© 2014 IJIRT | Volume 1 Issue 7 | ISSN: 2349-6002

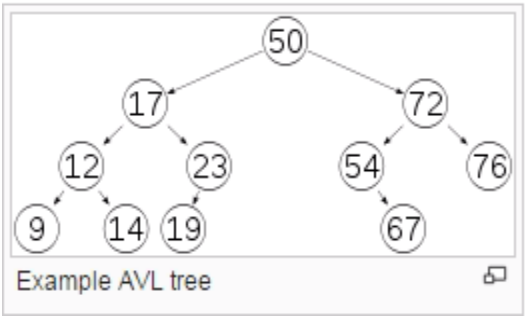
AVL TREE AND HASHING

Gajender, Gaurav, Himanshu Sharma

Abstract- **AVL tree (Georgy Adelson-Velsky and Landis' tree, named after the inventors) is a self-balancing binary search tree. It was the first such data structure to be invented. 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. In this paper we have written about AVL tree and various operations of AVL tree and what is hashing.**

I. INTRODUCTION

In computer science, an AVL tree (Georgy Adelson-Velsky and Landis' tree, named after the inventors) is a self-balancing binary search tree. It was the first such data structure to be invented. 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". 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. Similar to red-black trees, AVL trees are height-balanced.Both are in general not weight-balanced nor μ-balanced for any $\mu \leq \frac{1}{2}$; that is, sibling nodes can have hugely differing numbers of descendants.



Example AVL tree

II. OPERATIONS

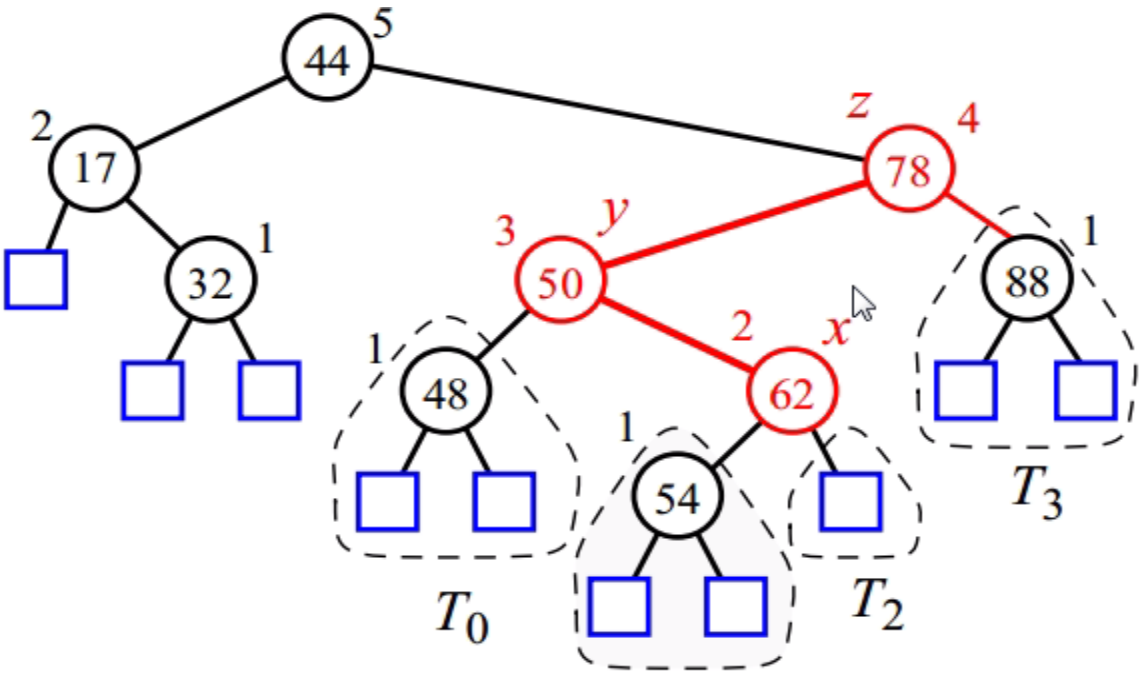
Basic operations of an AVL tree involve carrying out the same actions as would be carried out on an unbalanced binary search tree, but modifications are followed by zero or more operations called tree rotations, which help to restore the height balance of the subtrees.

