AVL TREE AND ITS OPERATIONS

Siddharth Nair, Simran Singh Oberoi, Shubham Sharma
ECE Dept, Dronacharya College of Engg

***Abstract* - This research paper focuses on AVL tree and the various operations that are performed in AVL tree.These operations are namely insertion, deletion, searching, and traversal. This paper in brief discusses about the all the operation of AVL tree.**

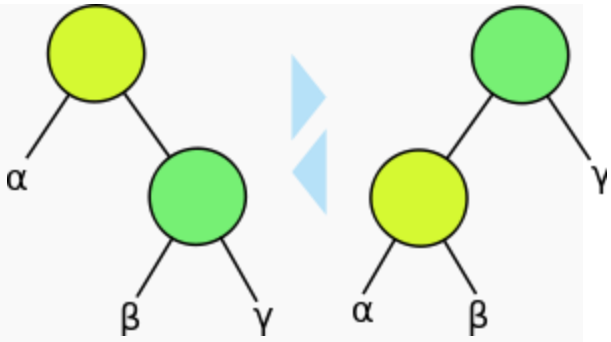
***Index Terms*- AVL, binary, insertion, deletion, self-balancing, subtrees, nodes, traversal, rotation, operation**

I. INTRODUCTION

In computer science, an **AVL tree** is a self-balancing binary search tree. It was the first such data structure to be invented.[1] In an AVL tree, 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. 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. Insertions and deletions may require the tree to be rebalanced by one or more tree rotations.

The AVL tree is named after its two Soviet inventors, Georgy Adelson-Velsky and E. M. Landis, who published it in their 1962 paper "An algorithm for the organization of information".[2]AVL trees are often compared with red-black trees because both support the same set of operations and take O(log *n*) time for the basic operations. For lookup-intensive applications, AVL trees are faster than red-black trees because they are more rigidly balanced.[3] Similar to red-black trees, AVL trees are height-balanced. Both are in general not weight-balanced nor μ-balanced for any $\mu \leq \frac{1}{2}$;[4] that is, sibling nodes can have hugely differing numbers of descendants.

II. OPERATIONS



Tree rotations

Basic operations of an AVL tree involve carrying out the same actions as would be carried out on an unbalanced binary search tree, but modifications are followed by zero or more operations called tree rotations, which help to restore the height balance of the subtrees.

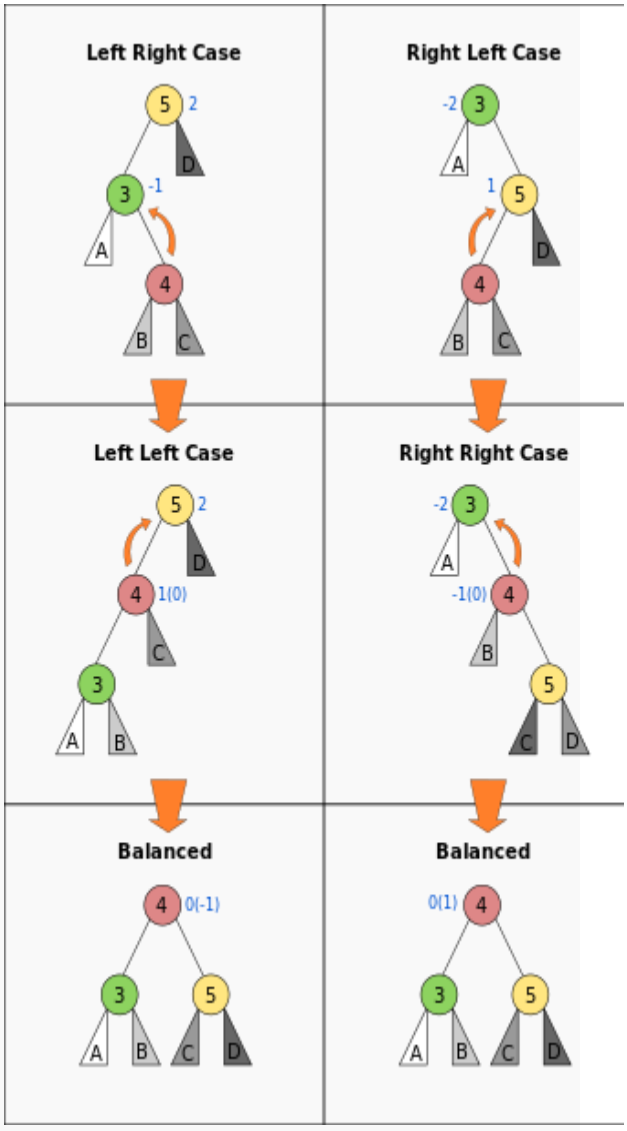
Searching

Searching in an AVL tree is done as in any binary search tree. The special thing about AVL trees is that the number of comparisons required, i.e. the AVL tree's height, is guaranteed never to exceed log(*n*).

Traversal

Once a node has been found in a balanced tree, the *next* or *previous* nodes can be explored in amortized constant time. Some instances of exploring these "nearby" nodes require traversing up to log(*n*) links (particularly when moving from the rightmost leaf of the root's left subtree to the root or from the root to the leftmost leaf of the root's right subtree; in the example AVL tree, moving from node 14 to the *next but one* node 19 takes 4 steps). However, exploring all *n*nodes of the tree in this manner would use each link exactly twice: one traversal to enter the subtree rooted at that node, another to leave that node's subtree after having explored it. And since there are *n*−1 links in any tree, the amortized cost is found to be 2×(*n*−1)/*n*, or approximately 2.

