Rank-Balanced Trees

BERNHARD HAEUPLER, Massachusetts Institute of Technology
SIDDHARTHA SEN, Princeton University
ROBERT E. TARJAN, Princeton University & Microsoft Research

Since the invention of AVL trees in 1962, many kinds of binary search trees have been proposed. Notable are red-black trees, in which bottom-up rebalancing after an insertion or deletion takes $O(1)$ amortized time and $O(1)$ rotations worst-case. But the design space of balanced trees has not been fully explored. We continue the exploration. Our contributions are three. We systematically study the use of *ranks* and *rank differences* to define height-based balance in binary trees. Different invariants on rank differences yield AVL trees, red-black trees, and other kinds of balanced trees. By relaxing AVL trees, we obtain a new kind of balanced binary tree, the *weak AVL tree*, whose properties we develop. Bottom-up rebalancing after an insertion or deletion takes $O(1)$ amortized time and at most two rotations, improving the three or more rotations per deletion needed in all other kinds of balanced trees of which we are aware. The height bound of a weak AVL tree degrades gracefully from that of an AVL tree as the number of deletions increases, and is never worse than that of a red-black tree. Weak AVL trees also support top-down, fixed look-ahead rebalancing in $O(1)$ amortized time. Finally, we use exponential potential functions to prove that in weak AVL trees rebalancing steps occur exponentially infrequently in rank. Thus most of the rebalancing is at the bottom of the tree, which is crucial in concurrent applications and in those in which rotations take time that depends on the subtree size.

Categories and Subject Descriptors: E.1 [**Data**]: Data Structures—*Trees*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Sorting and searching*

General Terms: Algorithms, Theory

Additional Key Words and Phrases: balanced binary trees, exponential potential function, amortized complexity, AVL trees, red-black trees, search trees, data structures

1. INTRODUCTION

Balanced search trees are fundamental and ubiquitous in computer science. Since the invention of AVL trees [Adel'son-Vel'skii and Landis 1962] in 1962, many alternatives have been proposed, with the goal of simpler implementation or better performance or both. See *e.g.* [Andersson 1993; Bayer 1971; 1972; Bayer and McCreight 1972; Brown 1978; Guibas and Sedgewick 1978; Nievergelt and Reingold 1973; Olivié 1982; Aho et al. 1983; Sedgewick 2008]. Simpler implementations of balanced trees include Andersson's implementation [Andersson 1993] of Bayer's binary B-trees [Bayer 1971] and Sedgewick's related left-leaning red-black trees [Sedgewick 2008]. These data structures are asymmetric, which simplifies rebalancing by eliminating symmetric cases. Andersson further

A condensed preliminary version of this article appeared in *Proceedings of the 11th International Workshop on Algorithms and Data Structures* (*WADS*) (Banff, Canada, Aug. 21-23), Springer, Berlin-Heidelberg, 2009, pp. 351-362.
Author's addresses: Bernhard Haeupler, MIT Computer Science and Artificial Intelligence Lab, Cambridge, MA 02139, United States, haeupler@mit.edu. Siddhartha Sen, Department of Computer Science, Princeton University, Princeton, NJ 08540, United States, sssix@cs.princeton.edu. Robert E. Tarjan, Department of Computer Science, Princeton University, Princeton, NJ 08540, United States and HP Laboratories, Palo Alto, CA 94304, United States, ret@cs.princeton.edu.
Sen and Tarjan's research at Princeton University partially supported by NSF grants CCF-0830676 and CCF-0832797 and US-Israel Binational Science Foundation grant 2006204. The information contained herein does not necessarily reflect the opinion or policy of the federal government and no official endorsement should be inferred.
Tarjan's research while visiting Stanford University supported in part by an AFOSR MURI grant.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

simplified the implementation by factoring rebalancing into two procedures, *skew* and *split*, and by adding a few other clever ideas. On the other hand, standard red-black trees [Guibas and Sedgewick 1978], a representation of Bayer's symmetric binary B-trees [Bayer 1972], have update algorithms with better efficiency: rebalancing after an insertion or deletion takes $O(1)$ rotations worst-case and $O(1)$ time amortized [Tarjan 1983; 1985b]. As a result of these developments, one author [Skiena 1998, p. 177] has said, "AVL... trees are now passé."

Yet the design and analysis of balanced trees is a rich area, not yet fully explored. We continue the exploration. Our results include a new framework for defining height-based balance, a new kind of balanced binary tree, and a new way of tightly analyzing rebalancing. These results suggest that AVL trees are anything but passé. Our framework assigns a non-negative integer *rank* to each tree node, and imposes balance by restricting the rank differences between children and their parents; different rank rules give AVL trees, red-black trees, and other kinds of trees. In particular, a natural relaxation of AVL trees in our framework gives a new data structure, the *weak AVL tree* or *wavl tree*. Weak AVL trees have properties similar to those of red-black trees but better in several ways. If no deletions occur, a weak AVL tree is exactly an AVL tree; with deletions, its height is at most that of an AVL tree with the same number of insertions but no deletions. Weak AVL trees are a proper subset of red-black trees, with a different balance rule and different rebalancing algorithms. Insertion and deletion take at most two rotations in the worst case and $O(1)$ amortized time; red-black trees need three rotations in the worst case for a deletion. Indeed, we know of no other type of balanced binary tree in which deletions can be done in only two rotations. Insertion and deletion in wavl trees can be done top-down with fixed look-ahead in $O(1)$ amortized rebalancing time per update.

We introduce exponential potential functions to measure the amortized efficiency of operations on a balanced tree, and use them to show that rebalancing in weak AVL trees affects nodes exponentially infrequently in their heights, which is crucial in concurrent applications and in applications in which rotations take time that depends on subtree size. This is true of both bottom-up and top-down rebalancing. Mehlhorn and Tsakalidis [Mehlhorn and Tsakalidis 1986] proved a similar result for bottom-up rebalancing in AVL trees if only insertions are allowed, not deletions. (If deletions are allowed, rebalancing in AVL trees can take $\Omega(\log n)$ amortized time per update.) They used a multilevel credit method to obtain their result. Huddleston and Mehlhorn [Huddleston and Mehlhorn 1981; 1982] previously used this method to obtain similar results for "weak" B-trees (with red-black trees as a special case). Our approach uses exponential potential functions, a tool that unifies, generalizes, and simplifies the multilevel credit method, handles binary trees directly, and can be easily applied to other kinds of balanced trees.

