key)
  else if key > node.key:
    node.right = AVL_INSERT(node.right, key)
  else: // Duplicate keys are not inserted.
    return node
  // 2. Update height of the current ancestor node.
  UPDATE_HEIGHT(node)
  // 3. Get the balance factor to check if this node became unbalanced.
  balance = GET_BALANCE(node)
  // 4. If unbalanced, perform rotations.
  // Left Left Case
  if balance > 1 and key < node.left.key:
    return ROTATE_RIGHT(node)
  // Left Right Case
  if balance > 1 and key > node.left.key:
    node.left = ROTATE_LEFT(node.left)
    return ROTATE_RIGHT(node)
  // RR and RL Cases omitted.

  // 5. Return the (possibly new) root of the subtree.
  return node
```

**2.2.2 AVL Deletion** Deletion also starts as a standard BST deletion. After the node is removed, the algorithm retraces the path upwards to the root, updating heights and rebalancing each node along the path.

```
function AVL_DELETE(node, key):
  // 1. Perform standard BST delete.
  if node is null: return null
  if key < node.key:
    node.left = AVL_DELETE(node.left, key)
  else if key > node.key:
    node.right = AVL_DELETE(node.right, key)
  else:
    // Deletion logic for 0, 1, or 2 children (same as BST).
    ...

  // If the tree had only one node then return.
  if node is null: return null
  // 2. Update height of the current node.
  UPDATE_HEIGHT(node)
  // 3. Get balance factor and perform rotations to rebalance the tree.
  // The logic is similar to insertion but checks the balance of children subtrees.
  balance = GET_BALANCE(node)
  // Left Heavy
  if balance > 1:
    if GET_BALANCE(node.left) >= 0: // LL Case
      return ROTATE_RIGHT(node)
    else: // LR Case
      node.left = ROTATE_LEFT(node.left)
      return ROTATE_RIGHT(node)
  // Right Heavy
  // Similar to Left Heavy case, omitted.

  return node
```

### 2.3 Splay Tree

A Splay Tree is a self-adjusting binary search tree with the additional property that recently accessed elements are quick to access again. It performs basic operations such as insertion, look-up and removal in O(log n) amortized time. All operations on a splay tree are based on the `splay` operation.

**2.3.1 Splay Operation** The splay operation brings a target key to the root of the tree through a series of rotations. Our implementation is a recursive, top-down splay.

- **Zig:** A single rotation (left or right) when the target's parent is the root.
- **Zig-Zig:** Two rotations in the same direction (e.g., two right rotations).
- **Zig-Zag:** Two rotations in opposite directions (e.g., a left then a right rotation).

```
function SPLAY(node, key):
  // Base case: node is null or key is at root.
  if node is null or node.key == key:
    return node
  // Key lies in left subtree.
  if key < node.key:
    if node.left is null: return node // Key not found.
    // Zig-Zig (Left Left)
    if key < node.left.key:
      node.left.left = SPLAY(node.left.left, key)
      node = ROTATE_RIGHT(node)
    // Zig-Zag (Left Right)
    else if key > node.left.key:
      node.left.right = SPLAY(node.left.right, key)
      if node.left.right is not null:
        node.left = ROTATE_LEFT(node.left)
    // Final Zig rotation for the node.
    if node.left is null: return node
    else: return ROTATE_RIGHT(node)
  // Key lies in right subtree (symmetric to the left case).
  // omitted.
```

**2.3.2 Splay Insertion** To insert a key, we first perform a standard BST insertion. Then, we splay the newly inserted key to the root. This makes the new item quick to access again.

```
function SPLAY_INSERT(node, key):
  // Insert the key as in a normal BST.
  node = BST_INSERT(node, key)
  // Splay the newly inserted key to the root.
  return SPLAY(node, key)
```

**2.3.3 Splay Deletion** To delete a key, we first splay the tree on that key. If the key is found, it becomes the root. We then remove the root, which leaves two subtrees (left and right). We then join these two subtrees by splaying on the largest key in the left subtree (making it the new root of the left part) and then attaching the right subtree as its right child.

