

A brief Study of Balancing of AVL Tree

Shivani Chauhan , ShashankThakur, Sheetal Rana, Saurav Sharma

Department of Computer Science Engineering, Dronacharya college of Engineering, Gurgaon
(Haryana) INDIA

Email: shivanichauhan2011@gmail.com

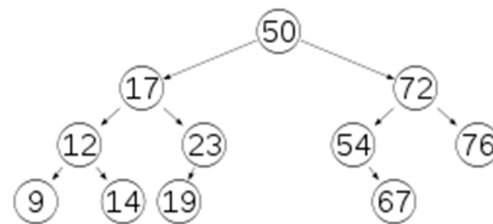
Abstract:

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.

Keywords: AVL, insertion, deletion.

1. Introduction

Binary search trees are an excellent data structure to implement associative Arrays, maps, sets, and similar interfaces. The main difficulty, as discussed in last lecture, is that they are efficient only when they are balanced. Straight forward sequences of insertions can lead to highly unbalanced trees with poor asymptotic complexity and unacceptable practical efficiency. In this paper we use AVL trees, which is a simple and efficient data structure to maintain balance, and is also the first that has been proposed. It is named after its inventors, G.M. Adelson-Velskii and E.M. Landis, who described it in 1962.



2. 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.

2.1 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 $\log(n)$. Searching for a key in an AVL tree is identical to searching for it in a plain Binary search tree as described in The reason is that we only Need the ordering invariant to find the element; the height invariant is only Relevant for inserting an element. To describe AVL trees we need the concept of tree height, which we define as the maximal length of a path from the root to a leaf. So the empty tree has height 0, the tree with one node has height 1, a balanced tree with three nodes has height 2. If we add one more node to this last tree is will have height 3. Alternatively, we can define it recursively by saying that the

empty tree has height 0, and the height of any node is one greater than the maximal height of its two children. AVL trees maintain a height invariant (also sometimes called a balance invariant).

2.2 Traversal

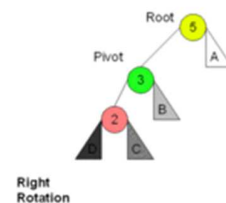
Once a node has been found in a balanced tree, the next or previous nodes can be explored in amortized constant time. Some instances of exploring these "nearby" nodes require traversing up to $\log(n)$ links (particularly when moving from the rightmost leaf of the root's left subtree to the root or from 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 $2 \times (n-1)/n$, or approximately 2.

3.Insertion :- The basic recursive structure of inserting an element is the same as for searching for an element. We compare the element's key with the keys associated with the nodes of the trees, inserting recursively into the left or right subtree. When we find an element with the exact key we overwrite the element in that node. If we encounter a null tree, we construct a new tree with the element to be inserted and no children and then return it. As we return the new subtrees (with the inserted element) towards the root, we check if we violate the height invariant. If so, we rebalance to restore the invariant and then continue up the tree to the root. The main cleverness of the algorithm lies in analyzing the situations when we have to rebalance and applying the appropriate rotations to restore the height invariant. It turns out that one or two rotations on the whole tree always suffice for each insert operation, which is a very elegant result. First, we keep in mind that the left and right subtrees' heights before the insertion can differ by at most one. Once we insert an element

into one of the subtrees, they can differ by at most two. We now draw the trees in such a way that the height of a node is indicated by the height that we are drawing it at. The first situation we describe is where we insert into the right subtree, which is already of height $h + 1$ where the left subtree has height h . If we are unlucky, the result of inserting into the right subtree will give us a new right subtree of height $h + 2$ which raises the height of the overall tree to $h + 3$, violating the height invariant. In the new right subtree has height $h+2$, either its right or the left subtree must be of height $h+1$ (and only one of them; think about why). If it is the right subtree we are in the situation depicted below on the left. The code for inserting an element into the tree is mostly identical with the code for plain binary search trees. The difference is that after we insert into the left or right subtree, we call a function `rebalance_left` or `rebalance_right`, respectively, to restore the invariant if necessary and calculate the new height.

3. 1Rotation in insertion operation In case of insertion, we have following rotations.

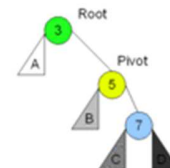
Left Left Case



Right Rotation

RR Rotation When a node X is inserted in the right sub tree of right

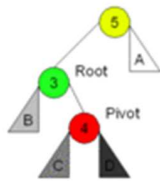
Right Right Case



Left Rotation

sub tree of node N. **LR Rotation** When a node X is inserted in the right sub tree of left sub tree of node N.