Our paper is a rewritten, improved, and expanded version of a conference paper [Haeupler et al. 2009]. It contains eight sections in addition to this introduction. Section 2 contains our binary tree terminology. Section 3 presents our rank framework for specifying balance and uses it to define AVL trees, various kinds of red-black trees, and wavl trees. Section 4 discusses bottom-up rebalancing algorithms for wavl trees. Section 5 presents and analyzes top-down rebalancing algorithms with fixed look-ahead. Section 6 uses exponential potential functions to obtain inverse-exponential bounds on the number of rebalancing steps of a given rank. Section 7 presents a variant rebalancing method for deletion that improves some of our bounds. Section 8 compares AVL, red-black, and wavl trees. Section 9 summarizes our results and mentions a few open questions.

2. TREE TERMINOLOGY

A *binary tree* is an ordered tree in which each node x has a *left child* $left(x)$ and a *right child* $right(x)$, either or both of which may be missing. We denote a missing node by $null$. Missing nodes are also called *external*; non-missing nodes are *internal*. A node with no missing children, one missing child, or two missing children is *binary*, *unary*, or a *leaf*, respectively. Each node is the *parent* of its children. We denote the parent of a node x by $p(x)$; if x has no parent, $p(x) = null$, and x is the *root* of the tree. The *ancestor*, respectively *descendant* relationship is the reflexive, transitive closure of the parent, respectively child relationship. If node x is an ancestor of node y and $y \neq x$, x

is a *proper ancestor* of y and y is a *proper descendant* of x. If x is a node, its *left*, respectively *right* subtree is the binary tree containing all descendants of $left(x)$, respectively $right(x)$. The *size* $s(x)$ of a node x is its number of descendants, including itself. The *height* $h(x)$ of a node x is defined recursively by $h(x) = 0$ if x is a leaf, $h(x) = \max\{h(left(x)), h(right(x))\} + 1$ otherwise. The height h of a tree is the height of its root. The *depth* $d(x)$ of a node x is defined recursively by $d(x) = 0$ if x is the root, $d(x) = d(p(x)) + 1$ otherwise. The depth of a tree is the maximum of the depths of its leaves, which is equivalent to the tree's height.

We are most interested in binary trees as search trees. A binary search tree stores a set of *items*, each of which has a *key* selected from a totally ordered universe. We shall assume that each item has a distinct key; if not, we break ties by item identifier. In an *internal binary search tree*, each node contains an item, and the items are arranged in *symmetric order*: the key of the item in node x is greater, respectively less than those of all items in its left, respectively right subtree. Given such a tree and a key, we can search for the item having that key by comparing the key with that of the item in the root. If they are equal, we have found the desired item. If the search key is less, respectively greater than that of the item in the root, we search recursively in the left, respectively right subtree of the root. Each key comparison is a *step* of the search; the *current node* is the one whose item's key is compared with the search key. Eventually the search either locates the desired item or reaches a missing node, the left or right child of the last node reached by the search.

To insert a new item into such a tree, we first do a search on its key. When the search reaches a missing node, we replace this node with a node containing the new item. Deletion is a little harder. First we find the item to be deleted by doing a search on its key. If neither child of the node x containing the item is missing, we find either the next item or the previous item, by walking down through left, respectively right children of the right, respectively left child of x until reaching a node with a missing left, respectively right child. We swap the item to be deleted with the item found. Now the item to be deleted is in either a leaf or a unary node. In the former case, we replace the leaf by a missing node; in the latter case, we replace the unary node by its non-missing child. An access, insertion, or deletion takes $O(h + 1)$ time in the worst case, if h is the tree height.

An alternative kind of search tree is an *external binary search tree*: the external nodes contain the items, the internal nodes contain keys but no items, and all the keys are in symmetric order. We allow an internal node to have the same key as an external node. Every search proceeds all the way to an external node; when the search key and the key of an internal node are equal, the search proceeds in the left subtree of the node. To insert a new item, we do a search on its key. When the search reaches an external node, we replace it by an internal node having the old external node and a node containing the new item as its children, with the left child the one of smaller key, and with the new internal node containing this smaller key. To delete an item, we do a search on its key. When the search reaches the node containing the item, we delete this node and replace its parent by the other child of the parent. As in an internal search tree, an access, insertion, or deletion takes $O(h + 1)$ time worst-case. An external search tree needs one less than twice as many nodes as an internal search tree containing the same set of items, but deletion is simpler: swapping of items is unnecessary.

Henceforth by a binary tree we mean an internal binary search tree, with each node having pointers to its children. Our results extend to external binary search trees and to other binary tree data structures. We denote by n the number of nodes in the tree and by m and d, respectively, the number of insertions and the number of deletions in a sequence of intermixed searches, insertions, and deletions that starts with an empty tree. These numbers are related: $n = m - d$; that is, n is the number of nodes remaining in the tree after the sequence of operations.

To maintain balance in a binary tree, we need a restructuring primitive that preserves symmetric order (preserving the ability to search), changes the heights of certain nodes, and takes $O(1)$ time. We use the standard restructuring primitive, the (*single*) *rotation* shown in Figure 1: a rotation at a left child x with parent y makes y the right child of x while preserving symmetric order; a rotation at a right child is symmetric.

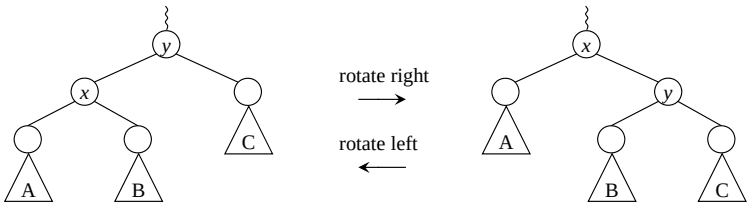

Fig. 1. Right rotation at node x. Triangles denote subtrees. The inverse operation is a left rotation at y.