```
function SPLAY_DELETE(node, key):
  if node is null: return null
  // Splay the key to the root.
  node = SPLAY(node, key)
  // If key is not in the tree, do nothing.
  if node.key != key:
    return node
  // Key is now at the root.
  // If there is no left subtree, the right subtree becomes the new tree.
  if node.left is null:
    new_root = node.right
  else:
    // Splay the largest element in the left subtree to its root.
    new_root = SPLAY(node.left, key) // Note: key is not in left subtree, so this brings max element to root.
    // Attach the original right subtree.
    new_root.right = node.right
  free(node)
  return new_root
```

## 3 Test Design

To evaluate and compare the performance of the BST, AVL Tree, and Splay Tree implementations, we designed a series of tests to measure the total runtime of a sequence of insertions followed by a sequence of deletions. The experiment is designed to highlight the strengths and weaknesses of each data structure under different data access patterns.

### 3.1 Environment and Tools

- **Hardware**: The tests were run on a standard local machine.
- **Compiler**: The C code was compiled using `gcc`.
- **Measurement**: The `clock()` function from `<time.h>` was used to measure the CPU time for the operations.
- **Data Analysis**: The results were saved to a CSV file, which was then processed using Python program `plot.py` with the `pandas` library for creating tables and `matplotlib` for generating plots.

### 3.2 Test Variables

The experiment was structured around the following variables:

1. **Tree Type**: We tested our three implementations: BST, AVL Tree, Splay Tree
2. **Input Size (N)**: The number of nodes to be inserted and then deleted. The tests were run for `N` ranging from 100 to 1,000 in increments of 100.
3. **Test Scenarios (Data Order)**: To analyze performance under different conditions, we defined three distinct scenarios for the order of keys used in insertions and deletions:
   - **Scenario 1: Increasing Insert + Increasing Delete**
     - **Insertion Order**: Keys are inserted in ascending order ($e.g., 0, 1, 2, ..., N-1$). This creates a degenerate, chain-like tree for a standard BST.
     - **Deletion Order**: Keys are deleted in the same ascending order.
   - **Scenario 2: Increasing Insert + Reverse Delete**
     - **Insertion Order**: Keys are inserted in ascending order, as in Scenario 1.
     - **Deletion Order**: Keys are deleted in descending order ($e.g., N-1, N-2, ..., 0$).
   - **Scenario 3: Random Insert + Random Delete**
     - **Insertion Order**: Keys from 0 to $N-1$ are inserted in a random permutation.

- **Deletion Order**: Keys from 0 to $N-1$ are deleted in another random permutation. This simulates a more typical, “average-case” usage pattern.

### 3.3 Test Procedure

The overall testing process is automated by the `main` function and can be summarized as follows:

1. **Initialization**: A `result.csv` file is created with headers for storing the test outcomes.
2. **Iteration**: The program iterates through each combination of `InputSize (N)`, `TreeType` and `TestMode`.
3. **Repetition**: For each specific combination, the core test (a full cycle of insertions and deletions) is repeated 1,000 times (`REPEATITION` macro) to ensure that the measured runtime is stable and to minimize the impact of system noise.
4. **Core Test Cycle**:
   a. An empty tree is initialized.
   b. `N` keys are inserted into the tree according to the insertion order defined by the current test scenario.
   c. `N` keys are deleted from the tree according to the deletion order defined by the current test scenario.
5. **Timing**: The total CPU time for all 1,000 repetitions of the core test cycle is measured.
