Information Processing Letters 116 (2016) 327–330

Contents lists available at ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl

Amortized rotation cost in AVL trees

Mahdi Amani [a], Kevin A. Lai [b,*], Robert E. Tarjan [b,c]

[a] Università di Pisa, Dipartimento di Informatica, Italy
[b] Princeton University, Computer Science Department, USA
[c] Intertrust Technologies, Sunnyvale, CA, USA

ARTICLE INFO

Article history:
Received 11 June 2015
Received in revised form 28 December 2015
Accepted 29 December 2015
Available online 31 December 2015
Communicated by M. Chrobak

Keywords:
Data structures
AVL tree
Lower bound
Rotation

1. Introduction

An AVL tree [1] is the original type of balanced binary search tree. An insertion in an n-node AVL tree takes at most two rotations, but a deletion in an n-node AVL tree can take $\Theta(\log n)$. A natural question is whether deletions can take many rotations not only in the worst case but in the amortized case as well. A sequence of n successive deletions in an n-node tree takes $O(n)$ rotations [3], but what happens when insertions are intermixed with deletions?

Haeupler, Sen, and Tarjan [2] conjectured that alternating insertions and deletions in an n-node AVL tree can cause each deletion to do $\Omega(\log n)$ rotations, but they provided no construction to justify their claim. We provide such a construction: we show that, for infinitely many n, there is a set E of *expensive* n-node AVL trees with the property that, given any tree in E, deleting a certain leaf and then reinserting it produces a tree in E, with the deletion having done $\Theta(\log n)$ rotations. One can do an arbitrary number of such expensive deletion-insertion pairs. The difficulty in obtaining such a construction is that in general the tree produced by an expensive deletion-insertion pair is not the original tree. Indeed, if the trees in E have even height k, $2^{k/2}$ deletion-insertion pairs are required to reproduce the original tree.

2. Definition and rebalancing of AVL trees

We begin with some terminology. A node in a binary tree is *binary*, *unary*, or a *leaf* if it has two, one, or no children, respectively. A unary node or a leaf has one or two *missing children*, respectively. We define the *depth* of a node in a binary tree recursively as 1 plus the depth of its parent, or 0 if it is the root. Similarly, we define the *height* of a node in a binary tree as 1 plus the maximum of the heights of its children, or 0 if it is a leaf. We adopt the convention that the height of a missing node is −1. The *height difference* of a non-root node is its parent's height minus its own height.

An *AVL tree* is a binary tree in which the heights of any two siblings differ by at most 1. Equivalently, it is a binary tree in which all height differences are 1 or 2. An inser-

* Corresponding author.
E-mail address: klai9@gatech.edu (K.A. Lai).

http://dx.doi.org/10.1016/j.ipl.2015.12.009
0020-0190/© 2015 Elsevier B.V. All rights reserved.

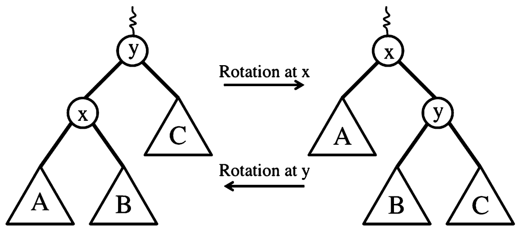

Fig. 1. Right rotation at node x. Triangles denote subtrees. The inverse operation is a left rotation at y.

tion or deletion of a node can destroy the AVL property. To restore it, we update the height information and if necessary rebalance the tree by doing *rotations*, which change the tree structure locally while preserving the symmetric order of nodes. Fig. 1 illustrates a rotation.

To specify exactly how height information is updated and when rotations are done, we introduce *node ranks*, as proposed by Haeupler, Sen, and Tarjan [2]. Each node x has an integer *rank* $x.r$, which is equal to the height of x except possibly during rebalancing, when $x.r$ can be the previous height of x. By convention, missing nodes always have rank -1, and leaves always have rank 0. The *rank difference* of a node is its parent's rank minus its own rank. We call a node an i, j *node* if the height differences of its children are i and j. This definition does not distinguish between left and right children. Node ranks equal node heights precisely when every node is a $1, j$ node for some $j > 0$, possibly a different j for each node. A binary tree with node ranks is an AVL tree if every node is a $1, 1$ node or a $1, 2$ node. We call this the *rank rule*. Henceforth we do not speak of heights, only ranks.