3. RANK-BALANCED TREES

To make search, insertion, and deletion efficient, we keep the tree height logarithmic. We do this indirectly, by giving each node x an integer *rank* and imposing a *rank rule* that guarantees (i) the height of a node is at most a constant factor times its rank (possibly plus $\mathrm{O}(1)$), and (ii) the rank of a node is at most a constant factor times the logarithm of its size (possibly plus $\mathrm{O}(1)$). Different rank rules give different kinds of balanced binary trees. Although the notion of rank has been used previously to define height-based balance in binary trees, *e.g.*, in [Tarjan 1983], to our knowledge no one has explored the idea systematically. We do so here.

A *ranked binary tree* is a binary tree each of whose nodes x has a non-negative integer *rank* $r(x)$. We adopt the convention that missing nodes have rank -1. The *rank* of a ranked binary tree is the rank of its root. If x is a node with parent $p(x)$, the *rank difference* of x is $r(p(x)) - r(x)$. A non-root node is an *i-child* if its rank difference is i. A node is i, j if its left and right children have rank differences i and j, respectively. The definition of an i, j node does not distinguish between left and right children, and it allows children to be missing. For example, a leaf of rank zero is $1, 1$. All of our rank rules require that all rank differences be non-negative: we have not found a need for negative rank differences.

A *perfect* binary tree is one in which all leaves have equal depth k. Such a tree has size $2^{k+1} - 1$ and height k. If we give each node in such a tree a rank equal to its height, then all nodes are $1, 1$. This is the ideal situation, which we cannot achieve in a dynamically changing tree, not least because n is not necessarily one less than a power of two. To obtain balanced trees that can be updated efficiently, we allow rank differences other than 1. A generic rank rule that guarantees (i) and (ii) is: all rank differences are between 1 and c inclusive, where $c \geq 2$ is an integer constant. Another generic rank rule that guarantees (i) and (ii) is: (a) all rank differences are between 0 and c inclusive, and (b) no more than c' consecutive nodes along a path have rank difference zero, where $c \geq 1$ and $c' \geq 1$ are integer constants.

Moving from the generic to the specific, we present eight different rank rules, each of which is a restriction of one of the two generic rules above. The first gives AVL trees, the next six give different types of red-black trees, and the last gives a new kind of balanced tree.

AVL rule: Every node is $1, 1$ or $1, 2$.

The AVL rule gives the AVL trees [Adel'son-Vel'skii and Landis 1962]: the rank is the height (as it is for any rank rule that requires all ranks to be positive and each node to have at least one 1-child). The original definition of an AVL tree is that the heights of siblings are within one of each other; our definition is equivalent. The original representation of an AVL tree stores a ternary digit (trit) in each node indicating whether its two children have the same height, the left child is higher by one, or the right child is higher by one. Instead, we can store a bit in each child indicating whether its rank difference is 1 or 2. This pushes the balance information down a level, thereby reducing the storage needed from a trit to a bit per node. This representation, previously suggested by Brown [Brown 1978], emerges naturally from our framework. AVL trees need at most two rotations in the worst case to rebalance after an insertion, but $\mathrm{O}(\log n)$ to rebalance after a deletion.

The minimum number of nodes n_k in an AVL tree of rank k satisfies the recurrence $n_0 = 1, n_1 = 2, n_k = 1 + n_{k-1} + n_{k-2}$ for $k > 1$. This recurrence gives $n_k = F_{k+3} - 1$, where F_k is the k^{th}

Fibonacci number. Since $F_{k+2} \geq \phi^k$ [Knuth 1973], where $\phi = (1+\sqrt{5})/2$ is the golden ratio, $k \leq \log_\phi n \leq 1.4404 \lg n$, where lg is the base-two logarithm.

Two-Three Rule: Every node is $1,1$ or $0,1$, and no parent of a 0-child is a 0-child.

The two-three rule gives binarized 2-3 trees[Bayer and McCreight 1972; Aho et al. 1983]: a node having three children is represented by two binary nodes, one a child of the other. This rule is the natural analogue of the AVL rule, with rank difference 0 replacing rank difference 2.

Red-Black Rule: All rank differences are 0 or 1, and no parent of a 0-child is a 0-child.

The red-black rule relaxes the two-three rule by allowing $0,0$ nodes. It gives the standard version of red-black trees [Guibas and Sedgewick 1978], which are equivalent to the symmetric binary B-trees of Bayer [Bayer 1972]. These trees binarize 2-4 trees: a node having four children is represented by a binary node and its two children. In a ranked binary tree obeying the red-black rule, the 0-children are the red nodes, the 1-children are the black nodes. All missing nodes have rank difference 1 and are black. The rank of a node is the number of black nodes on a path from the node to a leaf, not counting the node itself: this number is independent of the path. Some authors require that the root of a red-black tree be black, others allow it to be either red or black. In our formulation, the root has no rank difference, and hence no color. Since all rank differences are 0 or 1, we can store the balance information in one bit per node, indicating whether its rank difference is zero (it is red) or one (it is black).

The two-three rule and the red-black rule allow the 0-child of a $0,1$ node to be either left or right, but we do not need both: if x is a left or right 0-child whose parent y is $0,1$, rotating at x without changing any ranks makes y a right or left 0-child, respectively, whose parent x is $0,1$, and preserves the two-three and red-black rules. Breaking the symmetry by disallowing a 0-child of a $0,1$ node to be left or right, respectively, gives us *right-leaning* or *left-leaning* two-three or red-black trees, defined by the following rank rules:

Right-Leaning Two-Three Rule: Every node is $1,1$ or $0,1$, no parent of a 0-child is a 0-child, and no 0-child is left.
Left-Leaning Two-Three Rule: Every node is $1,1$ or $0,1$, no parent of a 0-child is a 0-child, and no 0-child is right.
Right-Leaning Red-Black Rule: Every node is $1,1$ or $0,1$ or $0,0$, no parent of a 0-child is a 0-child, and no 0-child of a $0,1$ node is left.
Left-Leaning Red-Black Rule: Every node is $1,1$ or $0,1$ or $0,0$, no parent of a 0-child is a 0-child, and no 0-child of a $0,1$ node is right.

