AVL tree

From Wikipedia, the free encyclopedia

In computer science, an **AVL tree** is a self-balancing binary search tree. It was the first such data structure to be invented.[2] 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 Evgenii Landis, who published it in their 1962 paper "An algorithm for the organization of information".[3]

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 strictly balanced.[4] Similar to red–black trees, AVL trees are height-balanced. Both are, in general, neither weight-balanced nor μ-balanced for any $\mu \leq 1/2$;[5] that is, sibling nodes can have hugely differing numbers of descendants.

AVL tree

Type	tree
Invented	1962
Invented by	Georgy Adelson-Velsky and Evgenii Landis

Time complexity in big O notation

Algorithm	Average	Worst Case
Space	$O(n)$	$O(n)$
Search	$O(\log n)$[1]	$O(\log n)$[1]
Insert	$O(\log n)$[1]	$O(\log n)$[1]
Delete	$O(\log n)$[1]	$O(\log n)$[1]

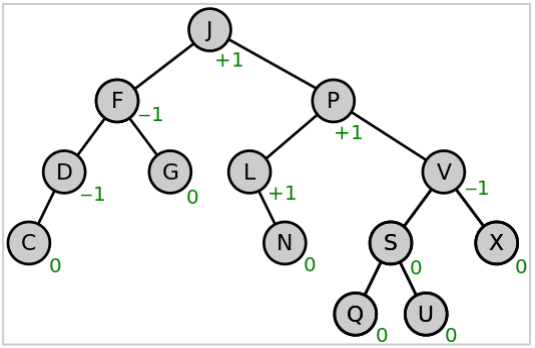

Fig. 1: AVL tree with balance factors (green)

Contents

Definition

Balance factor

In a binary tree the *balance factor* of a node N is defined to be the height difference

$$\text{BalanceFactor}(N) := -\text{Height}(\text{LeftSubtree}(N)) + \text{Height}(\text{RightSubtree}(N))$$ [6]

of its two child subtrees. A binary tree is defined to be an *AVL tree* if the invariant

$$\text{BalanceFactor}(N) \in \{-1,0,+1\}$$

holds for every node N in the tree.

A node N with $\text{BalanceFactor}(N) < 0$ is called "left-heavy", one with $\text{BalanceFactor}(N) > 0$ is called "right-heavy", and one with $\text{BalanceFactor}(N) = 0$ is sometimes simply called "balanced".

Remark

In the sequel, because there is a one-to-one correspondence between nodes and the subtrees rooted by them, we sometimes leave it to the context whether the name of an object stands for the node or the subtree.

Properties

Balance factors can be kept up-to-date by knowing the previous balance factors and the change in height – it is not necessary to know the absolute height. For holding the AVL balance information, two bits per node are sufficient.[7]

The height h of an AVL tree with n nodes lies in the interval:[8]

$$\log_2(n+1) \le h < c \log_2(n+2)+b$$

with the golden ratio $\varphi := (1+\sqrt{5})/2 \approx 1.618$, $c := 1/\log_2 \varphi \approx 1.44$, and $b := c/2 \log_2 5 - 2 \approx -0.328$. This is because an AVL tree of height h contains at least $F_{h+2} - 1$ nodes where $\{F_h\}$ is the Fibonacci sequence with the seed values $F_1 = 1, F_2 = 1$.

Operations

Read-only operations of an AVL tree involve carrying out the same actions as would be carried out on an unbalanced binary search tree, but modifications have to observe and restore the height balance of the subtrees.

Searching

Searching for a specific key in an AVL tree can be done the same way as that of a normal unbalanced binary search tree. In order for search to work effectively it has to employ a comparison function which establishes a total order (or at least a total preorder) on the set of keys. The number of comparisons required for successful search is limited by the height h and for unsuccessful search is very close to h, so both are in $O(\log n)$.

Traversal

Once a node has been found in an AVL tree, the *next* or *previous* node can be accessed in amortized constant time. Some instances of exploring these "nearby" nodes require traversing up to $h \propto \log(n)$ links (particularly when navigating 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 AVL tree of figure 1, moving from node P to the *next but one* node Q takes 3 steps). However, exploring all n nodes of the tree in this manner would visit each link exactly twice: one

downward visit to enter the subtree rooted by that node, another visit upward 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.

