CS 520 Research Paper 2 – AVL Trees

Caroline Harrison

June 12, 2020

**Abstract:**

Binary search trees are a commonly used abstract data structure built for quick data storage and sorting purposes. In a binary search tree, the height and order of the overall structure is determined by the order of node insertions in the tree during construction, and if the added nodes create an unbalanced tree the height will grow and the performance time to search for a node within that tree can deteriorate quickly. The average runtime to search through a tree for a node is directly related to the height of that tree, so building a balanced tree with the shortest height at any number of nodes is the best solution to keep the runtime of search operations efficient. However, maintaining a perfectly balanced tree is often very expensive, and may require complete structural overhauls of the tree as new nodes are inserted. The compromise between the tree construction and search time is a self-balancing binary search tree, which perform alterations on the tree when necessary in order to reduce the height of the tree when it becomes unbalanced.

Several algorithms have been developed as methods of keeping a tree balanced during construction in order to reduce the inefficiencies of working with an unbalanced tree, the first of which was the AVL self-balancing tree. The AVL algorithm was proposed in 1962 by Georgy Adelson-Velsky and Evgenii Landis (for whom the algorithm is named after) as a method for minimizing the number of operations required to maneuver through a binary search tree[1]. In an AVL tree, when the structure becomes unbalanced, or critical, the subtree where the violation occurred goes through a rebalancing process where the nodes are rotated to create a new subtree that absorbs the excess height and returns the tree to a balanced state.

**AVL Description:**

The AVL algorithm was presented as a way to organize information elements in memory in such a way that the number of operations required to find any element is kept to a minimum[1]. The structure proposed was an ‘almost’ perfectly balanced tree, where the height difference between any subtree can be no greater than one. During the construction of an AVL binary search tree, after each node is inserted the tree is assessed to determine whether it has become unbalanced and, if necessary, rebalances the nodes to absorb the excess branches and returns to a balanced state. The structure will take more time to construct but reduce the maximum number of operations required to search for elements in memory [2].

The process of inserting a new value into an AVL tree begins the same way as inserting a new value into any binary search tree, traversing down the tree to find a vacant position by comparing the input value to each node along its path and moving left or right down each branch based on whether the input value is less than or greater than the node on its path. Once a vacant position has been found and filled by the input value, the surrounding subtree is evaluated to determine whether the inclusion of this new node has caused it to become critical[2]. The subtree is considered to critical when its balance factor, or height of its right subtree minus the height of its left subtree, is greater than 1 or less than -1. If the process of inserting the new value does not cause the subtree to become too unbalanced, no further steps are taken. When a subtree is critical, it must absorb the excess through the rebalancing process[3].

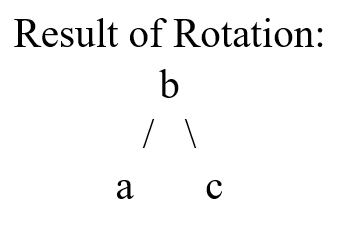
To rebalance a subtree, tree rotation operations are used to change the subtree structure without disrupting the order or validity of that structure. There are four main tree rotations methods used to restore balance in an AVL tree, each corresponding with a possible scenario for how a subtree may become unbalanced.

Figure 1: Rotation Requirements for Unbalanced Subtrees



For a left rotation, the node *b* becomes the new root, setting *a* as its new left child, and keeping *c* as its right child. A right rotation follows the similar steps in the opposite direction, setting *b* as the new root node, assigning *c* as the new right child, and keeping *a* as its left child. In some cases, a single rotation is not enough, in the case of the left-right and right-left rotations the bottom two nodes must be rotated before rotating the whole subtree. Whichever rotation is required to rebalance the nodes in the above figure, the result of the rotations is the same.

Figure 2: Rebalanced Subtree



**Performance Measurements:**

A perfectly balanced BST maintains all operations running in O(log *n*) time, where *n* is the number of nodes in the tree. As the tree becomes more unbalanced, the search time increases towards a limit of O(*n*) per search[]. Keeping a tree close to perfectly balanced will prevent the runtime from deteriorating, resulting in the worst-case runtime to remain close to equal for the average runtime for any node in a balanced binary tree.

The efficiency of the AVL tree results from keeping the tree close to balanced, where the height of the tree with *n* nodes is kept at roughly 1.44 \* log *n*[4]. The AVL tree will also see stronger performances in comparison to regular BST tree’s when working with partially sorted lists[4]. Inserting nodes into a regular BST from a sorted, or even partially sorted, array will quickly become unbalanced. Consider inserting *n* pre-sorted values into an empty binary search tree. The height of the tree becomes *n*-1, and the runtime to add each additional value increases the height creating the most unbalanced tree possible[4]. Searching for a node in an unbalanced tree is so inefficient because comparisons do not significantly reduce the number of comparisons still needed to find the desired node. In a perfectly balanced tree, each comparison would eliminate half of the remaining potential comparisons, whereas in a perfectly unbalanced tree, the first comparison would not eliminate any potential comparisons.

AVL trees are often applied to record indexing in databases to improve search efficiencies. Databases require more frequent data lookup than insertion and deletion, so the upfront building costs are offset by the improved search times[5].

**Implementation of AVL in Python**:

My implementation of an AVL tree algorithm in Python is broken out into three classes, the Node class, the Binary Search Tree class, and AVL algorithm class, where AVL is derived from the BST base class. Having AVL inherit from the BST class assists in highlighting the similarities and key differences between the two structures. The Node class creates instances of nodes used for both the tree structures, and defines the value, left child, right child, and height attributes for each node.