The right-leaning two-three rule gives the binary B-trees of Bayer [Bayer 1971], studied later by Andersson [Andersson 1993]. Sedgewick [Sedgewick 2008] studied left-leaning trees, both binarized 2,3 and red-black. Breaking the symmetry reduces the number of rebalancing cases in insertion and deletion. These cases can also be factored in a way that reduces the code length. See [Andersson 1993; Sedgewick 2008]. On the other hand, insertions and deletions in left-leaning or right-leaning binarized 2,3 or red-black trees require $\Omega(\lg n)$ rotations in the worst case. Allowing 0-children of $0,1$ nodes to be either left or right but not allowing $0,0$ nodes (the two-three rule) reduces the worst-case number of rotations for an insertion from $\mathrm{O}(\log n)$ to two; allowing $0,0$ nodes in addition (the red-black rule) reduces the worst-case number of rotations for a deletion from $\mathrm{O}(\log n)$ to three.

The minimum number of nodes n_k in a red-black tree of rank k satisfies $n_0 = 1$, $n_k \geq 2n_{k-1}+1$ for $k > 0$, which implies $n_k \geq 2^{k+1} - 1$. Hence $k \leq \lg n$. Also, the height of a node is at most twice its rank, so the height of a red-black tree of n nodes is at most $2 \lg n$. It is easy to construct a left-leaning binarized 2,3 tree of n nodes whose height is $2 \lg n - \mathrm{O}(1)$.

Our rank-based framework generalizes the *dichromatic framework* of Guibas and Sedgewick [Guibas and Sedgewick 1978]. They in effect allow rank differences of 0 and 1, and obtain specific kinds of balanced trees by adding appropriate additional restrictions. They map AVL trees into their framework by defining a node to be red if its height is even and that

of its parent is odd, and black otherwise. This maps every AVL tree to a red-black tree (one that satisfies the red-black rule), but the mapping is not onto, and Guibas and Sedgewick do not provide a sufficient condition for a red-black tree to be in the range of the mapping. They mention the alternative possibility of defining AVL trees using rank differences 1 and 2 as we have done, but they then dismiss it: "We have chosen to use zero weight links because the algorithms appear to be somewhat simpler." [Guibas and Sedgewick 1978]

On the contrary, we think that the best starting point for defining height-based balance is ranks, not rank differences, and that allowing rank differences 1 and 2 has merits beyond giving a nice definition of AVL trees. Indeed, it leads naturally to a new rank rule, which in turn gives a new kind of balanced tree. Specifically, we relax AVL trees in the same way that red-black trees relax binarized 2,3 trees: we allow non-leaf 2,2 nodes. This gives our new rank rule:

Weak AVL Rule: All rank differences are 1 or 2 and every leaf has rank 0.

We call a ranked binary tree that obeys the weak AVL rule a *weak AVL tree* or *wavl tree*. We represent the ranks by storing a bit in each child indicating whether its rank difference is 1 or 2. Weak AVL trees are in a way a hybrid of AVL and red-black trees in that they combine the good properties of both, as we shall see. The wavl trees with no $2,2$ nodes are exactly the AVL trees.

THEOREM 3.1. *If k, h, and n are the rank, height, and size of a wavl tree, respectively, then $h \leq k \leq 2h$, and $k \leq 2 \lg n$.*

PROOF. It is immediate by induction on n that $h \leq k \leq 2h$. The minimum size n_k of a wavl tree of rank k satisfies $n_0 = 1$, $n_1 = 2$, $n_k = 1 + 2n_{k-2}$ for $k \geq 2$. By induction, $n_k \geq 2^{\lceil k/2 \rceil}$, giving the second half of the theorem. □

4. BOTTOM-UP REBALANCING

In this section we describe bottom-up rebalancing algorithms for insertion and deletion in a wavl tree. Bottom-up insertion rebalancing is identical to AVL-tree insertion rebalancing; deletion rebalancing is similar to insertion rebalancing but has one extra case, or two counting symmetries.

A *promotion* of a node increases its rank by one; a *demotion* decreases it by one. When inserting a new node x into a wavl tree, we give it a rank of 0, making it $1,1$. Either the tree was previously empty; or the parent of the new node was previously a $1,2$ unary node, now a $1,1$ binary node; or the parent of the new node was previously a $1,1$ leaf, now a $0,1$ unary node. The third case violates the rank rule: the new node is a 0-child. In this case we rebalance the tree as follows (See Figure 2):

While $p(x) \neq null$ and $p(x)$ is $0,1$, repeat the following step:

Promote: Promote $p(x)$. Replace x by $p(x)$.

Now either the rank rule holds or $p(x)$ is $0,2$. If the rank rule does not hold (x is a 0-child), proceed as follows. Assume $x = left(p(x))$; the other possibility is symmetric. Let $z = p(x)$ and $y = right(x)$. Apply the appropriate one of the following two steps:

Rotate: y is $null$ or a 2-child. Rotate at x and demote z. This restores the rank rule.
Double Rotate: y is a 1-child. Rotate at y twice, making x its left child and z its right child. Promote y and demote x and z. This restores the rank rule.

During rebalancing there is exactly one violation of the rank rule: x is a 0-child. The rebalancing process, if it occurs, walks up the path from the newly inserted node, doing one or more promote steps followed by at most one rotate or double rotate step. After the first promote step, x is always $1,2$. The *rank* of a rebalancing step (a promote, rotate or double rotate) is the rank of $p(x)$ just before the step. The *rank* of an insertion is the rank of the last rebalancing step, or zero if there is no rebalancing.

