AVL Trees

Ashwin asokan

April 30, 2016

A case study on AVL, a type of height balanced trees.

Contents

Chapter 1

Introduction

A binary search tree is deemed self balancing when it attempts to keep its height, or the number of levels of nodes beneath the root, as small as possible all the time. They serve as efficient implementations for wide variety of abstract data structures with some sort of ordering such as associative arrays, priority queues and sets. Over the years there have been different approaches proposed to achieve balanced tree structure and get better performance.

AVL Tree is one of the oldest approaches to self balancing tree with height as the measure. It is named after its inventors (Georgy Adelson-Velsky and Evgenii Landis' tree) and one of the first such data structure being proposed.

In an AVL tree, the key property is 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. As a result Lookup, insertion, and deletion all take O(log n) time in both the average and worst cases, where n is the number of nodes in the tree prior to the operation. But as a side effect insertions and deletions may require the tree to be rebalanced by one or more tree rotations.

The key contribution of this report is a comprehensive study of AVL tree particularly it's balancing act, its simplicity, correctness, best, average and worst case performance under simulated inputs.

Rest of this chapter tries to go through trees and its self balancing act in general as a build up for the whole report, second chapter describes the AVL property, its construction & divulges in its impact on key binary search tree operations highlighting the differences from normal binary search tree. Third chapter builds upon the previous one highlighting the performance gains by adhering to the AVL property, best , worst and average case characteristic of inputs, some numerical results to back up the claims. In the fourth chapter we try to question the common claims made by AVL and establish their correctness. Fifth chapter concludes the report comparing the results of

AVL with other contemporary approaches & advice on its usage.

1.1 Self balancing binary search trees

Binary search trees (BST), sometimes called ordered or sorted binary trees, are a particular type of data structures that store data (could be numbers, names etc.) in memory which facilitates fast lookup, addition and removal of items. They can be used to implement either dynamic sets of items, or lookup tables that allow finding an item by its key (e.g., finding the phone number of a person by name).

Binary search trees achieve their performance by keeping their keys in sorted order, so that lookup and other operations can use binary search: when looking for a key in a tree, they traverse the tree from root to leaf, making comparisons to keys stored in the nodes of the tree and making a binary decision, based on the comparison to continue searching either in the left or on the right sub tree. On average, this means that each comparison allows the operations to skip about half of the tree, so that each lookup, insertion or deletion takes time proportional to the logarithm of the number of items stored in the tree. This is much better than the linear time required to find items by key in an (unsorted) array, but slower than the corresponding operations on hash tables.

1.2 Ordering invariant

In a binary search tree, at any node with key k, all keys of the elements in the left sub tree are strictly less than k, while all keys of the elements in the right sub tree are strictly greater than k.

1.3 No order is really good

But we somehow assumed in our claim that on average each comparison allows us to skip half of the items which may not be the case unless its enforced every time the tree is modified. Lets consider some input scenarios and hypothesize their respective effect on the performance of the data structure,

1. Input being inserted in ascending order

2. Input being inserted in descending order

3. Input being inserted in alternating, outside-in order

4. Input being inserted in no particular order

None of the first three make for particularly efficient binary search trees because they all result in binary search trees structurally being equivalent to linked lists. The result of all of this is that the good O(log N) performance of binary search trees degenerates to O(N).

But if we consider the fourth scenario its particularly favourable, If we inserted the elements into the tree in a random order, it would generally turn out to be balanced enough to guarantee logarithmic insert and search times on the average. But we cannot reorder our input to disrupt any orderliness all the time, there could be cases where we don't get to the whole input in advance like an online scenario.

So it becomes essential to perform transformations on the tree at key insertion times, in order to keep the height in logarithmic proportion to its input size. Although a certain overhead is involved, it could be justified in the long run by ensuring fast execution of later operations.

1.4 AVL Trees

AVL trees enforce a key height property to make sure that the number of items are equally divided at each and every node among its left and right side. It is a special binary search tree such that for every internal node the heights of its left and right children can differ by at most 1. So AVL tree bears the cost of balancing the tree every time modifications are being made to it. This would in turn help in performing lookups better in average.