6. **Calculation and Storage**: The average runtime for a single test cycle is calculated in milliseconds using the formula: $runtime = (total\_time/CLOCKS\_PER\_SEC/REPETITIONS) * 1000$ The result, along with the test parameters (Tree Type, Test Scenario, N), is appended to `result.csv`.

This structured approach allows for a comprehensive comparison of how each tree structure performs under ordered, reverse-ordered, and random workloads, which is crucial for understanding their practical efficiency.

## 4 Test Results

The performance of the three tree structures was measured across the three scenarios described in the Test Design section. The results are presented below in the form of plots and summary tables.

### 4.1 Scenario 1: Increasing Insert + Increasing Delete

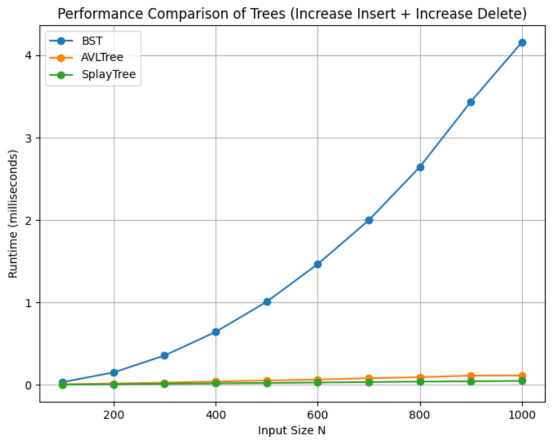
**Figure 1: Performance with ordered insertion and deletion.**

In this scenario, both BST and AVL/Splay trees show vastly different performance characteristics. The runtime for BST grows quadratically with the input size `N`, while AVL and Splay trees maintain a near-constant and highly efficient runtime.

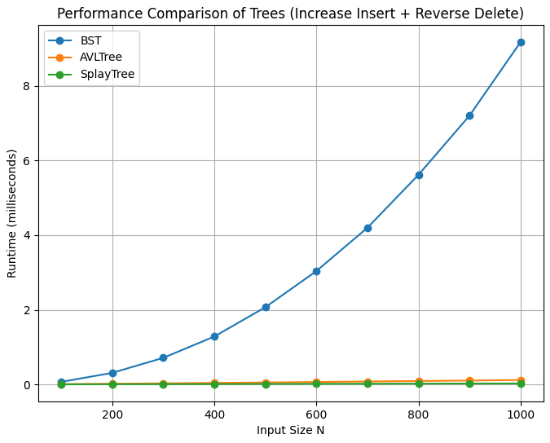
### 4.2 Scenario 2: Increasing Insert + Reverse Delete

**Figure 2: Performance with ordered insertion and reverse-ordered deletion.**

This scenario yields results very similar to the first one. The BST’s performance degrades significantly as `N` increases. In contrast, AVL and Splay trees handle this workload efficiently, with runtime scaling much more favorably.



(a) Figure 1: Increase Insert + Increase Delete



(b) Figure 2: Increase Insert + Reverse Delete

### 4.3 Scenario 3: Random Insert + Random Delete

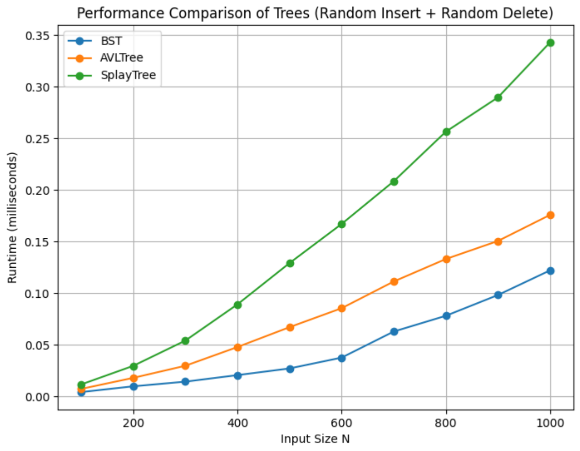


**Figure 3: Performance with random insertion and deletion.**

Under random data access, the performance roles are reversed. All three trees exhibit much better performance, with runtimes scaling gracefully. BST is the fastest, followed by the AVL Tree. The Splay Tree, while still efficient, is consistently the slowest of the three in this random workload.

### 4.4 Runtime Data Tables

The following tables, generated by `table.py`, provide the precise runtime data corresponding to the plots above.

Runtime Table for Increase Insert + Increase Delete

| N | AVLTree | BST | SplayTree |
|---|---|---|---|
| 100 | 0.008033 | 0.034503 | 0.004729 |
| 200 | 0.018009 | 0.152235 | 0.009655 |
| 300 | 0.028828 | 0.357559 | 0.015008 |
| 400 | 0.041613 | 0.642980 | 0.019678 |
| 500 | 0.054255 | 1.009675 | 0.025020 |
| 600 | 0.065791 | 1.465696 | 0.030103 |
| 700 | 0.081520 | 1.997971 | 0.035185 |

| N | AVLTree | BST | SplayTree |
|---|---|---|---|
| 800 | 0.094838 | 2.644596 | 0.040631 |
| 900 | 0.113490 | 3.435682 | 0.044752 |
| 1000 | 0.114746 | 4.157236 | 0.049617 |

Runtime Table for Increase Insert + Reverse Delete

| N | AVLTree | BST | SplayTree |
|---|---|---|---|
| 100 | 0.007814 | 0.068726 | 0.002808 |
| 200 | 0.018259 | 0.310562 | 0.006896 |
| 300 | 0.029533 | 0.713138 | 0.008049 |
| 400 | 0.041061 | 1.284329 | 0.010997 |
| 500 | 0.052658 | 2.069601 | 0.013535 |
| 600 | 0.065269 | 3.039098 | 0.016697 |
| 700 | 0.079142 | 4.199734 | 0.018919 |
| 800 | 0.092495 | 5.617617 | 0.021936 |
| 900 | 0.103658 | 7.203970 | 0.024557 |
| 1000 | 0.118520 | 9.174363 | 0.027226 |

Runtime Table for Random Insert + Random Delete

| N | AVLTree | BST | SplayTree |
|---|---|---|---|
| 100 | 0.007255 | 0.004192 | 0.011579 |
| 200 | 0.017969 | 0.009773 | 0.029609 |
| 300 | 0.029682 | 0.014284 | 0.054069 |
| 400 | 0.047829 | 0.020635 | 0.089177 |
| 500 | 0.067098 | 0.027097 | 0.129222 |
| 600 | 0.085479 | 0.037597 | 0.167130 |
| 700 | 0.111359 | 0.062865 | 0.208494 |
| 800 | 0.133131 | 0.078123 | 0.256544 |
| 900 | 0.150629 | 0.098257 | 0.289776 |
| 1000 | 0.175909 | 0.122136 | 0.343086 |

# 5 Analysis and Comments

Based on the test results, we can draw clear conclusions about the behavior and practical trade-offs of each tree structure.

