Colby Wirth
Professor Behrooz Mansouri
12 November 2024

An Analysis of How AVL Trees Optimize Binary Search Trees

Intro

Binary search trees make use of an implicitly ordered, recursively defined, hierarchical tree-like structure. The first node to be inserted to the tree is the tree's root. When a new node is added to the tree, if its value is less than the root, then it is inserted into the root's left subtree; if it is greater than the root, it is inserted into the root's right subtree tree. This definition holds true for all nodes in a tree. This ordered, binary structure enables insertion, deletion and search operations to be performed in $O(log_2(n))$ time complexity because half of the tree's possible paths are removed with each iteration.

However, the performance of these algorithms become degraded with frequent insertions and deletions to the tree. By definition, nodes are only added as leaves, and repeated insertions may lead to an imbalanced or skewed tree. Skewing can also result from deletions, as order is preserved without maintaining balance. These skewed branches often resemble a linked list, rather than the balanced tree. As a result, such operations are slowed, often approaching a time complexity of $O(n)$. This effectively degrades the advantages that the binary search tree has over the linked list, as the linked list generally performs these operations with a time complexity of $O(n)$. A tree with these skewed branches is said to be a *degenerate tree*.

The AVL tree addresses this issue by maintaining balance through additional operations performed during insertions and deletions. Each node in the AVL tree is defined with a new field called "balance factor" that maintains an integer representation of the difference between the heights of its left and right subtrees. This field is maintained, and updated with each insertion and deletion. When a tree is considered unbalanced, a set of *rotation* operations are performed to rebalance the tree.

Main Content

Rotation Operations

A rotation operation will occur when a root has an unbalanced subtree. This is defined as when one subtree has a depth that is two nodes larger than its sibling. Each node is assigned a balance factor, after each iteration. These balance factors are numerically represented as:

-2: A node's left subtree's height is 2 nodes deeper than its right subtree.
-1: A node's left subtree's height is 1 node deeper than its right subtree.
0: A node's subtrees are of equal height.

1: A node's right subtree's height is 1 node deeper than its left subtree.
2: A node's right subtree's height is 2 nodes deeper than its left subtree.

Conditions for Rotation Operations

A rotation operation is triggered on a node when its balance factor is 2 or -2. This will always occur after insertion or deletion to ensure proper balance. There are four rotation operations that can occur based on a node's balance factor:

Left Rotation: Parent node balance factor: 2, left child balance factor: 1 or 0

Right Rotation: Parent node balance factor: -2, right child balance factor: -1 or 0

Left-Right Rotation: Parent node balance factor: -2, right child balance factor: 1

Right-Left Rotation: Parent node balance factor: 2, left child balance factor: -1

Performances of AVL Tree vs Binary Search Tree:

Dataset and Methodology Overview

The node's data type used in this experiment is a custom Song object. A Song object's natural ordering is defined by its view-count field, called "views", an integer. The dataset contains six genres which each song can be classified into. Each genre is a mutually disjoint subset with 480,000 elements. For each of the following tasks, an AVL and a Binary Search Tree is built for each of the six genres.

Analysis Part 1: Building an Online Ordered Tree

Both trees utilize a modified recursive pre-order traversal algorithm to find the insertion leaf. This operates *in* $O(log_2(n))$. However the AVL tree calls two additional helper methods involved with rebalancing the tree. These operations are conducted in constant time, $O(k)$, where k is the number of additional operations associated rotating subsets and updating balance factors. Thus the time complexity of the AVL tree's insertion method is $O(log_2(n)+k)$. This simplifies to $O(log_2(n))$, but the additional "k" operations are noticeable in runtime.

By definition, a perfectly balanced AVL Binary Search Tree has height of $\lceil \log_2(n) \rceil$. With all of the AVL trees, the heights are $\lceil \log_2(480{,}000) \rceil = 19$. In comparison, the Binary Search Trees do not maintain balance, as their heights ranged from 119 to 811 nodes for five of the six trees. In these five instances, the binary search trees executed online building ~ 1% faster than the AVL trees. With each case, the skewness of the binary search trees does not degrade the performance enough to be beat by the AVL's computationally expensive optimizations.

