Project I:

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

Group 1: Lin Zhiyu, Hu Junyu, Huang Zhenhua

2025-09-25

1 Introduction

Binary Search Trees (BST), AVL Trees, and Splay Trees are fundamental data structures in computer science, widely used for efficient data storage, retrieval, and manipulation. Each of these tree structures has its own unique properties and performance characteristics that make them suitable for different applications.

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, find, rotation, etc. We then conducted experiments to compare the efficiency of a series of these operations (specifically insertion and deletion) 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, as well as a sketch of the main program.

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.

2.2.0 Rotations

Rotations are the key operations that maintain the balance of an AVL tree. Here are our implementations of the basic rotations (take right rotation as an example):

```
function ROTATE_RIGHT(y):
  x = y.left
  T2 = x.right
  // Perform rotation.
  x.right = y
  y.left = T2
  // Update heights.
  UPDATE_HEIGHT(y)
  UPDATE_HEIGHT(x)
  // Return new root.
  return x
```

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
```

2.4 Main Program

2.4.1 Workflow

The `main` function in `main.c` serves as the driver for the entire experiment. It systematically evaluates the performance of each tree type across all defined test scenarios and for a range of input sizes.

```
function MAIN():
  // Initialize a CSV file to store results.
  INIT_CSV("result.csv")
  // Define tree types and test scenarios.
  tree_types = ["BST", "AVLTree", "SplayTree"]
  test_modes = ["Increase+Increase", "Increase+Reverse", "Random+Random"]

  // Loop through input sizes from 1000 to 10000.
  for n from 1000 to 10000 step 1000:
    // Loop through each tree type.
    for each tree_type in tree_types:
      // Loop through each test scenario.
      for each test_mode in test_modes:
        // Generate insertion and deletion sequences.
        insert_array = GENERATE_INSERT_ORDER(n, test_mode)
        delete_array = GENERATE_DELETE_ORDER(n, test_mode)

        // Start timer.
        start_time = clock()

        // Repeat the test to get a stable average.
        for i from 1 to REPETITIONS:
          // Core test cycle: insert and then delete N keys.
          root = null
          for j from 0 to n-1:
            root = INSERT(root, insert_array[j], tree_type)
          for j from 0 to n-1:
            root = DELETE(root, delete_array[j], tree_type)

        // Stop timer.
        end_time = clock()

        // Calculate average runtime in milliseconds.
        runtime_ms = (end_time - start_time) / CLOCKS_PER_SEC / REPETITIONS * 1000

        // Store the result in the CSV file.
        STORE_RESULT("result.csv", tree_type, test_mode, n, runtime_ms)

        // Free allocated memory.
        free(insert_array)
        free(delete_array)
```

The procedure can be summarized as follows:

1. **Initialization:** Initialize a CSV file to store results.
2. **Iteration:** Use nested loops to iterate through each combination of `InputSize (N)`, `TreeType`, and `TestMode`.
3. **Data Generation:** For each specific combination, generate the appropriate insertion and deletion sequences based on the test scenario.
4. **Timing:** Use the `clock()` function to measure the time taken to perform a full cycle of insertions and deletions. The test is repeated multiple times (`REPETITION` macro) to obtain an average runtime to ensure stability and minimize the impact of system noise.
5. **Calculation and Storage:** Calculate the average runtime in milliseconds using the formula:

$runtime = (totaltime/CLOCKS\_PER\_SEC/REPETITIONS) \times 1000$

Store the results in the CSV file in a structured format.

This automated workflow ensures that all test cases are executed consistently and that the performance data is collected in a structured format for subsequent analysis with our Python scripts.

2.4.2 Test Design

To evaluate and compare the performance of three 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 was structured around the following variables:

1. **Input Size (N)**: The number of nodes to be inserted and then deleted. The tests were run for `N` ranging from 1,000 to 10,000 in increments of 1,000.
2. **Tree Type**: We tested our three implementations: BST, AVL Tree, Splay Tree.
3. **Test Scenarios (Data Order)**: To analyze performance under different conditions, we defined three distinct scenarios:
   - **Scenario 1: Increasing Insert + Increasing Delete**: Keys are inserted and deleted in ascending order ($e.g., 0, 1, ..., N-1$).

```
function MAKE_INCREASING_ORDER(n):
  arr = new Array(n)
  for i from 0 to n-1:
    arr[i] = i
  return arr
```

   - **Scenario 2: Increasing Insert + Reverse Delete**: Keys are inserted in ascending order and deleted in descending order ($e.g., N-1, N-2, ..., 0$).

```
function MAKE_DECREASING_ORDER(n):
  arr = new Array(n)
  for i from 0 to n-1:
    arr[n - 1 - i] = i
  return arr
```

   - **Scenario 3: Random Insert + Random Delete**: Keys are inserted and deleted in a random permutation to simulate an average-case usage pattern.