Insertion with bottom-up rebalancing does not create any $2,2$ nodes (but it can destroy them). Thus a wavl tree built by starting with an empty tree and doing a sequence of insertions with bottom-

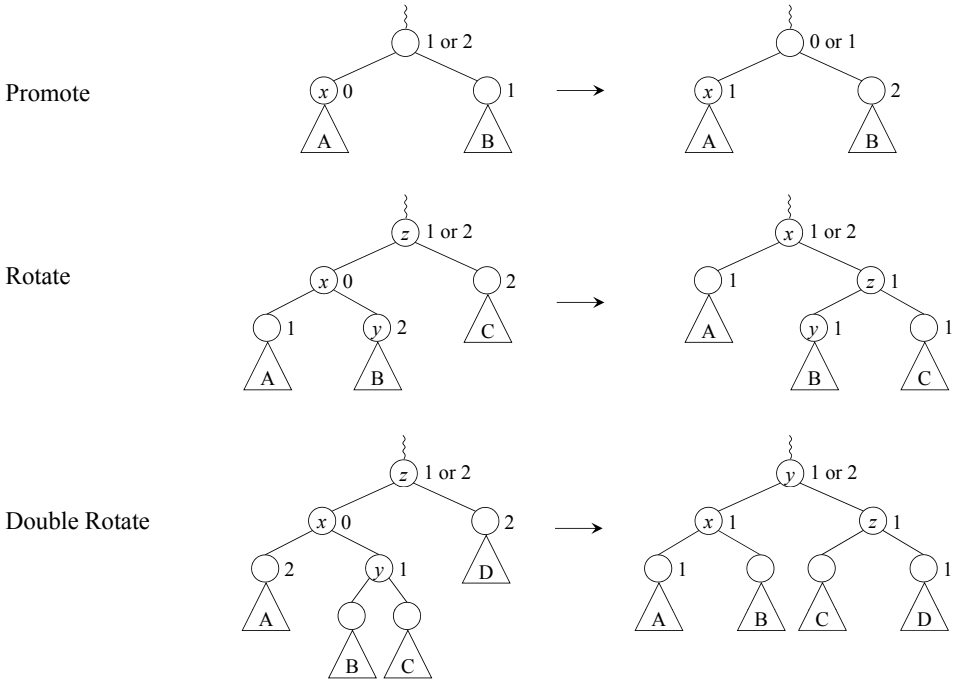

Fig. 2. Rebalancing steps after an insertion. Numbers next to nodes are their rank differences. The promote step may repeat. All cases have mirror images.

up rebalancing is an AVL tree. We introduce $2,2$ nodes to improve the efficiency of bottom-up deletion rebalancing and to support top-down rebalancing with fixed look-ahead (Section 5).

Deletion of a leaf in a wavl tree may convert its parent x, previously a $1,2$ node, into a $2,2$ leaf, violating the rank rule. In this case we begin the rebalancing by demoting x, which may make it a 3-child. Deletion of a unary node converts the child x that replaces it into a 2- or 3-child; the latter violates the rank rule. To do the rebalancing in this case, and to finish the rebalancing in the case of leaf deletion if the demotion of x leaves it a 3-child, we let y be the sibling of x and proceed as follows (see Figure 3):

While x is a 3-child and y is a 2-child or $2,2$, repeat the following step:

Demote: If y is a 2-child, demote $p(x)$; otherwise, demote both y and $p(x)$. In either case, let $x = p(x)$, and let y be the sibling of x.

Now either the rank rule holds, or $p(x)$ is $1,3$ and y is not $2,2$. If the rank rule does not hold (x is a 3-child), proceed as follows. Assume $x = \mathit{left}(p(x))$; the other possibility is symmetric. Let $z = p(x)$, $v = \mathit{left}(y)$, and $w = \mathit{right}(y)$. Apply the appropriate one of the following two steps:

Rotate: w is a 1-child. Rotate at y, promote y, and demote z. This restores the rank rule.
Double Rotate: w is a 2-child (so v is a 1-child). Rotate at v twice, making z its left child and y its right child. Promote v twice, demote y once, and demote z twice. This restores the rank rule.

During deletion rebalancing there is exactly one violation of the rank rule: x is a 3-child. The rebalancing process, if it occurs, walks up the path from the original x, doing one or more demote steps followed by at most one rotate or double rotate step. We call a demote step a *single demote* if it demotes only $p(x)$, a *double demote* if it demotes both y and $p(x)$. The *rank* of a rebalancing step