Insert

When inserting an element into an AVL tree, you initially follow the same process as inserting into a Binary Search Tree. Once this has been completed, you verify that the tree maintains the AVL property. If it does not, then you perform tree rotations going upwards from the inserted node to rectify this.

Delete

When deleting an element from an AVL tree, swap the desired element with the minimum element in the right subtree, or maximum element in the left subtree. Once this has been completed delete the element from the new position (the process may need to be repeated). If the element is now a leaf node, remove it completely. Make sure to perform rotations to maintain the AVL property.

Set operations and bulk operations

In addition to the single-element insert, delete and lookup operations, several set operations have been defined on AVL trees: union, intersection and set difference. Then fast *bulk* operations on insertions or deletions can be implemented based on these set functions. These set operations rely on two helper operations, *Split* and *Join*. With the new operations, the implementation of AVL trees can be more efficient and highly-parallelizable.[9]

- *Join*: The function *Join* is on two AVL trees t_1 and t_2 and a key k and will return a tree containing all elements in t_1, t_2 as well as k. It requires k to be greater than all keys in t_1 and smaller than all keys in t_2. If the two trees differ by height at most one, *Join* simply create a new node with left subtree t_1, root k and right subtree t_2. Otherwise, suppose that t_1 is higher than t_2 for more than one (the other case is symmetric). *Join* follows the right spine of t_1 until a node c which is balanced with t_2. At this point a new node with left child c, root k and right child t_1 is created to replace c. The new node satisfies the AVL invariant, and its height is one greater than c. The increase in height can increase the height of its ancestors, possibly invalidating the AVL invariant of those nodes. This can be fixed either with a double rotation if invalid at the parent or a single left rotation if invalid higher in the tree, in both cases restoring the height for any further ancestor nodes. *Join* will therefore require at most two rotations. The cost of this function is the difference of the heights between the two input trees.
- *Split*: To split an AVL tree into two smaller trees, those smaller than key x, and those larger than key x, first draw a path from the root by inserting x into the AVL. After this insertion, all values less than x will be found on the left of the path, and all values greater than x will be found on the right. By applying *Join*, all the subtrees on the left side are merged bottom-up using keys on the path as intermediate nodes from bottom to top to form the left tree, and the right part is asymmetric. The cost of *Split* is order of $O(n)$, the height of the tree.

The union of two AVLs t_1 and t_2 representing sets A and B, is an AVL t that represents $A \cup B$. The following recursive function computes this union:

```
function union(t1, t2):
    if t1 = nil:
        return t2
    if t2 = nil:
        return t1
    t<, t> ← split t2 on t1.root
    return join(t1.root,union(left(t1), t<),union(right(t1), t>))
```

Here, *Split* is presumed to return two trees: one holding the keys less its input key, one holding the greater keys. (The algorithm is non-destructive, but an in-place destructive version exists as well.)

The algorithm for intersection or difference is similar, but requires the *Join2* helper routine that is the same as *Join* but without the middle key. Based on the new functions for union, intersection or difference, either one key or multiple keys can be inserted to or deleted from the AVL tree. Since *Split* calls *Join* but does not deal with the balancing criteria of AVL trees directly, such an implementation is usually called the "join-based" implementation.

The complexity of each of union, intersection and difference is $O\left(m\log\left(\frac{n}{m}+1\right)\right)$ for AVLs of sizes m and $n(\geq m)$. More importantly, since the recursive calls to union, intersection or difference are independent of each other, they can be executed in parallel with a parallel depth $O(\log m \log n)$.[9] When $m = 1$, the join-based implementation has the same computational DAG as single-element insertion and deletion.

Comparison to other structures

Both AVL trees and red–black trees are self-balancing binary search trees and they are related mathematically. Indeed, every AVL tree can be colored red–black. The operations to balance the trees are different; both AVL trees and red-black require $O(1)$ rotations in the worst case, while both also require $O(\log n)$ other updates (to colors or heights) in the worst case (though only $O(1)$ amortized). AVL trees require storing 2 bits (or one trit) of information in each node, while red-black trees require just one bit per node. The bigger difference between the two data structures is their height limit.