Chapter 2

AVL Tree Operations

Since AVL tree rebalances itself on every modification the rebalance code segment will follow both insertion and deletion operation. This section details key operations in AVL highlighting the additional responsibilities to the normal binary search tree.

2.1 AVL Invariant

In addition to the invariants of the binary search tree,each node of an AVL tree has the invariant property that the heights of the sub-tree rooted at its children differ by at most one.

$height(node.left) - height(node.right) <= 1$

The difference between the height of left and right sub tree is termed as balance factor and it plays a crucial role to decide when and which side to rotate.

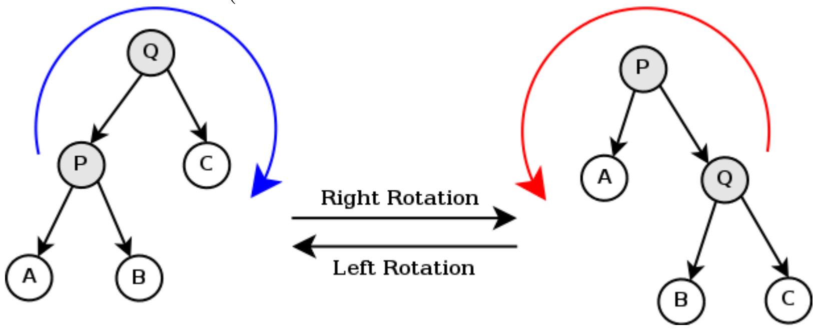
2.2 AVL Rotations

Tree rotation is an operation employed by AVL to change the structure of the tree without interfering with the order of the elements. A tree rotation moves one node up in the tree and one node down. Hence it changes the shape of the tree, and in particular it decreases its height by moving smaller sub trees down and larger sub trees up, resulting in improved performance of many tree operations.

There are two rotations one for either side which are exactly inverses of each other.

$LEFT - ROTATE(RIGHT - ROTATE(T)) = T$

The right rotation operation as shown in the image below is performed with Q as the root and hence is a right rotation on, or rooted at, Q. This operation results in a rotation of the tree in the clockwise direction. The inverse operation is the left rotation, which results in a movement in a counter-clockwise direction (the left rotation shown above is rooted at P).



Rotation Invariants

1. BST Invariant should hold after rotation

2. After a rotation, the side of the rotation increases its height by 1 while the side opposite the rotation decreases its height similarly.

AVL tree actually makes use of the second property to adjust the height imbalance.

Using the terminology of **Root** for the parent node of the subtrees to rotate, **Pivot** for the node which will become the new parent node, **RS** for rotation side upon to rotate and **OS** for opposite side of rotation. In the above diagram for the root Q, the **RS** is C and the OS is P. The pseudo code for the rotation is:

```
Pivot = Root.OS
Root.OS = Pivot.RS
Pivot.RS = Root
Root = Pivot
```

If we closely observe, this is actually a constant time operation which in turn make it very effective in practical.

2.3 Operation Recipe

Simple approach to any AVL operation is to do the naive binary search tree steps and then use the rotations to fix the AVL invariant violations.

$AVL \quad Operation = \{BSTOperation; Fix \quad AVL \quad Violations\}$

2.4 AVL Insertions

Insertion can be handled by above operation recipe,

After inserting a node, it is necessary to check each node starting with the added node and advancing up to the root for consistency with the invariants of AVL trees. This can be achieved by considering the balance factor of each node, which is defined as the difference in heights of the left and right sub trees.

Thus the balance factor of any node of an AVL tree is in the integer range [-1,+1]. Since with a single insertion the height of an AVL subtree cannot increase by more than one, the temporarily recomputed balance factor of a node after an insertion will be in the range [2,+2]. For each node checked, if the recomputed balance factor remains in the range [1,+1] no rotations are necessary. However, if the recomputed balance factor becomes less than 1 or greater than +1, the subtree rooted at this node is unbalanced, and a rotation is applied.

