Briefing Document: AVL Trees
Source: Excerpts from "ICS 46 Spring 2022, Notes and Examples_ AVL Trees.pdf"

Date: October 26, 2023

Prepared For: [Intended Audience - e.g., Software Development Team, Algorithm Study Group]

Prepared By: Gemini AI

Subject: Review of AVL Trees: Concepts, Motivation, and Properties

This briefing document summarizes the key concepts and properties of AVL trees as presented in the provided excerpts from ICS 46 Spring 2022 course materials. The document outlines the motivation behind balanced binary search trees, the limitations of aiming for perfect or complete binary trees, and how AVL trees offer a practical compromise by maintaining near-balance through specific properties and rotation operations, ultimately ensuring efficient performance for fundamental tree operations.

1. The Importance of Binary Search Tree Balancing:

The document begins by highlighting the significant impact of a binary search tree's shape, specifically its height, on its performance. While the binary search tree property dictates the arrangement of keys (smaller to the left, larger to the right), it imposes no restrictions on the tree's structure. This can lead to drastically different performance characteristics.

"We've seen previously that the performance characteristics of binary search trees can vary rather wildly, and that they're mainly dependent on the shape of the tree, with the height of the tree being the key determining factor."

The document illustrates this with the example of a "perfect binary tree" (minimal height) and a "degenerate tree" (maximal height, resembling a linked list) both containing the same keys.

"Yet, while both of these are legal, one is better than the other, because the height of the first tree (called a perfect binary tree) is smaller than the height of the second (called a degenerate tree). These two shapes represent the two extremes — the best and worst possible shapes for a binary search tree containing seven keys."

The danger of degenerate trees is emphasized, noting that they can arise in common scenarios like inserting sequentially ordered keys.

2. Problems with Degenerate Trees:

The document clearly outlines the performance drawbacks of degenerate binary search trees:

Lookup Time: Lookups in a degenerate tree take O(n) time in the worst case, as it might be necessary to traverse every node.
"Every time you perform a lookup in a degenerate binary search tree, it will take O(n) time, because it's possible that you'll have to reach every node in the tree before you're done."
Memory Usage (Recursive Lookup): Recursive lookups in a degenerate tree can consume O(n) memory due to the potential for a stack frame for each node.
"If you implement your lookup recursively, you might also be using O(n) memory, too, as you might end up with as many as n frames on your run-time stack — one for every recursive call."
Construction Time: Building a degenerate tree by inserting ordered keys takes $\Theta(n^2)$ time.
"The total number of steps it would take to add n keys would be determined by the sum 1 + 2 + 3 + ... + n. This sum, which we'll see several times throughout this course, is equal to n(n + 1) / 2. So, the total number of steps to build the entire tree would be $\Theta(n^2)$."
These inefficiencies highlight the critical need for mechanisms to control the shape and maintain a more balanced structure in binary search trees, especially when dealing with a large number of keys.

3. The Ideal but Impractical Goal of Perfection:

The document discusses the idea of aiming for a "perfect binary tree" where every level is completely filled. While this would guarantee a height of $\Theta(\log n)$ and thus optimal performance, it is often unattainable because not every number of nodes allows for a perfect binary tree structure.

"The perfect binary trees pictured above have 1, 3, 7, and 15 nodes respectively, and are the only possible perfect shapes for binary trees with that number of nodes. The problem, though, lies in the fact that there is no valid perfect binary tree with 2 nodes, or with 4, 5, 6, 8, 9, 10, 11, 12, 13, or 14 nodes."

4. The More Realistic Concept of Complete Binary Trees:

A "complete binary tree" is introduced as a more flexible notion of balance. It requires all levels to be fully filled except possibly the last level, where nodes are filled from left to right. Complete binary trees also maintain a logarithmic height ($\Theta(\log n)$).

"We've seen that the height of a perfect binary tree is $\Theta(\log n)$. It's not a stretch to see that the height a complete binary tree will be $\Theta(\log n)$, as well, and we'll accept that via our intuition for now and proceed. All in all, a complete binary tree would be a great goal for us to attain: If we could keep the shape of our binary search trees complete, we would always have binary search trees with height $\Theta(\log n)$."

However, the document points out the significant cost of maintaining completeness after insertions. A single insertion in a complete binary tree might require moving a large number of nodes, leading to a worst-case time complexity of $\Omega(n)$ for re-balancing.

"Every key in the tree had to move! So, no matter what algorithm we used, we would still have to move every key. If there are n keys in the tree, that would take Ω(n) time..."

This high re-balancing cost makes strictly maintaining completeness impractical for efficient operations.

5. Defining a "Good" Balance Condition:

The document establishes the criteria for a "good" balance condition in a binary search tree:

Logarithmic Height: The height of the tree must be Θ(log n) to ensure logarithmic time complexity for lookups.
Logarithmic Re-balancing Cost: The time required to re-balance the tree after insertions and removals should not exceed O(log n), so the re-balancing cost doesn't negate the benefits of the logarithmic height.
"A 'good' balance condition has two properties: The height of a binary search tree meeting the condition is Θ(log n). It takes O(log n) time to re-balance the tree on insertions and removals."

6. Introduction to AVL Trees as a Compromise:

AVL trees are presented as a well-known approach that achieves a "good" balance condition by being "nearly balanced." They guarantee logarithmic height with logarithmic re-balancing costs.

"There are a few well-known approaches for maintaining binary search trees in a state of near-balance that meets our notion of a 'good' balance condition. One of them is called an AVL tree, which we'll explore here."