For a tree of size $n \geq 1$

- an AVL tree's height is at most

$$\begin{aligned} h &\leqq c\log_2(n+d)+b \\ &< c\log_2(n+2)+b \end{aligned}$$

  where $\varphi := \frac{1+\sqrt{5}}{2} \approx 1.618$ the golden ratio, $c := \frac{1}{\log_2 \varphi} \approx 1.44$, $b := \frac{c}{2}\log_2 5 - 2 \approx -0.328$, and $d := 1 + \frac{1}{\varphi^4\sqrt{5}} \approx 1.07$.

- a red–black tree's height is at most

$$h \leqq 2\log_2(n+1)$$ [10]

AVL trees are more rigidly balanced than red–black trees, leading to faster retrieval but slower insertion and deletion.

See also

- Trees
- Tree rotation
- Red–black tree
- Splay tree
- Scapegoat tree
- B-tree
- T-tree
- List of data structures

References

1. Eric Alexander. "AVL Trees".
2. Robert Sedgewick, *Algorithms*, Addison-Wesley, 1983, ISBN 0-201-06672-6, page 199, chapter 15: Balanced Trees.
3. Georgy Adelson-Velsky, G.; Evgenii Landis (1962). "An algorithm for the organization of information". *Proceedings of the USSR Academy of Sciences* (in Russian). **146**: 263–266. English translation by Myron J. Ricci in *Soviet Math. Doklady*, 3:1259–1263, 1962.
4. Pfaff, Ben (June 2004). "Performance Analysis of BSTs in System Software" (PDF). Stanford University.
5. AVL trees are not weight-balanced? (meaning: AVL trees are not μ-balanced?) (http://cs.stackexchange.com/questions/421/avl-trees-are-not-weight-balanced)
   Thereby: A Binary Tree is called μ-balanced, with $0 \leq \mu \leq \frac{1}{2}$, if for every node N, the inequality

   $$\frac{1}{2} - \mu \leq \frac{|N_l|}{|N|+1} \leq \frac{1}{2} + \mu$$

   holds and μ is minimal with this property. $|N|$ is the number of nodes below the tree with N as root (including the root) and N_l is the left child node of N.
6. Knuth, Donald E. (2000). *Sorting and searching* (2. ed., 6. printing, newly updated and rev. ed.). Boston [u.a.]: Addison-Wesley. p. 459. ISBN 0-201-89685-0.
7. More precisely: if the AVL balance information is kept in the child nodes – with meaning "when going upward there is an additional increment in height", this can be done with one bit. Nevertheless, the modifying operations can be programmed more efficiently if the balance information can be checked with one test.
8. Knuth, Donald E. (2000). *Sorting and searching* (2. ed., 6. printing, newly updated and rev. ed.). Boston [u.a.]: Addison-Wesley. p. 460. ISBN 0-201-89685-0.
9. Blelloch, Guy E.; Ferizovic, Daniel; Sun, Yihan (2016), "Just Join for Parallel Ordered Sets", *Proc. 28th ACM Symp. Parallel Algorithms and Architectures (SPAA 2016)*, ACM, pp. 253–264, doi:10.1145/2935764.2935768, ISBN 978-1-4503-4210-0.
10. Red–black tree#Proof of asymptotic bounds

Further reading

- Donald Knuth. *The Art of Computer Programming*, Volume 3: *Sorting and Searching*, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89685-0. Pages 458–475 of section 6.2.3: Balanced Trees.

External links

- This article incorporates public domain material from the NIST document: Black, Paul E. "AVL Tree". *Dictionary of Algorithms and Data Structures*.
- AVL tree demonstration (https://www.cs.usfca.edu/~galles/visualization/AVLtree.html) (HTML5/Canvas)
- AVL tree demonstration (http://www.qmatica.com/DataStructures/Trees/AVL/AVLTree.html) (requires Flash)
- AVL tree demonstration (http://www.strille.net/works/media_technology_projects/avl-tree_2001/) (requires Java)

The Wikibook *Algorithm Implementation* has a page on the topic of: ***AVL tree***

Wikimedia Commons has media related to ***AVL-trees***.