After a rotation a subtree has the same height as before, so retracing can stop. In order to restore the balance factors of all nodes, first observe that all nodes requiring correction lie along the path used during the initial insertion. If the above procedure is applied to nodes along this path, starting from the bottom (i.e. the inserted node), then every node in the tree will again have a balance factor of 1, 0, or 1.

The time required is O(log n) for lookup, plus a maximum of O(log n) retracing levels on the way back to the root, so the operation can be completed in O(log n) time.

2.5 AVL Deletions

We can use similar approach for deletion as well. We need to search for the key and once we find the node to delete there are three cases, leaf, single child, both the child nodes. In the two children case we need to find the replacement node and switch with the current node to be deleted.

After we replace, the ritual is similar walk back towards the root and update the height, compute balance factor and rebalance whenever necessary.

The deletion code is little bit messier than the insertion code, but the idea is similar. Suppose we have deleted a key from the left subtree and that as a result this subtree's height has decreased by one, and this has caused some ancestor to violate the invariant. There could be four cases,

1. Balance is greater than one and its left child is heavy then we just need to rotate it right.

2. Balance is greater than one and its left child is lighter then we need to first balance the left child then right rotate the current node

3. Balance is lesser than one and its right child is heavy as well then in this case both nodes need to be fixed, we need to rotate the right child right and left rotate the current node.

4. The last possibility is current node's balance is negative expecting a left rotation but the right side is lighter as well. Left rotation that we perform for the parent is enough to fix this as well.

The rotation operations (left and right rotate) both take constant time as only respective fields are being changed every time. Updating the height and getting the balance factor also take constant time, both can be computed locally and we can easily propagate it from bottom to top. So the time complexity of AVL delete remains same as BST delete which is O(h) where h is height of the tree. Since AVL tree is balanced, the height is O(Logn).

Chapter 3

AVL Tree Performance plots

To numerically characterize the performance of the data structure randomized uniform distribution of input data has been used to perform the operation and measure the respective rotations being performed and time taken to complete the requests

3.1 AVL Tree Height

Worst case value: $O(\log n)$

Key characteristic that AVL tree sets out to achieve is to maintain the height of the search tree balanced all the time.

We can show that an AVL tree with n nodes has $O(\log n)$ height. Let N_h represent the minimum number of nodes that can form an AVL tree of height h.

If we know N_{h-1} and N_{h-2}, we can compute N_h. Since the height for this tree is h, the root must have $h-1$ height child nodes. By the AVL property if one of the child node has N_{h-1} then the other should be N_{h-2}