Insertion



Pictorial description of how rotations rebalance an AVL tree. The numbered circles represent the nodes being rebalanced. The lettered triangles represent subtrees which are themselves balanced AVL trees. A blue number next to a node denotes possible balance factors (those in parentheses occurring only in case of deletion).

After inserting a node, it is necessary to check each of the node's ancestors for consistency with the rules of AVL ("retracing"). The balance factor is calculated as follows: balanceFactor = height(left subtree) - height(right subtree). Since with a single insertion the height of an AVL subtree cannot increase by more than one, the temporary balance factor of a node will be in the range from −2 to +2. For each node checked, if the balance factor remains in the range from −1 to +1 then only corrections of the balance factor, but no rotations are necessary. However, if balance factor becomes less than −1 or greater than +1, the subtree rooted at this node is unbalanced.

Description of the Rotation

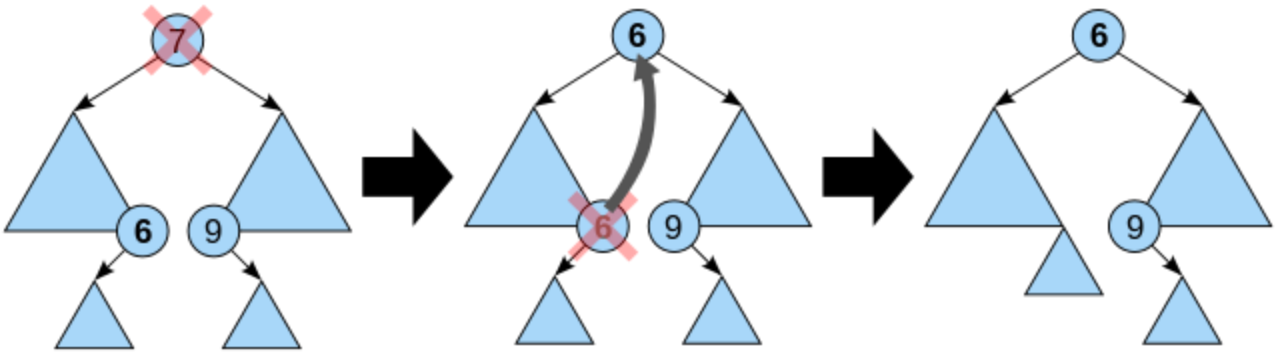
Let us first assume the balance factor of a node P is 2 (as opposed to the other possible unbalanced value −2). This case is depicted in the left column of the illustration with P:=**5**. We then look at the left subtree (the larger one) with root N. If this subtree does not lean to the right - i.e. N has balance factor 1 (or, when deletion also 0) - we can rotate the whole tree to the right to get a balanced tree. This is labelled as the "Left Left Case" in the illustration with N:=**4**. If the subtree does lean to the right - i.e. N:=**3** has balance factor −1 - we first rotate the subtree to the left and end up the previous case. This second case is labelled as "Left Right Case" in the illustration.If the balance factor of the node P is −2 (this case is depicted in the right column of the illustration P:=**3**) we can mirror the above algorithm. I.e. if the root N of the (larger) right subtree has balance factor −1 (or, when deletion also 0) we can rotate the whole tree to the left to get a balanced tree. This is labelled as the "Right Right Case" in the illustration with N:=**4**. If the root N:=**5** of the right subtree has balance factor 1 ("Right Left Case") we can rotate the subtree to the right to end up in the "Right Right Case".

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.

Deletion

Let node X be the node with the value we need to delete, and let node Y be a node in the tree we need to find to take node X's place, and let node Z be the actual node we take out of the tree.



Deleting a node with two children from a binary search tree using the in-order predecessor (rightmost node in the left subtree, labelled **6**).

Steps to consider when deleting a node in an AVL tree are the following:

1. If node X is a leaf or has only one child, skip to step 5 with Z:=X.
2. Otherwise, determine node Y by finding the largest node in node X's left subtree (the in-order predecessor of X − it does not have a right child) or the smallest in its right subtree (the in-order successor of X − it does not have a left child).
3. Exchange all the child and parent links of node X with those of node Y. In this step, the in-order sequence between nodes X and Y is temporarily disturbed, but the tree structure doesn't change.
4. Choose node Z to be all the child and parent links of old node Y = those of new node X.
5. If node Z has a subtree (which then is a leaf) attach it to Z's parent.
6. If node Z was the root (its parent is null), update root.
7. Delete node Z.
8. Retrace the path back up the tree (starting with node Z's parent) to the root, adjusting the balance factors as needed.

Since with a single deletion the height of an AVL subtree cannot decrease by more than one, the temporary balance factor of a node will be in the range from −2 to +2.

If the balance factor becomes ±2 then the subtree is unbalanced and needs to be rotated. The various cases of rotations are depicted in section "Insertion" together with a brief description.

The retracing can stop if the balance factor becomes ±1 indicating that the height of that subtree has remained unchanged. This can also result from a rotation when the higher child tree has a balance factor of 0. (In case of an insertion the higher child tree involved in a rotation always has a balance factor of ±1.)

If the balance factor becomes 0 then the height of the subtree has decreased by one and the retracing needs to continue. This can also result from a rotation.

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.

III. CONCLUSION

With the help of this paper we are able to study and understand the AVL tree and also its various operations.

REFERENCES

1) www.google.com
2) www.wikipedia.com
3) Data Structure- AK Sharma