However with the sixth instance, the Binary Search Tree "misc" genre is outperformed by its paired AVL tree. With runtimes of 5848 milliseconds and 4256 milliseconds, respectively, the Binary Search Tree was constructed with all nodes by 37% slower than the AVL Tree. This is consequence of the denigration of the Binary Search Tree with heavily skewed insertions. Its height was 4744 nodes, while the AVL tree maintains a balanced height of 19.

Genre	Left	Right	Left-Right	Right-Left	Height	Build Time (milliseconds)
pop	113279	1468	8654	1468	19	4352
rock	79288	2709	9411	2709	19	4094
rap	107295	1447	9572	1557	19	4069
rb	104481	1775	9544	1775	19	4094
misc	128334	842	10154	842	19	4256
country	117367	1164	9779	1164	19	4058

Building an Online-Ordered AVL Tree Tree with 480,000 elements

Genre	Height	Build Time (milliseconds)
pop	499	4397
rock	119	4035
rap	575	4205
rb	581	4142
misc	4744	5848
country	811	4171

Building an Online-Ordered Binary Search Tree Tree with 480,000 elements

Analysis Part 2: Searching Algorithms

In this experiment, a modified in-order traversal searching algorithm was utilized across all cases for both types of trees. In total, five searches were made per tree. Each of the cases were selected in order to capture most general and edge cases of the searching algorithm. The average time of the five searches was calculated and analyzed below. The five cases used for searching were derived from the entire set: i. minimum value, ii. maximum value, iii. mean value, iv. out of bounds with low value, v. randomly generated value.

The AVL trees runtime outperformed the binary search trees with the searching algorithm by factors ranging from 380 to 1820. The discrepancies between the runtimes is due to the unbalanced nature of the binary search tree. The long chains of nodes found in these degenerate binary search trees force the algorithm to traverse thousands of nodes with each iteration. These chains of nodes degrade the logarithmic time complexity of

the searching algorithm, and the time complexity of the algorithm approaches *O(n)*.

In comparison the AVL trees maintain balance, therefore the logarithmic time complexity is maintained for all searching operations. This can be shown with the equation for the height, *h* of a balanced tree, *T*: $h(T) = \lceil \log_2(n) \rceil = \lceil \log_2(480{,}000) \rceil = 19$. This proves that the AVL tree will traverse at most 19 nodes with every search.

Genre	AVL avg. Search Time (microseconds)	BST avg. Search Time (microseconds)
pop	10	3823
rock	10	3959
rap	10	3614
rb	10	3617
misc	2	3684
country	3	4020

Searching Algorithms Performances for AVL and Binary Search Trees

Conclusion

In essence, an AVL tree is an optimized Binary Search Tree. As demonstrated in Analysis Part 1, the AVL tree pays a marginal cost in runtime to balance its branches with rotation operations. However, as a binary search tree becomes more skewed, the advantages of the AVL tree in insertion become evident. In this case, the "misc:" binary search tree became heavily skewed with a height of 4744. Compared to the AVL with a balanced height of 19, the "misc" binary search tree performed insertion insertion operations 37% slower over 480,000 elements.

The AVL tree's advantages are not just with insertion, as the balanced structure of the tree allows it to perform searching operations magnitudes faster than the binary search tree. As shown in Analysis Part 2, The AVL tree performed searching 380 to 1820 times faster than the binary search tree. The AVL tree would never traverse more than $\lceil \log_2(n) \rceil$ nodes, as it always maintains a height of that magnitude. In comparison, the heavily skewed binary search trees created long chains of nodes

that resembled a linked list which degraded the time complexity of the searching operations towards *O(n)*.

In conclusion, the additional code, and marginal additional run time used in building and maintaining the structure of an AVL prevents the tree from becoming skewed. This guarantees that all searching and traversing related operations maintain *O(log(n))* time complexity. In comparison, the unbalancing of a binary search tree from frequent insertions and deletions degrades significantly degrades the runtime of all operations.