III. SEARCHING

We begin by examining the root node. If the tree is null, the key we are searching for does not exist in the tree. Otherwise, if the key equals that of the root, the search is successful and we return the node. If the key is less than that of the root, we search the left subtree. Similarly, if the key is greater than that of the root, we search the right subtree. This process is repeated until the key is found or the remaining subtree is null. If the searched key is not found before a null subtree is reached, then the item must not be present in the tree.

IV. 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 onenode 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.

V. INSERTION

• A binary search tree T is called balanced if for every node v, the height of v's children differ by at most one.

• Inserting a node into an AVL tree involves performing an expand External(w) on T, which changes the heights of some of the nodes in T.

• If an insertion causes T to become unbalanced, we travel up the tree from the newly created node until we find the first node x such that its grandparent z is unbalanced node.

• Since z became unbalanced by an insertion in the subtree rooted at its child y, height(y) = height(sibling(y)) + 2

• To rebalance the subtree rooted at z, we must perform a restructuring

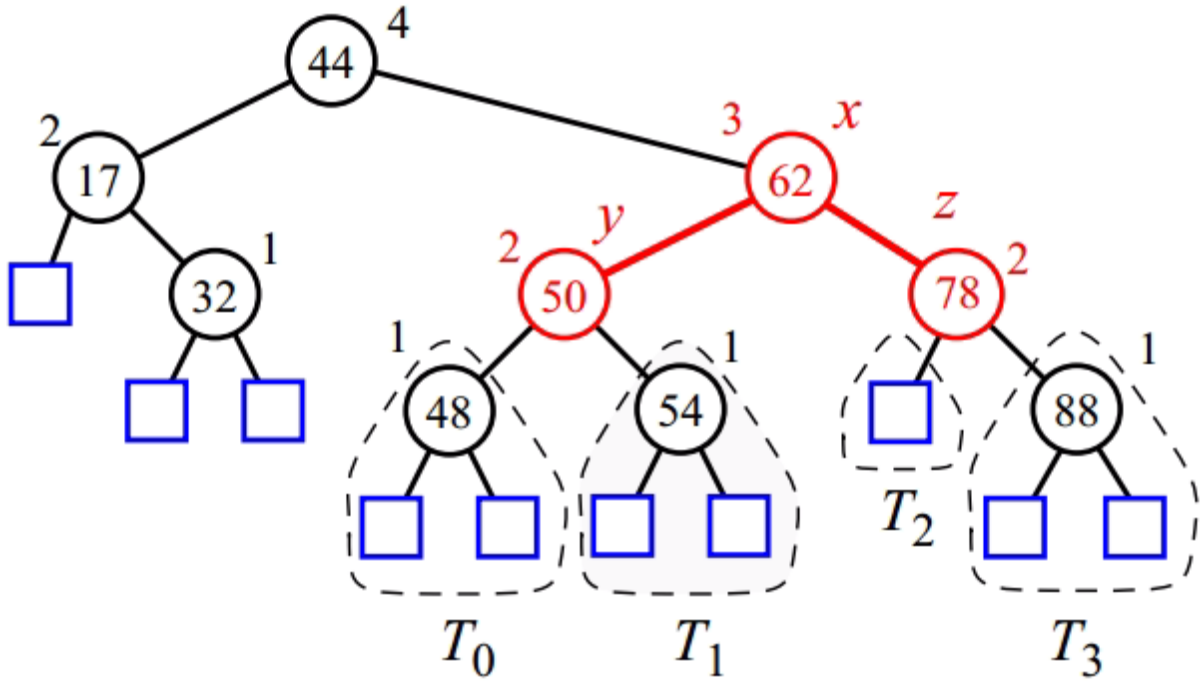
- we rename x, y, and z to a, b, and c based on the order of the nodes in an in-order traversal.