"AVL trees are what you might called 'nearly balanced' binary search trees. While they certainly aren't as perfectly-balanced as possible, they nonetheless achieve the goals we've decided on: maintaining logarithmic height at no more than logarithmic cost."

7. The AVL Property:

The core of AVL tree balancing lies in the AVL property:

"We say that a node in a binary search tree has the AVL property if the heights of its left and right subtrees differ by no more than 1."

The document clarifies the definition of height, stating that a tree with only a root has a height of 0, and an empty tree has a height of -1. This convention helps in consistently applying the AVL property.

An AVL tree is then defined as a binary search tree where all nodes satisfy the AVL property. The document emphasizes that AVL balance is a precise definition, not just a visual assessment.

"An AVL tree is a binary search tree in which all nodes have the AVL property."

8. Maintaining Balance with Rotations:

AVL trees maintain their balance after insertions and removals through a process of detecting imbalances and correcting them using rotations. These rotations rearrange the tree structure while preserving the binary search tree property. The document introduces four types of rotations:

LL Rotation (Left-Left): Corrects an imbalance where the left subtree of the left child is too heavy.
RR Rotation (Right-Right): Corrects an imbalance where the right subtree of the right child is too heavy (mirror of LL).
LR Rotation (Left-Right): Corrects an imbalance where the right subtree of the left child is too heavy.
RL Rotation (Right-Left): Corrects an imbalance where the left subtree of the right child is too heavy (mirror of LR).
The document provides diagrams and explanations of how these rotations work by adjusting pointers between nodes and subtrees. Importantly, each rotation takes Θ(1) time as it involves a constant number of pointer updates.

"Performing this rotation would be a simple matter of adjusting a few pointers — notably, a constant number of pointers, no matter how many nodes are in the tree, which means that this rotation would run in Θ(1) time."

9. The Insertion Algorithm in AVL Trees:

The insertion process in an AVL tree follows these steps:

Perform a standard binary search tree insertion.
Traverse back up the tree from the inserted node.
At each node, compare the heights of its left and right subtrees.
If the height difference exceeds 1 (violating the AVL property), perform a rotation to restore balance. The type of rotation (LL, RR, LR, or RL) is determined by the path taken (left or right links) from the imbalanced node down to the newly inserted node.
The document highlights the importance of storing the height of each node to efficiently compare subtree heights during the upward traversal.

"The solution to this problem is for each node to store its height (i.e., the height of the subtree rooted there). This can be cheaply updated after every insertion or removal as you unwind the recursion."

The document provides an example of an insertion causing an imbalance and demonstrates how an LR rotation corrects it. It emphasizes that a single rotation is sufficient to fix an imbalance caused by an insertion.

10. The Removal Algorithm in AVL Trees:

The removal process is similar to insertion:

Perform a standard binary search tree removal.
Traverse back up the tree from the location of the removed node.
At each node, check for AVL property violations and perform rotations as needed.
A key difference from insertion is that removals might require more than one rotation to rebalance the tree. However, these rotations are still limited to the path back up to the root, ensuring a logarithmic number of rotations in the worst case.

"Removals are somewhat similar to insertions, in the sense that you would start with the usual binary search tree removal algorithm, then find and correct imbalances while the recursion unwinds. The key difference is that removals can require more than one rotation to correct imbalances, but will still only require rotations on the path back up to the root from where the removal occurred — so, generally, O(log n) rotations."

11. Asymptotic Analysis of AVL Trees:

The crucial aspect of AVL trees is their height. The document poses the question:

"The key question here is What is the height of an AVL tree with n nodes? If the answer is Θ(log n), then we can be certain that lookups, insertions, and removals will take O(log n) time."

The reasoning for logarithmic time complexity is provided:

Lookups: Similar to standard binary search trees, the time complexity is proportional to the height, thus O(log n).
Insertions and Removals: These operations involve traversing a path down the tree (O(log n)) and potentially traversing back up while performing a constant number of Θ(1) rotations at each step. Since the height is logarithmic, the overall time complexity is O(log n).
12. Proof of Logarithmic Height (Optional):

The document includes an optional section providing a lower bound for the minimum number of nodes in an AVL tree of height h, denoted as M(h). The recurrence relation for M(h) is given:

$M(0) = 1$
$M(1) = 2$
$M(h) = 1 + M(h - 1) + M(h - 2)$ for $h \geq 2$
Through a series of inequalities and repeated substitution, it is shown that:

"$M(h) \geq 2h/2$"

This lower bound on the number of nodes for a given height is then used to derive an upper bound on the height for a given number of nodes (n):

"$h \leq 2 \log_2 n$"

This result confirms that the height of an AVL tree with n nodes is indeed $\Theta(\log n)$. The document notes that the actual upper bound is tighter (around $1.44 \log_2 n$), but the asymptotic result remains the same.

"So, ultimately, we see that the height of an AVL tree with n nodes is $\Theta(\log n)$."

Conclusion:

The provided excerpts effectively explain the motivation for balanced binary search trees, the limitations of strictly maintaining perfect or complete balance, and how AVL trees provide a practical and efficient solution. By enforcing the AVL property and utilizing rotations for rebalancing, AVL trees guarantee a logarithmic height, ensuring that fundamental operations like lookups, insertions, and removals can be performed in $O(\log n)$ time, making them a valuable data structure for applications requiring efficient dynamic sets.