So Total No of nodes $= N_{h-1} + N_{h-2} + 1$

$N_h = N_{h-1} + N_{h-2} + 1$

$N_{h-1} = N_{h-2} + N_{h-3} + 1$

$N_h = N_{h-2} + N_{h-3} + 1 + N_{h-2} + 1$

$N_h > 2 * N_{h-2}$

$N_h > 2^{h/2}$

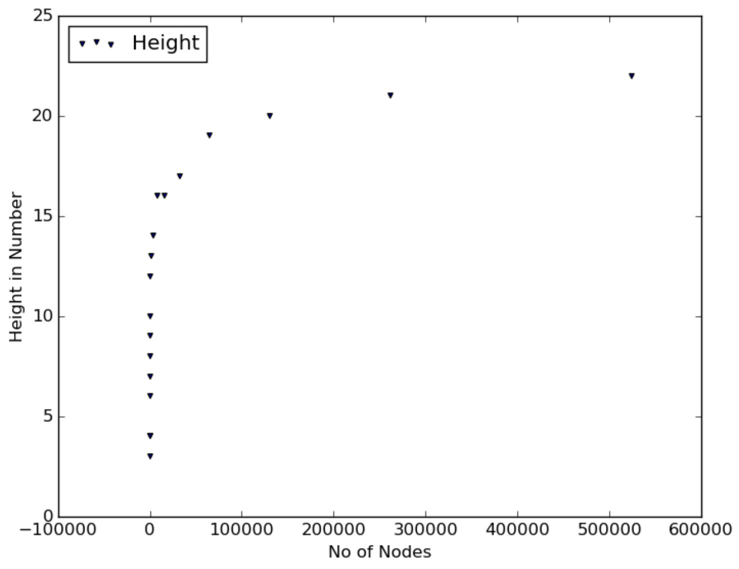
$\log N_h > \log 2^{h/2}$

$2 * \log N_h > h$

$h = O(\log N_h)$

Empirical results to back above claim,

Figure 3.1: Height of tree vs input size



The x axis refers to no of nodes present in the tree and its y axis refers to the height of the tree.

3.2 AVL Tree Insertions

Worst case performance: $O(\log n)$

1. AVL tree invariant keeps the height of the tree being constructed logarithmic. The highest path from root to tip of the tree is going to log n.

2. Insertion of a node can increase the size of the tree by one and hence disrupt the balancing factor at each node at max by one on either side.

3. To rebalance the tree we need to walk back from the tip of insertion back to root and at each step we need to rebalance it if needed.

So Total cost = [Steps to reach the insertion point + (Steps to reach root node back * rebalancing cost)]

By above three arguments,

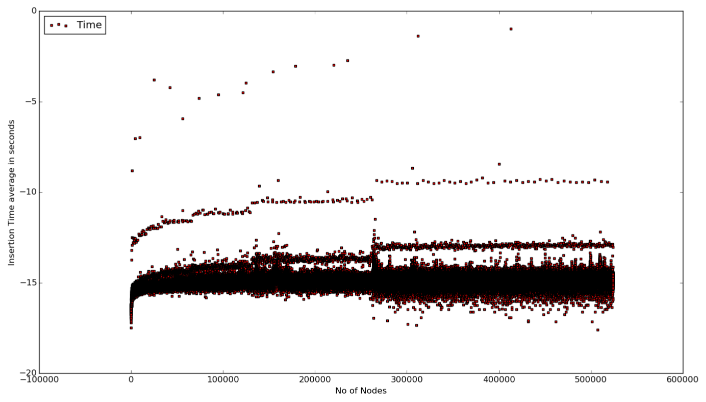
Total cost = $[O(\log n) + (O(\log n) * c)]$

On simplifying,

Insertion cost = $O(\log n)$

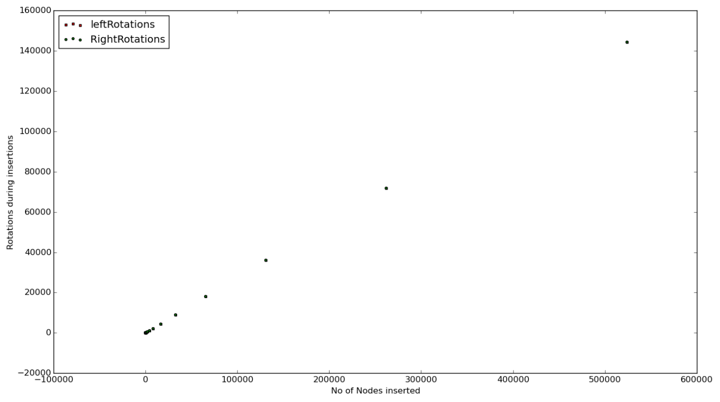
Empirical plot to back the above claim,

Figure 3.2: Insertion time vs input size



The x axis refers to no of nodes already present in the tree when insertion is being attempted and its y axis refers to the time taken to perform the insertion. That en-composes rebalancing and traversal efforts together to accomplish the insertion.

Figure 3.3: No of Rotations vs input size



Similar to previous figure, x axis refers to no of nodes already present in the tree when insertion is being attempted but y axis here refers to the total no of left and right rotations being carried out to complete insertion.

3.3 AVL Tree Search

Worst case performance: $O(\log n)$

1. AVL tree invariant keeps the height of the tree being constructed logarithmic. The highest path from root to tip of the tree is going to log n.

2. BST tree invariant keeps orderliness of data item between the node and its left and right child nodes. This enables the search operation to ignore half of the search space using single comparison during each step.