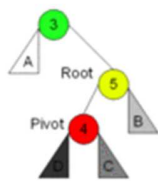
Left Right Case



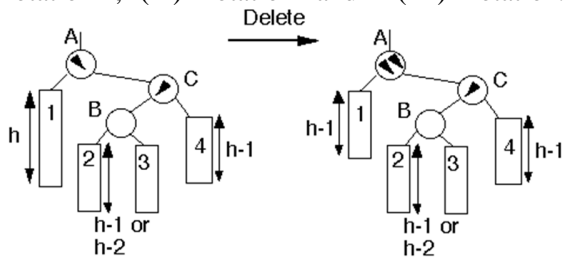
Left Rotation

RL Rotation When a node X is inserted in the left sub tree of right sub

Right Left Case



tree of node N. **Right** LL rotation and RR rotation are mirror image of each other. Similarly LR rotation and RL rotation are mirror image of each other. **4. Rotation in deletion operation** In case of deletion, we have following rotations. 4.1 Let the deletion is being performed into right sub tree then we have three types of rotations R(0) rotation ,R(-1) rotation and R(+1) rotation. 4.2 Let the deletion is being performed into left sub tree then we have three types of rotations L(0) rotation ,L(-1) rotation and L(+1) rotation.



5. ALGORITHM 1-Traverse the BST into in order and count the total number of nodes m. **2**-if $m \equiv 3 \pmod{4}$ **3**-The mid node of the in order traversal will be the root node, its predecessor will be the left node and successor will be the right node. **4**-if m is even then **5**-Introduce virtual node into the list at $(m/2 + 1)$ th index. **6**-Make this virtual node root. Keep the first $m/2$ nodes of original list into left sub tree and $m/2 + 1$ to m nodes into right sub tree. **7**-Repeat step 5th and 6th until each node contains only one key. **8**- Delete the virtual nodes. **9**- Else (if m is odd) **10**-Make the $(m/2)$ th index node as root node. **11**-put the first

$m/2$ nodes into left sub tree and $(m/2 + 1)$ th to m index nodes into right sub tree. That is left sub tree and right sub tree has even number of nodes. **12**-Repeat step 5, 6, 7. **13**-Delete the virtual nodes. **14**-Stop This algorithm will be used into both the cases i.e. insertion as well as in deletion. U = unbalanced node C = child of unbalanced node G = grandchild of unbalanced node N = target value that was just inserted T = pointer to new root of balanced subtree T = U
 $U.height = U.height - 2$
 if $N > U$
 if $N < U.right$
 $U.right = \text{Right Rotation on } U.right$
 $(U.right.left++)$
 $T = \text{Left Rotation on } U$
 else
 if $N > U.left$
 $U.left = \text{Left Rotation on } U.left$
 $(U.left.right++)$
 $T = \text{Right Rotation on } U$
 return T

6. References [1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data structures and algorithms*. Reading, Mass: Addison-Wesley, 1983. [2] www.agner.org/random/theory/. [3] J. Bentley, "Programming Pearl: How to sort", *Comm. ACM*, Vol. 27 Issue 4, April 1984. [4] J. L. BENTLEY, M. D. McILROY, "Engineering a Sort Function" *SOFTWARE—PRACTICE AND EXPERIENCE*, Vol. 23(11), Nov. 1993, pp 249 – 1265 . [5] J. L. Bentley and R. Sedgwick, "Fast algorithms for sorting and searching strings", *In Proc. 8th annual ACM-SIAM symposium on Discrete algorithms*, New Orleans, Louisiana, USA, 1997, pp 360 - 369 . [6] R. Chaudhuri and A. C. Dempster, "A note on slowing Quicksort", *SIGCSE* Vol . 25, No . 2, Jan 1993. [7] E. W. Dijkstra, *A discipline of programming*, Englewood Cliffs, NJ Prentice-Hall, 1976. [8] W. Dobosiewicz, "Sorting by distributive partitioning," *Information Processing Letters* 7, 1 – 5., 1978. [9] C.A.R. Hoare, "Algorithm 64: Quicksort," *Comm. ACM* 4, 7 , 321, July 1961. [10] C. A. R. Hoare, "Quicksort," *Computer Journal*, 5, pp 10 - 15 1962.