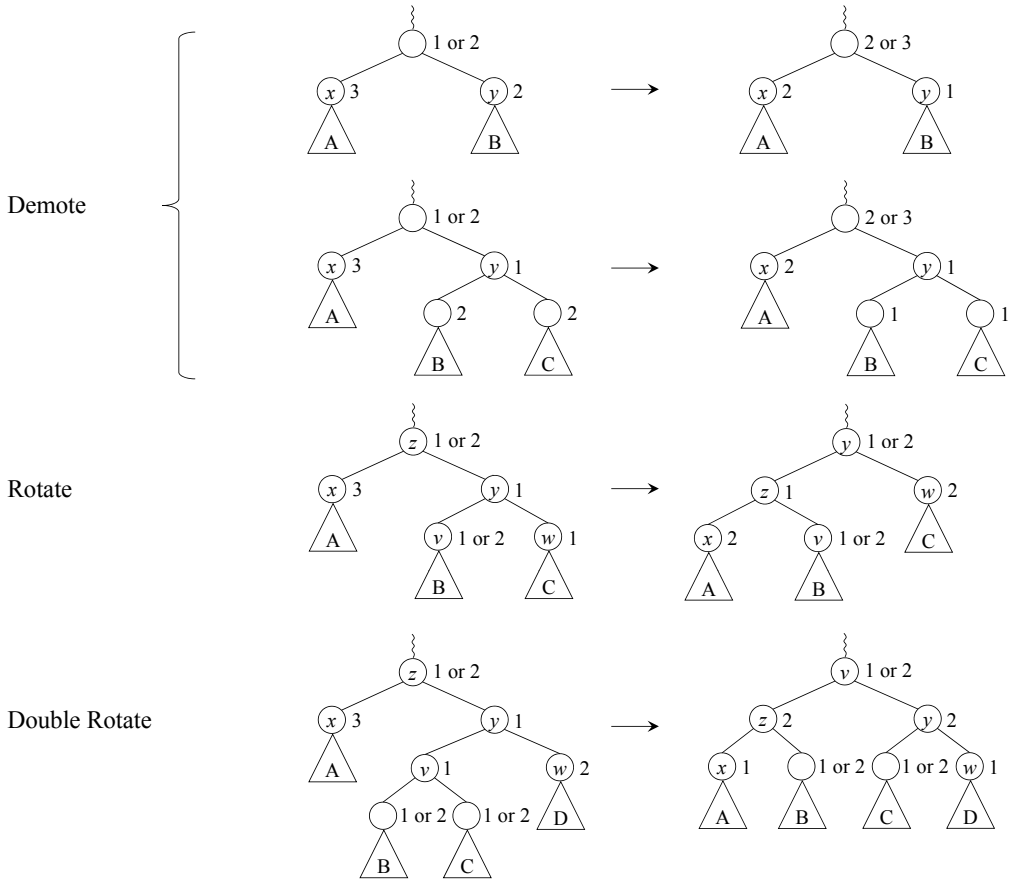

Fig. 3. Rebalancing steps after a deletion. Numbers next to nodes are their rank differences. The demote step may repeat. All cases have mirror images.

is the rank of $p(x)$ just before the step; the rank of a deletion is the rank of the last rebalancing step, or zero if there is no rebalancing.

Rebalancing after an insertion or deletion takes at most two rotations and $O(\log n)$ rank changes in the worst case. In a red-black tree, insertion rebalancing takes at most two rotations in the worst case, but deletion rebalancing can take three. Indeed, we know of no other kind of balanced binary tree in which rebalancing after a deletion takes at most two rotations. As is typical in balanced tree updating, deletion is more complicated than insertion, but only slightly: a promote step has only one case, but a demote step has two (single and double demote). The reason for the extra case is that the insertion cases depend on the states of the 0-child and its parent; after the first promotion, the 0-child is a $1, 2$ node in one of two states: its left or right child is its 1-child. In contrast, the deletion cases depend on the states of the sibling and the parent of the 3-child; the sibling can be in one of four states: $1, 1$, or $1, 2$ with a left or right 1-child, or $2, 2$. The rotate step actually handles two cases, reducing the number of extra cases from two to one (not counting mirror images, which double the number of cases).

The reason we have disallowed $2, 2$ leaves is that deleting a $2, 2$ leaf that is a 2-child creates a 4-child. Rebalancing after such a deletion takes up to four rotations in the worst case, not two.

Although rebalancing after an insertion or deletion takes $O(\log n)$ rank changes in the worst case, it takes only $O(1)$ amortized. To prove this, we do a potential-based amortized analysis [Tarjan

1985a]. To each configuration of the data structure we assign a numeric *potential*. We define the *amortized cost* of an operation to be its actual cost plus the increase in potential it causes. The total actual cost of a sequence of operations is then the total amortized cost plus the final potential minus the initial potential. If the initial potential is zero and the final potential is non-negative, the total amortized cost is an upper bound on the total actual cost. By making the potential well-defined even in the middle of rebalancing, when the rank rule is temporarily violated, we can analyze the effect of individual rebalancing steps directly.

In all our uses of this technique, we define the potential of a tree to be the sum of the potential of its nodes. We give each node a non-negative potential that depends on the rank differences of its children.

THEOREM 4.1. *In a wavl tree with bottom-up rebalancing, there are at most d demote steps over all deletions, where d is the number of deletions.*

PROOF. We define the potential of a $2,2$ or $2,3$ node to be 1, and that of all other nodes to be zero. The potential is initially zero and always non-negative. We define the cost of a rebalancing to be the number of demote steps it does. An insertion does no demote steps and creates no nodes that are $2,2$ or $2,3$, so its amortized cost is non-positive. A deletion that does no rebalancing increases the potential by at most one. A demote step other than the last in a deletion decreases the potential by one and hence has an amortized cost of zero. The last demote step in a deletion cannot increase the potential; a rotate or double rotate step in a deletion increases the potential by at most one. It follows that the amortized cost of a deletion is at most one. □

By Theorem 4.1, the total deletion rebalancing time in wavl trees is linear in the number of deletions, independent of the number of insertions. This is not true in red-black trees: rebalancing after the first deletion can take $\Theta(\log n)$ time.

THEOREM 4.2. *In a wavl tree with bottom-up rebalancing, there are at most $3m + 2d \leq 5m$ promote steps over all insertions, where m and d are the number of insertions and deletions, respectively.*

PROOF. We define the potential of a $1,1$ or $0,1$ node to be 1, and that of all other nodes to be zero. The potential is initially zero and always non-negative. We define the cost of a rebalancing to be the number of promote steps it does. A deletion does no promote steps, but it can create $1,1$ nodes, thereby increasing the potential. Specifically, each demote or double rotate step in a deletion increases the potential by at most one. By Theorem 4.1 there are at most d demote steps. There can be at most one double rotate step per deletion, for a total of d. Thus the total amortized cost of the deletions is at most $2d$.

An insertion that does no rebalancing increases the potential by at most one. In an insertion that does rebalancing, each promote step except the last one decreases the potential by one, for an amortized cost of zero. If the last promote step is the last rebalancing step (there are no rotations), it increases the potential by at most one, for an amortized cost of at most two. If the last promote step is followed by a rotate or double rotate, it does not increase the potential, and its amortized cost is one. A single rotate increases the potential by two, a double rotate by at most one. We conclude that the amortized cost of an insertion is at most three. □

As we have observed, a wavl tree built from an empty tree by doing only insertions is an AVL tree; hence its height is at most $\log_\phi n$, much smaller than the $2 \lg n$ bound of Theorem 3.1. Our next result generalizes the $\log_\phi n$ height bound to one that degrades gracefully as the number of deletions increases. The proof uses an idea similar to the exponential potential functions we use in Section 6 to obtain rank-based bounds on the number of rebalancing steps.