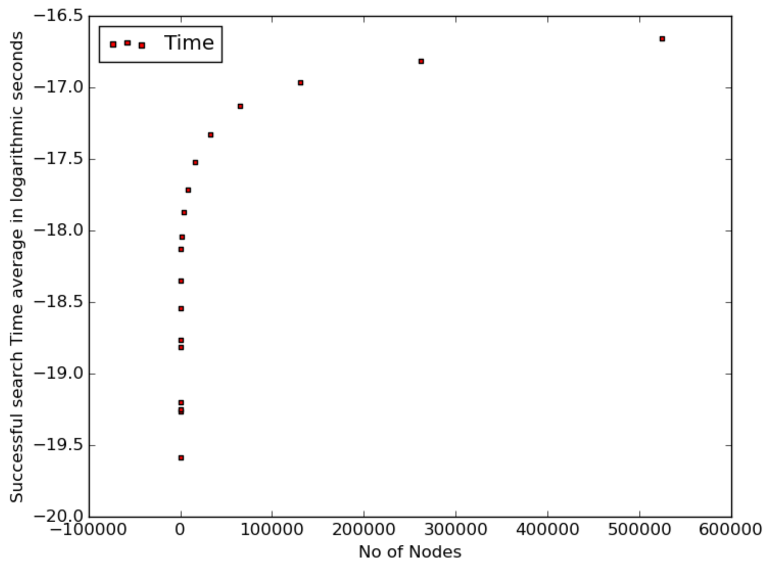
So Total cost = [Steps to reach bottom node from root]

By above argument,

Search cost = $O(\log n)$

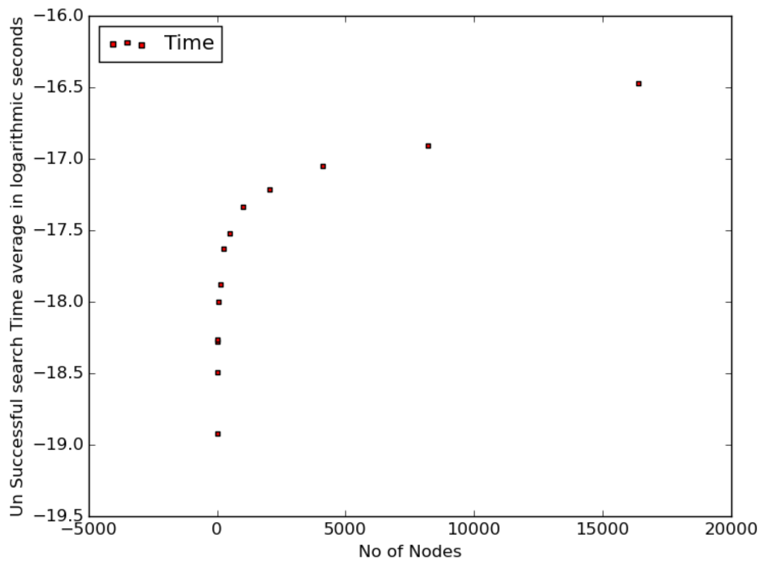
Empirical plot to back the above claim,

Figure 3.4: Successful Search time vs input size



The x axis refers to no of nodes present in the tree when search is being attempted and its y axis refers to the time taken to perform the search.

Figure 3.5: Un Successful Search time vs input size



Similar to previous figure, x axis refers to no of nodes present in the tree when search is being attempted but y axis here refers to the time taken for un successful searches. We can selectively picked items which aren't present in the tree for this plot.

3.4 AVL Tree Deletions

Worst case performance: $O(\log n)$

1. AVL tree invariant keeps the height of the tree being constructed logarithmic. The highest path from root to tip of the tree is going to log n.

2. Deletion of a node can decrease the size of the tree by one and hence disrupt the balancing factor at each node at max by one on either side.

3. To rebalance the tree we need to walk back from the point of deletion to root and at each step we need to rebalance it if needed.