### 5.1 Analysis of Ordered Data Scenarios (1 & 2)

In the scenarios with ordered insertions, the standard **BST** performs exceptionally poorly. That's because inserting keys in ascending order degenerates the BST into a linked list. Each insertion takes $O(i)$ time for the $i$-th element, leading to a total insertion time of $\sum_{i=1}^{N} O(i) = O(N^2)$. Deletions on this skewed structure are similarly inefficient. This quadratic complexity is clearly visible in the steep curve of the BST plot in Figures 1 and 2.

In conclusion, BST is entirely unsuitable for workloads involving sorted or nearly-sorted data.

The **AVL Tree** and **Splay Tree**, however, excel in these scenarios.

**AVL Tree**, by performing rotations (LL rotations in this case) after each insertion, maintains a balanced structure with a height of $O(\log N)$. This guarantees that each of the $2N$ operations (insert and delete) takes $O(\log N)$ time, resulting in a total runtime of $O(N \log N)$. This is reflected in its nearly flat, highly efficient performance curve.

**Splay Tree** utilizes the splay operation to provide excellent amortized performance. When inserting sequential data, each insertion splays the new node to the root. This series of operations effectively keeps the tree from becoming deeply unbalanced. For sequential access patterns (like increasing insertion or deletion), the total time is closer to $O(N)$, which is even better than the AVL tree's $O(N \log N)$ in theory, although in practice they appear similarly efficient at this scale.

### 5.2 Analysis of Random Data Scenario (3)

With random insertions and deletions, the performance landscape changes significantly.

**BST** performs the best in this scenario. Random insertions tend to produce a reasonably balanced tree on average, with an expected height of $O(\log N)$. Since BST does not have the overhead of balancing operations, it is faster than both AVL and Splay trees when the data itself doesn't create worst-case scenarios.

**AVL Tree** is slightly slower than the BST. While it also maintains an $O(\log N)$ height, it incurs a constant overhead for checking the balance factor at each node and performing rotations when necessary. This overhead, though small, makes it slower than a BST that is already "naturally" balanced by the random data.

**Splay Tree** is the slowest of the three for this purely random workload. The reason is the high cost of the splay operation itself. After every single insertion and deletion, the accessed node is moved to the root via a series of complex rotations. While this provides strong amortized guarantees and is beneficial for non-uniform access patterns (e.g., accessing the same few elements repeatedly), it imposes a significant constant-factor overhead on every operation in a uniform random workload.

### 5.3 Overall Conclusion

**BST** is the simplest and fastest option if you can guarantee that the input data is sufficiently random. However, it is unreliable and vulnerable to catastrophic performance degradation with ordered or semi-ordered data.

**AVL Tree** is the most reliable "all-rounder." It provides a strict $O(\log N)$ worst-case guarantee for all operations, making it a safe and predictable choice for general-purpose applications where the data pattern is unknown or potentially adversarial. Its only drawback is a minor performance overhead compared to BST in average cases.

**Splay Tree** offers excellent amortized performance and is particularly well-suited for applications with **locality of reference**, where recently accessed elements are likely to be accessed again. Its self-optimizing nature shines in non-uniform workloads. However, for purely random access, its operational overhead makes it less competitive than BST or AVL trees.

## Declaration

***We hereby declare that all the work done in this project is of our own independent effort.***