THEOREM 4.3. *With bottom-up rebalancing, a wavl tree has height at most $\log_\phi m$, where m is the number of insertions and ϕ is the golden ratio.*

PROOF. We define a *count* $c(x)$ for each node x, as follows: when x is first inserted, its count is 1. When a child is deleted, its count is added to that of its parent. The *total count* $C(x)$ of a node x is the sum of the counts of its descendants. This is equal to the sum of its count and the total counts of its children. The total count of the root is at most m, the number of insertions. (It can only be less than m if the root is deleted.) We prove by induction on the number of rebalancing steps that if a node x has rank k, $C(x) \geq F_{k+3} - 1$, from which it follows that $m \geq F_{k+3} - 1 \geq \phi^k$, giving the theorem.

We noted earlier that $F_{k+3} - 1$ satisfies the recurrence $x_0 = 1, x_1 = 2, x_k = 1 + x_{k-1} + x_{k-2}$ for $k > 1$. This gives $C(x) \geq F_{k+3} - 1$ if $k = 0$; $k = 1$; or $k > 1$, x is $0, 1$ or $1, 1$ or $1, 2$, and the inequality holds for both children of x. This implies that the inequality holds after insertion of a new leaf if it holds before, and after each insertion rebalancing step if it holds before: each node affected by the step is $1, 1$ or $1, 2$ after the step; since the inequality holds for its children, it holds for the node as well. The inequality holds after deletion of a node if it holds before, since the parent of a deleted child inherits its count. The demotion of a new leaf cannot violate the inequality, nor can the one or two demotions that occur during a demote step. A rotate or double rotate step can violate the inequality only at a node that becomes $2, 2$. In a rotate, y becomes $2, 2$, but it has the same rank and total count as z before the step, and hence satisfies the inequality. The same argument applies to v in a double rotate step. The only other node that becomes $2, 2$ is z in a rotate step if v is a 2-child. But x was demoted, either by the previous rebalancing step (a demote), or because the deletion made x a leaf. In either case x satisfied the inequality before its demotion, which implies by the recurrence that z satisfies the inequality after it becomes $2, 2$. □

The count used in the proof of Theorem 4.3 is history-based: it depends on the sequence of updates, not just on the current state of the tree. We do not know if such dependence can be avoided. Theorem 4.3 implies that if $d \leq (1 - \epsilon)m$, then $h \leq \log_\phi n + \log_\phi(1/\epsilon)$. That is, as long as the number of undeleted items is a fixed fraction of the total number of insertions, the height bound of a wavl tree is within an additive constant of that of an AVL tree, and smaller by a constant factor than that of a red-black tree. (The height bound of a wavl tree never exceeds that of a red-black tree, by Theorem 3.1.) Smaller height bounds are important in practice because they reduce the cost of a search, which affects all operations on the tree.

We conclude this section by discussing the implementation of rebalancing. The rebalancing process needs access to the affected nodes on the search path. There are several ways to provide such access. One is to add parent pointers to the tree. This uses extra space, three pointers per node instead of two, and increases the cost of rotations by a constant factor: six pointers change per rotation instead of three. Two pointers per node suffice if we use an alternative representation of a binary tree: each node points to its left child, or to its right child if its left child is missing; each left child points to its right sibling, or to its parent if its sibling is missing; and each right child points to its parent. This saves space but costs time.

Instead of adding or modifying pointers to support parental access, we can store the search path as the search proceeds from the root, either in a separate stack or by reversing child pointers along the path.

A third method is to maintain a *safe node* during the search. This node is the topmost node that will be affected by rebalancing. Metzger [Metzger 1975] and Samadi [Samadi 1976] used safe nodes to limit the amount of locking in a concurrent B-tree. Assume that all accesses proceed from the root, so that locking a node x prevents access by other processes to the entire subtree rooted at x. As an insertion search proceeds, it needs to maintain a lock only on the bottommost non-full node, which is the safe node. When the search encounters a new non-full node x, it locks x and unlocks the old safe node: any node splitting caused by the insertion will not propagate above x. A similar idea applies to deletions.

We apply this idea to binary trees and use it for a slightly different purpose: to avoid the need for parent pointers or a stack to do rebalancing. In wavl tree insertion, the safe node is either the root or the parent of the last node reached by the search that is a 2-child or a 1,2 node. We initialize the safe

node to be the root and change it to the parent of the current node of the search each time the current node is a 2-child, or is a 1,2 node other than the root. In wavl tree deletion, the safe node is either the root, or the parent of the last node reached by the search that is a 1-child, or is a 1,2 node whose 1-child is not a 2,2 node. We initialize the safe node to be the root and change it to the parent of the current node each time this node is a 1-child, or is a 1,2 node whose 1-child is not a 2,2 node. In either an insertion or a deletion, once the search reaches the bottom of the tree, we do appropriately modified rebalancing steps top-down starting from the safe node. This method needs only $\mathrm{O}(1)$ extra space, but it incurs additional overhead during the search and during the rebalancing, to maintain the safe node and to determine the next node on the search path, respectively. Its advantages are that it can avoid the need for parent pointers or a stack to do rebalancing, it provides the minimum context needed for locking if searches and updates are concurrent (during an insertion or deletion, lock each new safe node and unlock the old one), and it extends to support top-down rebalancing with fixed look-ahead, as we discuss next.

5. TOP-DOWN REBALANCING

If we use a safe node to support rebalancing and change the rebalancing method slightly, we can do the rebalancing top-down with fixed look-ahead. This significantly improves the concurrency of the tree, because the critical section of an insertion encompasses only $\mathrm{O}(1)$ nodes at any time. If the fixed look-ahead is sufficiently large, the amortized number of rebalancing steps per update is $\mathrm{O}(1)$ (although the worst-case number of rotations per update becomes $\Theta(\log n)$). The idea is to force a reset of the safe node after $\mathrm{O}(1)$ search steps. In an insertion, if the current node of the search and its parent are both $1,1$, we can force a reset on the next search step by promoting the current node and rebalancing top-down from the safe node. (The first rebalancing step will promote the parent of the current node.) In a deletion, if the current node is $2,2$, or it is $1,2$ and its 1-child is $2,2$, we can force a reset on the next search step by demoting the current node in the former case, or the current node and its 1-child in the latter, and rebalancing top-down from the safe node. With top-down rebalancing, the *rank* of an insertion or deletion is the highest rank of a rebalancing step, or zero if there is no rebalancing.