AVL trees grow by leaf insertions and shrink by deletions of leaves and unary nodes. To add a leaf to an AVL tree, replace a missing node by the new leaf and give the new leaf a rank of 0. If the parent of the new leaf was itself a leaf, it is now a $0, 1$ (unary) node, violating the rank rule. In this case, rebalance the tree by repeatedly applying the appropriate case in Fig. 2 until the rank rule holds.

A promotion (Fig. 2a) increases the rank of a node (x in Fig. 2a) by 1. We call the node whose rank increases the *promoted node*. Each promotion either creates a new violation at the parent of the promoted node or restores the rank rule and terminates rebalancing. Each single or double rotation (Figs. 2b and 2c, respectively) restores the rank rule and terminates rebalancing.

To delete a leaf in an AVL tree, replace it by a missing node; to delete a unary node, replace it by its only child (initially changing no ranks).[1] Such a deletion can violate the rank rule by producing a $2, 2$ or $1, 3$ node. In this case, rebalance the tree by applying the appropriate case in Fig. 3 until there is no violation. Each application of a case in Fig. 3 either restores the rank rule or creates a new violation at the parent of the previously violating node. Whereas each rotation case in insertion terminates

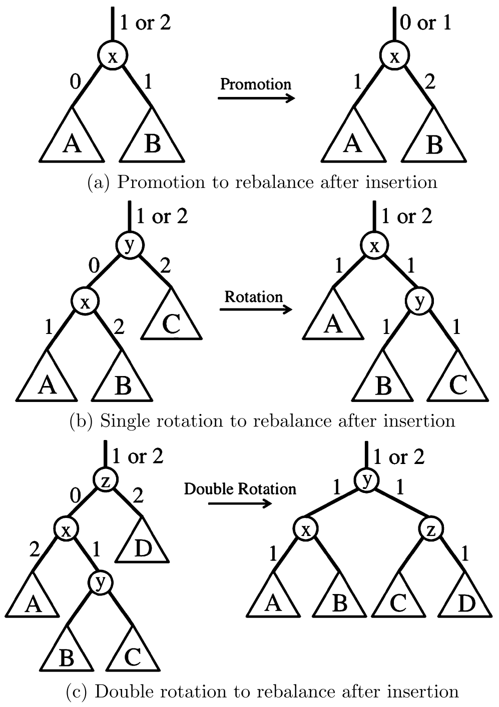
1 or 2
0
1
Promotion
0 or 1
2
(a) Promotion to rebalance after insertion
Rotation
(b) Single rotation to rebalance after insertion
Double Rotation
(c) Double rotation to rebalance after insertion
x
y
z
A
B
C
D

Fig. 2. Rebalancing cases after insertion. Numbers next to edges are rank differences. Rank differences of unmarked edges do not change. The promote step may repeat. All cases have mirror images.

rebalancing, the rotation cases in deletion can be nonterminating.

3. Construction of AVL trees

In order to obtain an initial tree in our expensive set E, we must build it from an empty tree. Thus the first step in our construction is to show that any n-node AVL tree can be built from an empty tree by doing n insertions. Although this result is easy to prove, we have not seen it in print before.[2]

Lemma 1. *A tree formed from an AVL tree by deleting a leaf of maximum depth is an AVL tree.*

Proof. Let T be an AVL tree, and suppose a leaf x of maximum depth is deleted. If the new tree is not an AVL tree, there is an ancestor y of x in T whose two children have heights differing by two or more. Let v be the child of y that is an ancestor of x (possibly x itself), and w the other child of y (possibly a missing node). Since x has maximum depth, the height of v in T must be no less than that of w. But deletion of x decreases the height of v by at most one, so its new height remains within one of that of w, a contradiction. □

[1] Our expensive examples only delete leaves. To delete a binary node x, swap x with its symmetric-order successor or predecessor and proceed as described in the text; the swap makes x a leaf or unary node.

[2] It also happens to be false for more relaxed types of balanced trees, such as weak AVL (wavl) trees [2]. Not all n-node wavl trees can be built from an empty tree by doing insertions only; many require a number of intermixed insertions and deletions exponential in n. This follows from an analysis using an exponential potential function like those in [2].