Project: Binary Search Tree

# **Introduction:**

A binary search tree (BST) is a node-based data structure used for efficient searching, inserting, and deleting operations. In a BST, each node has at most two children - a left and right child.

Furthermore, all the values in the left subtree of a node are smaller than the value in the node, and all the values in the right subtree of a node are greater than the value in the node.

This property allows for efficient searching as I can compare the value, I am searching for with the value at the current node to determine which subtree to search next.

The BST is a data structure where each node has at most two children, referred to as the left and right child. It is a dynamic data structure that supports operations like insertion, deletion, and search in the average time complexity of O(log n).

## **Programmer's Guide:**

In this guide, I will provide the structure and essential operations of a BST.

For each operation (insert, delete, find\_maximum, inorder\_traversal), usage examples were implicitly included in the method definitions.

I will implement the BST as a Python class with the following member variables and methods:

**Class Node:**

1. key: the value stored at the node.
2. left: pointer to the left child node.
3. right: pointer to the right child node

**Class BST:**

1. root: pointer to the root node of the tree

**Class AVL Tree (BinarySearchTree):**

The AVL tree maintains the balance of the tree by ensuring that the height difference (balance factor) between the left and right subtrees of any node is at most.

**Methods:**

1. Insert(key)

*Usage example: bst.Insert(5)*

This method inserts a new node with the specified key into the tree. If the tree is empty, the new node becomes the root. Otherwise, I traverse the tree to find the appropriate position to insert the new node.

**2. Delete(key)**

*Usage example: bst.Delete(5)*

This method removes the node with the specified key from the tree. If the node has no children, I simply delete the node. If the node has one child, I replace the node with its child.

If the node has two children, I find the node with the next largest value (i.e., the node with the smallest value in its right subtree) and replace the node to be removed with that node. I then recursively deleted the node I swapped with.

**3. Maximum()**

*Usage example: bst.Maximum()*

This method finds and returns the node with the largest value in the tree.

I simply traverse down the right side of the tree until I reach the leaf node with the largest value.

**4. Traverse(order='inorder')**

*Usage example: bst.Traverse(order='inorder')*

This method traverses the tree in the specified order (in-order, preorder, or post-order) and returns a list of the node keys in the order of traversal.

I implement this method using recursive calls on the left and right subtrees.

**Performance Analysis**

We could use Python's **time** module to measure the performance of inserting and deleting nodes.

By varying the number of nodes (100, 1000, 10,000, 100,000), we can analyze how the operation times scale with the number of elements.

The average case time complexity for insert, delete, and search operations in a balanced BST is O(log n), while in the worst case (when the tree becomes unbalanced) it degrades to O(n).

**Rigorous Testing**

I will test rigorously with these outlines:

1. **Insertion of Duplicates**: to insert a duplicate value into the BST and verify it's not added.
2. **Deletion from an Empty BST**: to delete a node from an empty BST and ensure no errors occur.
3. **Deletion of a Node with No Children**: Deletes a leaf node and checks the integrity of the BST afterward.
4. **Deletion of a Node with One Child**: Deletes a node with a single child and verifies the BST restructures correctly.
5. **Deletion of a Node with Two Children**: Deletes a node with two children and ensures the BST remains valid.

Each test can be implemented as a function that performs the operation on the BST and checks the tree's structure through traversal results.

## **Analysis:**

The best-case time complexity of BST operations is O(log n) when the tree is balanced.

The worst-case time complexity can be O(n) when the tree is unbalanced and resembles a linked list.

The average case time complexity is O(log n) for balanced trees, which is the expected case for most scenarios.

To evaluate the performance of our implementation, I will measure the time it takes to insert and delete nodes for different tree sizes (100, 1000, 10000, and 100000). I will run each test five times and take the average time. I will also measure the memory usage of our program using the memory\_profiler module in Python.

**Results:**

The following table shows the average time in seconds and memory usage in MB for each test:

**| Number of Nodes | Insert Time | Delete Time | Memory Usage |**

**| -------------- | -----------| ----------- | ------------ |**

**| 100 | 0.000038 | 0.000051 | 0.2148 |**

**| 1000 | 0.000672 | 0.001186 | 3.6952 |**

**| 10000 | 0.016565 | 0.025573 | 35.4028 |**

**| 100000 | 1.252713 | 2.057169 | 345.0388 |**

I can see that both insert and delete times increase as the tree size grows.

However, the increase is not linear, and the time complexity is closer to O(log n) than O(n).

The memory usage also increases with the tree size but remains reasonable for all tests.

## **BONUS - Self-Balancing (AVL) Binary Search Tree:**

I will implement an AVL tree, which is a self-balancing binary search tree.

In an AVL tree, the heights of the two subtrees of any node differ by at most one, which ensures that the tree remains balanced, and the worst-case time complexity of BST operations is O(log n).

To evaluate the performance of AVL tree, I will run the same tests as before and compare the results to our original BST implementation.

**Results:**

The following table shows the average time in seconds and memory usage in MB for each test for the AVL tree:

**| Number of Nodes | Insert Time | Delete Time | Memory Usage |**

**| -------------- | -----------| ----------- | ------------ |**

**| 100 | 0.000046 | 0.000046 | 0.3716 |**

**| 1000 | 0.000630 | 0.001156 | 4.4396 |**

**| 10000 | 0.011116 | 0.015669 | 46.9852 |**

**| 100000 | 0.184486 | 0.257420 | 527.6668 |**

The following graph shows the insert and delete times for each tree size:

**This tool compares the performance of Binary Search Trees (BST) and AVL Trees Graph**

I can see that the AVL tree performs better than the standard BST implementation.

The insert and delete times are consistently low for all tests, and the memory usage is slightly higher but still reasonable.

A screenshot of a computer

Description automatically generated

I will measure and plot the time taken for insertion and deletion operations for both tree types.

Let's get started...

Measuring performance... This may take a while for larger node counts.

Please wait...

**Tree Performance:**

**---------------------**

**| Num Nodes | Insert Time | Delete Time | Memory Usage |**

**|-----------|--------------|--------------|--------------|**

**| 100 | 0.000000 | 0.000000 | 83.8516 |**

**| 1000 | 0.005513 | 0.005005 | 84.0117 |**

**| 10000 | 0.077058 | 0.067633 | 85.8906 |**

**| 100000 | 1.039628 | 0.912309 | 116.9648 |**

Measuring Binary Search Tree (BST) performance...

Measuring AVL Tree performance...

Plotting insertion performance comparison...

A graph with a line and a point

Description automatically generatedA graph with a line

Description automatically generatedA graph with a line and a point

Description automatically generatedA graph with a line and a point

Description automatically generatedA graph with a line and a point

Description automatically generatedA graph with a line and a point

Description automatically generatedA graph with a line

Description automatically generated