```
function MAKE_RANDOM_ORDER(n):
  // First, create an increasing array.
  arr = new Array(n)
  for i from 0 to n-1:
    arr[i] = i

  // Shuffle the array using Fisher-Yates algorithm.
  for i from n-1 down to 1:
    j = RANDOM_INTEGER(0, i) // Get a random index from 0 to i.
    SWAP(arr[i], arr[j])
  return arr
```

2.4.3 Makefile

We used a `Makefile` to automate the compilation and execution of the C program, as well as running the Python scripts for plotting and tabulating results.

3 Test Results

The performance of the three tree structures was measured across the three scenarios described in the Test Design section.

The following tables, generated by `table.py`, provide the precise runtime data.

Runtime Table for Increase Insert + Increase Delete

N	AVLTree	BST	SplayTree
1000	0.11556	4.15008	0.05278
2000	0.25532	17.89192	0.10166
3000	0.42094	41.76150	0.17468
4000	0.54224	75.16542	0.21156
5000	0.72848	118.97540	0.26376
6000	0.89066	171.92080	0.32904
7000	1.07318	237.43180	0.39870
8000	1.25396	311.22700	0.45160
9000	1.46202	398.96890	0.53692
10000	1.62622	494.29722	0.55066

Runtime Table for Increase Insert + Reverse Delete

N	AVLTree	BST	SplayTree
1000	0.11776	8.71784	0.02690
2000	0.24802	38.01712	0.05384
3000	0.41946	88.34712	0.08114
4000	0.57952	157.97596	0.11046
5000	0.73786	246.91108	0.13604
6000	0.90358	363.30590	0.16350
7000	1.09908	492.49904	0.18350
8000	1.22514	651.70606	0.21716
9000	1.46330	831.56928	0.24108
10000	1.67126	1034.07598	0.28262

Runtime Table for Random Insert + Random Delete

N	AVLTree	BST	SplayTree
1000	0.18036	0.12576	0.34044
2000	0.41334	0.30998	0.79180
3000	0.82726	0.52814	1.34090
4000	0.96902	0.77558	1.80462
5000	1.28400	0.98290	2.54944
6000	1.55136	1.24196	3.01268
7000	1.90750	1.52854	3.62244
8000	2.28250	1.81046	4.19604
9000	2.53008	2.04578	4.83002
10000	3.02948	2.42818	5.52356

4 Analysis and Comments

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

4.1 Space Complexity

The space complexity of these tree structures can be broken down into two main components: the space required to store the tree itself (node storage) and the auxiliary space used during operations (typically on the call stack for recursive implementations).

1. **Node Storage:** Despite the minor difference in individual node size, the total space required for all three tree types is directly proportional to the number of nodes. Therefore, the primary space complexity for storing the tree is $O(n)$ for all three implementations.
2. **Auxiliary Space (Recursion Stack):**

- **BST:** In the average case, the recursion depth is $O(\log n)$. In the worst case (a skewed tree), it becomes $O(n)$.
- **AVL Tree:** Since the tree is always balanced, the recursion depth is strictly bounded by the height, resulting in $O(\log n)$ space.
- **Splay Tree:** Similar to the BST, the recursion depth for a single splay operation can be $O(n)$ in the worst case, though on average it is $O(\log n)$.

4.2 Time Complexity and Comparison in Different Scenarios

4.2.1 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 ($e.g., 0, 1, 2, ...$) degenerates the BST into a long chain 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 plots for both scenarios, where the runtime for BST forms a steep, upward-curving line, quickly dominating the graph.

The **AVL Tree** and **Splay Tree**, however, excel in these scenarios. Their runtimes are so efficient that on the full-scale plots (see ***Figure 1(a)&(c)***), their performance curves are almost indistinguishable from the x-axis. The zoomed-in plots (***Figure 1(b)&(d)***) reveal that their runtimes grow very slowly as `N` increases, appearing as nearly flat lines. This visually confirms their efficiency and scalability in handling ordered data.

AVL Tree, by performing rotations after each insertion, maintains a balanced structure with a height of $O(\log N)$. For instance, when inserting keys `0, 1, 2`, the insertion of `2` creates an imbalance at the root `0`. This is a "Right-Right" case, which is fixed by a single Left Rotation. By consistently applying such corrections, the tree remains balanced. 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.