The Binary Search Tree class defines the functions getHeight, getPreorderTraversal, and insertValue. The getHeight functions returns the height of the tree from the node parameter given, the getPreorderTraversal function returns a list of node values in the tree using the preorder (left child, node, right child) order to traverse the tree. The insertValue function creates a node from the inputted value and places that node in the tree. Below shows the BST insertValue function:

class BST():

    def insertValue(self, root, insert\_value):

        if root == None:

            return Node(insert\_value)

        elif insert\_value < root.value:

            root.left = self.insertValue(root.left, insert\_value)

        else:

            root.right = self.insertValue(root.right, insert\_value)

        root.height = 1 + self.getMaxHeight(root)

        return root

The derived AVL class instantiates the base BST class and defines the methods required for node rebalancing when the tree becomes critical, including rotateRight and rotateLeft. As the names suggest, these methods define the process of rotating the nodes of a subtree in order to regain height balance within that subtree. The AVL class also defines a new insertValue method that overrides the one defined in the BST class, which extends the BST insertValue method to include the AVL algorithm. This AVL override calls the base method before performing balancing checks and necessary subtree rotations. The implementation for this AVL class is below:

class AVL(BST):

    #rotate functions

    def rotateRight(self, n):

        previous\_left = n.left

        new\_left = previous\_left.right

        previous\_left.right = n

        n.left = new\_left

        n.height = 1 + self.getMaxHeight(n)

        previous\_left.height = 1 + self.getMaxHeight(previous\_left)

        return previous\_left

    def rotateLeft(self, n):

        previous\_right = n.right

        new\_right = previous\_right.left

        previous\_right.left = n

        n.right = new\_right

        n.height = 1 + self.getMaxHeight(n)

        previous\_right.height = 1 + self.getMaxHeight(previous\_right)

        return previous\_right

    def insertValue(self, root, insert\_value):

        #insert value as if normal BST

        root = super().insertValue(root, insert\_value)

        #AVL algorithm

        if (self.getHeight(root.left) - self.getHeight(root.right)) > 1:

            #if true left-right, else right

            if insert\_value > root.left.value:

                root.left = self.rotateLeft(root.left)

            return self.rotateRight(root)

        if (self.getHeight(root.left) - self.getHeight(root.right) )< -1:

            #if true, right-left, else left

            if insert\_value < root.right.value:

                root.right = self.rotateRight(root.right)

            return self.rotateLeft(root)

        return root

**Tests:**

Tests were performed on both the regular binary search tree and the AVL binary search tree, with the same input array for both types of trees in order to maintain consistent results. The first tests used a truly randomly sorted array as the input for the tree, and the second tests used pre-sorted arrays. Comparisons were made on the time (in seconds) required to build the tree, the average number of comparisons required to find each value in the tree, the maximum number of comparisons for searching a tree, and the height of the tree. The purpose of these tests is to visualize the tradeoffs between the time to construct the two types of trees, and the number of operations needed to search for nodes within the tree once it has been built.

Table 1. Random Array Test Results



The results of these random input array tests clearly show the tradeoffs made between the construction time and required operations to search for elements in the tree, with the AVL construction time growing at a faster rate than the regular binary search tree time. However, the average number of comparisons to search for each element in a regular BST was much higher than in the AVL, due to the high worst-case scenario searches driving up the mean runtime. It can also be observed that the worst-case number of comparisons required to find an element in the AVL tree remains equal to that of the average number of comparisons in the regular BST.

The tests performed on the sorted arrays show even more extreme differences between the efficiencies of a regular BST and an AVL tree. The input arrays for the second test only reaches a size of 2,048 because Python has a default limit of 1000 recursive calls allowed in a function. This limit created caused the inputValue method to throw a recursion error, as the tree became so unbalanced that the number of times the recursive call was needed to find a free spot for the node exceeded Python’s limit.

Table 2. Sorted Array Test Results



The results from this second test show just how much the performance of a regular BST is negatively impacted by using a sorted input array, while the performance of the AVL tree remained the same. Even the construction time to build a regular BST tree deteriorated faster than the AVL tree. This happened because of the cost of inserting a new value into an unbalanced regular BST became greater than the sum cost of inserting a new value into the almost balanced AVL tree plus the cost of rebalancing that tree when it became critical.

**Conclusion:**

The AVL algorithm was designed to reduce the number of operations required to search for any node in a binary search tree by building a tree that is as balanced as possible. In order to maintain this status, node insertions are followed by an assessment to determine whether the new node has caused the tree to become critical, and then rotates the nodes in that subtree to regain a state of balance. While this process increases the time required to build the tree, searching for nodes within the tree becomes a more efficient process, and there is no deterioration in runtime if the tree becomes unbalanced. The efficient search operation runtimes make AVL trees particularly useful in situations of frequent lookups, rather than insertion and deletion, and are often applied to database record indexing.

**References:**

1. Adelson-Velskii, G.; E. M. Landis (1962). "An algorithm for the organization of information". Proceedings of the USSR Academy of Sciences 146: 263–266. (Russian) English translation by Myron J. Ricci in Soviet Math. Doklady, 3:1259–1263, 1962.

2. Foster, Caxton C. "Information retrieval: information storage and retrieval using AVL trees." *Proceedings of the 1965 20th national conference*. 1965.

3. Foster, Caxton C. "A generalization of AVL trees." *Communications of the ACM* 16.8 (1973): 513-517.

4. Pfaff, Ben. "Performance analysis of BSTs in system software." *ACM SIGMETRICS Performance Evaluation Review* 32.1 (2004): 410-411.

5. Lehman, Tobin J., and Michael J. Carey. *A study of index structures for main memory database management systems*. University of Wisconsin-Madison Department of Computer Sciences, 1985.