- z is replaced by b, whose children are now a and c whose children, in turn, consist of the four other subtrees

formerly children of x, y, and z

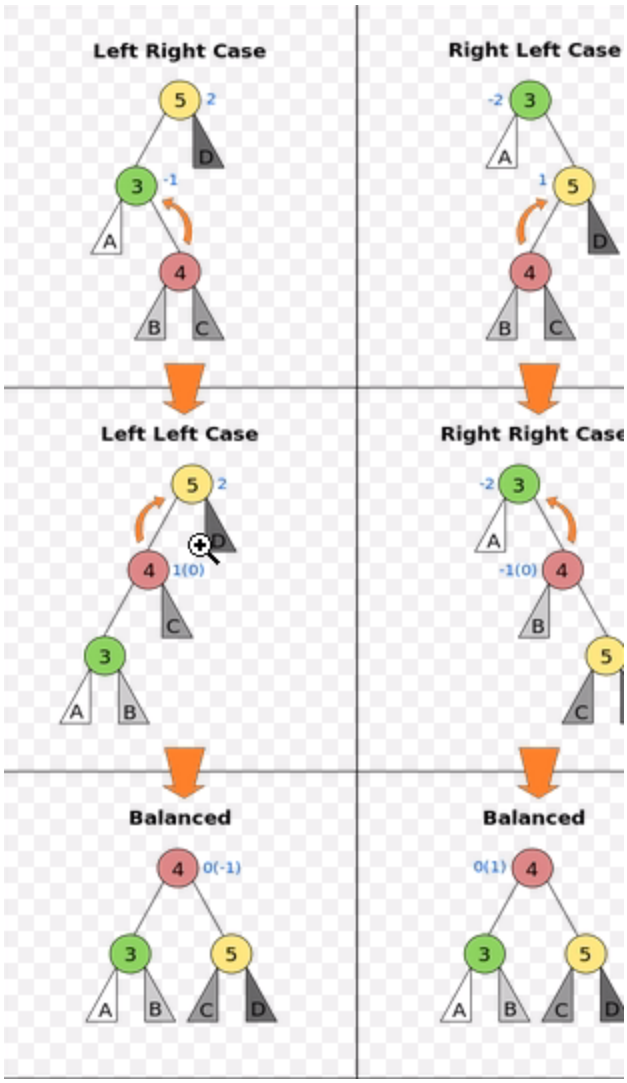
VI. ROTATIONS IN AVL TREES

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.

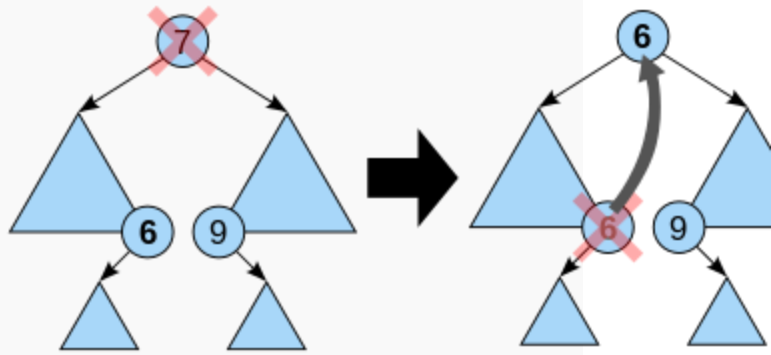
If the balance factor of the node P is −2 (this case is depicted in the right column of the illustration P:=3) we can mirror the above algorithm. I.e. if the root N of the (larger) right subtree has balance factor −1 (or, when deletion also 0) we can rotate the whole tree to the left to get a balanced 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"



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

VII. 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.

VIII. HASHING

Hashing is the primary form of organization used to provide direct access. It was developed during 1950's. The hashing algorithm uses a hash function—probably the term is derived from the idea that the resulting hash value can be thought of as a "mixed up" version of the represented value. Hashing is a technique where we store the records into a given address where it maps large data sets of variable-length called keys, to smaller data sets of a fixed length called hash address. Hashing is the transformation of a string of characters into a usually shorter fixed-length value or key that represents the original string. In hashing algorithm, hash functions are used to generate a random address based on the specified key value. The random address generated by the has functions will be used to store and retrieve the elements. Hashing provides the direct access to the records using hash functions, which generates the address where record needs to stored or retrieved respective to the key value. It is also used in many encryption algorithms. The major drawback of the hashing is occurrence of collisions, where in hashing function generates a same random address a output of hash function on more than one key value. as we can see in the above diagram, a hash function that maps names to integers from 0 to 15.Where 2 different persons names are being stored in the same address .insertion of the names are overlapped here and while retrieving it leads to a problem as 2 names are hashed to a same address.

REFERENCES:-

http://www.sahyadri.edu.in/e-journal/Hashing.pdf

http://jriera.webs.ull.es/Docencia/ch7.2.pdf

http://en.wikipedia.org/wiki/AVL_tree