4. Only difference from insertion scenario is there will be more rebalancing cases compared to insertion. But that's not a problem since we established that balancing is constant cost operation.

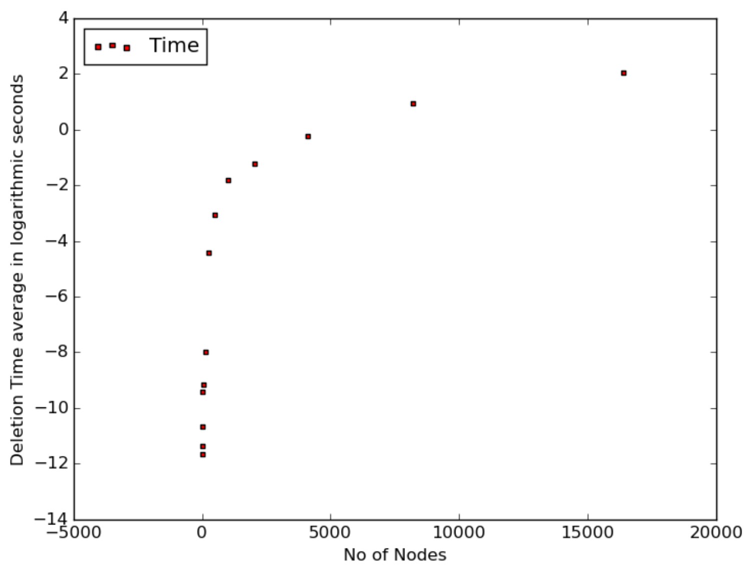
So Total cost = [Steps to reach the deletion point + (Steps to reach root node back * rebalancing cost)]

By above three arguments,

Total cost = $[O(\log n) + (O(\log n) * c)]$

On simplifying,
Deletion cost $= O(\log n)$
Empirical plot to back the above claim,

Figure 3.6: Deletion time vs input size



The x axis refers to no of nodes already present in the tree when deletion is being attempted and its y axis refers to the time taken to perform the deletion. That en-composes rebalancing and traversal efforts together to accomplish it.

Chapter 4

Implementation Notes

Implementation language - Python

1. Implementation of the data structure is accomplished with a single class named **node**, single package **avl** and defining all the operations as functions inside the class node.

2. Correctness of the individual operations are being verified with unit test per function housed in test file **test avl**

3. Performance evaluation are being computed using separate test scripts for each operation **deletion time test**, **insertion time test**, **search time test**.

The performance evaluation is being done using a randomized uniform distribution of inputs being sampled every time and results averaged relative to n being plotted.

Chapter 5

Conclusion

AVL trees are pretty much close to an optimal balanced binary search trees by simple balancing operations. We can be rest assured that the O(log N) performance of binary search trees is guaranteed with AVL trees, but the extra bookkeeping required to maintain an AVL tree can be prohibitive, especially if deletions are common. Insertion into an AVL tree only requires one single or double rotation, but deletion could perform up to O(log N) rotations. However, those cases are rare, and still very fast.

Among it contemporaries AVL trees are basically most strict enforcer of height balance. Hence they are more apt when degenerate input sequences are common, and there is not much locality of reference in nodes. If degenerate sequences are not common, and searches are random then a less rigid balanced tree such as red black trees are a better solution. If there is a significant amount of locality to searches, then other locality based balancing approach like splay tree is theoretically better than plain height based balancing approaches.

References

Author, I. (Year). *Book Title*, Publisher; Place of publication.

Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. (2001), *Introduction to Algorithms (2nd ed.)*, McGraw-Hill Higher Education.

Amani, M. and Lai, K. A. and Tarjan, R. E.. (2015). ‘Amortized Rotation Cost in AVL Trees’, *ArXiv e-prints*

MIT Introduction to Algorithms class AVL Trees recitation Notes.

AVL Trees, Self Balancing binary Search Trees Wikipedia page

Ben Pfaff. (2004). ‘Performance analysis of BSTs in system software.’, *SIGMETRICS ’04/Performance ’04*, **27**, pp.435–48.