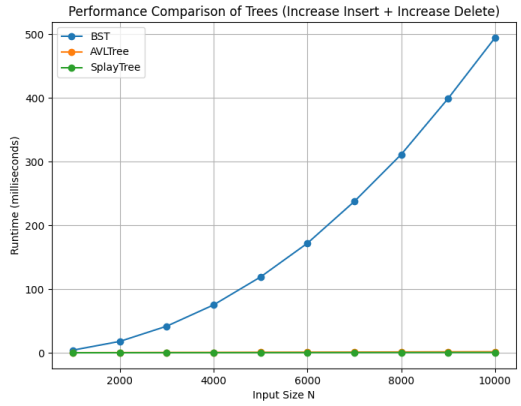
Splay Tree utilizes the splay operation to provide excellent amortized performance. When inserting sequential data, each insertion splays the new node to the root. This series of operations effectively keeps the tree from becoming deeply unbalanced. For sequential access patterns (like increasing insertion or deletion), the total time is closer to $O(N)$. The zoomed-in plots confirm this: the Splay Tree's runtime is consistently slightly lower than the AVL Tree's. This is because while the AVL tree incurs a constant overhead for balancing on every operation, the splay operation is particularly efficient for sequential access. For example, after inserting key `k`, it becomes the root. To insert `k+1`, the search starts at the root `k`, immediately goes to the right child, and finds the insertion spot. The subsequent splay on `k+1` is thus very fast. This exploitation of **locality of reference** gives the Splay Tree a slight edge in these specific scenarios.

A vertical comparison between these two scenarios also reveals interesting behaviors:

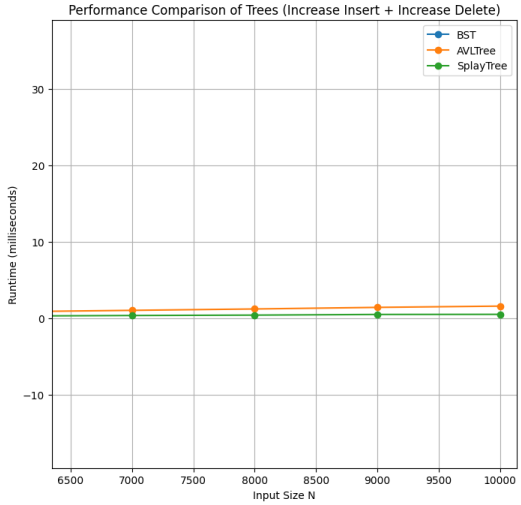
For **BST**, sequential deletion (Scenario 1) is significantly faster than reverse-order deletion (Scenario 2). In a tree built from ascending keys, the structure is a long right-leaning chain. Deleting in reverse order (e.g., `N-1`, `N-2`, …) repeatedly targets the leaf node at the end of this chain, requiring a full traversal each time, leading to a total deletion cost of $O(N^2)$. In contrast, sequential deletion (e.g., `0`, `1`, …) always removes the current root of the tree, which is a much faster $O(1)$ operation (after the node is found), resulting in a total deletion cost of only $O(N)$.

For **Splay Tree**, the opposite is true: reverse-order deletion is more efficient. This is a classic example of exploiting locality. When deleting in reverse order, after splaying and removing `k`, the next target `k-1` is very close to the new root, making the subsequent splay operation extremely fast. However, in the sequential deletion scenario, removing the root `k` and making the next smallest element `k+1` the new root creates a less optimal structure for the subsequent splay of `k+1`, resulting in slightly more work per operation on average.

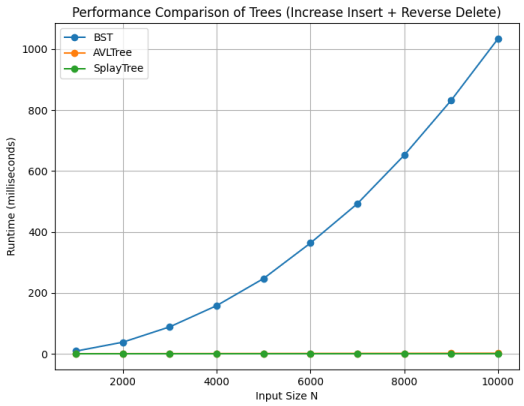
In conclusion, BST is entirely unsuitable for workloads involving sorted or nearly-sorted data. Both AVL and Splay trees handle such scenarios gracefully, with Splay trees demonstrating a slight performance advantage due to their self-adjusting nature being highly effective for sequential access patterns.



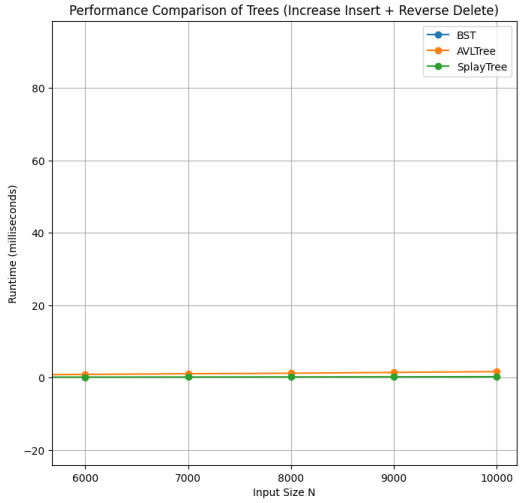
(a) Scenario 1: Full-scale View



(b) Scenario 1: Zoomed-in View



(c) Scenario 2: Full-scale View



(d) Scenario 2: Zoomed-in View

Figure 1: Performance Comparison in Ordered Data Scenarios.

4.2.2 Random Data Scenario (3)

With random insertions and deletions, the performance landscape changes significantly, as illustrated in ***Figure 2***. All three trees now appear to grow linearly with the input size `N`.

BST performs the best in this scenario. The plot shows its performance curve as the lowest of the three, indicating the fastest runtime. This is because 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. Its performance curve lies just above the BST's. 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.

Splay Tree is the slowest of the three for this purely random workload, with its performance curve being the highest on the plot. 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. This imposes a significant constant-factor overhead on every operation in a uniform random workload.

Figure 2: Performance with random insertion and deletion.

4.3 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.