Forcing a reset as often as possible minimizes the lookahead. But if we force a reset less often we can guarantee $\mathrm{O}(1)$ amortized rebalancing steps per update. Since forced resets during insertions can create $2,2$ nodes, we can no longer analyze deletions separately from insertions; we analyze both using one potential function.

THEOREM 5.1. *If rebalancing is top-down with a forced reset during insertion at the fifth* $1,1$ *node in a row and during deletion at the third node in a row that is* $2,2$*, or* $1,2$ *with a 1-child that is* $2,2$*, then the number of rebalancing steps is* $\mathrm{O}(m+d)$.

PROOF. We need a potential function such that each forced reset of the safe node reduces the potential. We define the potential of a $1,1$ or $0,1$ node to be 1, that of a $2,2$ or $2,3$ node to be $8/3$, and that of all other nodes to be zero. In an insertion, if a search step does not do a reset, every node along the search path from the grandchild of the safe node to the parent of the current node is $1,1$. If we force a reset after five search steps that do not do a reset (by promoting the fifth $1,1$ node in a row), the corresponding rebalancing reduces the potential by at least $1/3$: the bottom $1,1$ node becomes $2,2$, increasing the potential by $5/3$, each of the other four $1,1$ nodes becomes $1,2$, decreasing the potential by four, and the last rebalancing step increases the potential by at most two. (The last rebalancing step may create a new $1,1$ node, but the analysis accounts for this.) In a deletion, if a search step does not do a reset, every node along the search path from the grandchild of the safe node to the parent of the current node is either $2,2$, or it is $1,2$ and its 1-child is $2,2$. If we force a reset after three search steps that do not do a reset (by demoting the third node in a row that is either $2,2$, or $1,2$ with a 1-child that is $2,2$, and demoting its 1-child if it has one), the corresponding rebalancing reduces the potential by at least $1/3$: the initial demotion or pair of demotions decrease the potential by at least $2/3$, each of the two subsequent demote steps decreases it by at least $5/3$, and the last rebalancing step increases it by at most $11/3$. A forced reset during

either an insertion or deletion takes $O(1)$ time. If we scale this time to be at most one, then a forced reset takes non-positive amortized time. In an insertion or deletion, any rebalancing at the bottom of the search path takes $O(1)$ amortized time. □

One disadvantage of top-down rebalancing is that the proof of Theorem 4.3 is no longer valid: the induction does not apply to the $2, 2$ nodes created by forced resets during insertions.

6. RANK-BASED AMORTIZED ANALYSIS

The amortized analysis of rebalancing in Sections 4 and 5 implies that most rebalancing is low in the tree: if the rebalancing is bottom-up (Section 4), the number of insertions of rank k is $O(m/k)$, the number of deletions of rank k is $O(d/k)$; if the rebalancing is top-down (Section 5), the total number of insertions and deletions of rank k is $O(m/k)$. But something much stronger is true: the number of rebalancing steps of rank k is exponentially small in k. Thus most rebalancing occurs at the very bottom of the tree. This is crucial in at least two situations: (1) The tree is accessed concurrently. Searches, which are read-only, need not block each other, but insertions, deletions, and rebalancing change the tree and must block other operations. The lower in the tree rebalancing occurs, the less contention. (2) Rotations take time that is not $O(1)$ but a function of subtree size. This occurs in certain data structures for multidimensional search problems, such as priority search trees [McCreight 1985].

Such a result holds in weight-balanced trees [Nievergelt and Reingold 1973] for rotations [Blum and Mehlhorn 1980; Mehlhorn 1984] but not for size updates, which propagate all the way to the root on each insertion or deletion. Mehlhorn and Tsakalidis [Mehlhorn and Tsakalidis 1986] proved such a result for bottom-up rebalancing in AVL trees if only insertions are allowed, not deletions; if deletions are allowed, rebalancing can propagate all the way to the root on each insertion or deletion. Huddleston and Mehlhorn [Huddleston and Mehlhorn 1981; 1982] proved such a result for "weak" B-trees, which include 2-4 trees as a special case. Their result translates to a similar result for red-black trees via the standard binarization described in Section 3, as we discuss in Section 8. We prove such a result for wavl trees with either bottom-up or top-down rebalancing by using a direct potential-based analysis. Both Mehlhorn and Tsakalidis and Huddleston and Mehlhorn used a credit-based analysis, with different credit accounts for each node height. Our approach is to give each node a potential that is exponential in its rank. In addition to handling rotations directly, our method simplifies and unifies the multilevel credit method, and can easily be applied to other kinds of balanced trees.

We begin by analyzing bottom-up rebalancing. First we consider the special case in which there are no deletions, only insertions.

THEOREM 6.1. *In a wavl tree with bottom-up rebalancing and no deletions, the number of insertion rebalancing steps of rank k is $O(m/\phi^k)$, where m is the number of insertions and ϕ is the golden ratio.*

PROOF. We define the potential of a $1, 1$ or $0, 1$ node of rank j to be ϕ^j, and that of all other nodes to be zero. Consider the effect of an insertion of rank j on the potential. Inserting a leaf increases the potential by one. A non-terminal promote step of rank i converts a $0, 1$ node of rank i into a $1, 2$ node, reducing the potential by ϕ^i. Successive rebalancing steps differ by one in rank. Thus the last non-terminal promote step is of rank $j-1$, and the entire sequence of such steps reduces the potential by at least $\phi^{j-1} \sum_{i=0}^{\infty} 1/\phi^i - O(1) = \phi^j/(\phi - 1) - O(1) = \phi^{j+1} - O(1)$, since $\phi - 1 = 1/\phi$. A terminal rebalancing step of rank j increases the potential by at most $\phi^{j+2} - \phi^j = \phi^{j+1}$ if it is a promote, and by at most $\phi^j + \phi^{j-1} = \phi^{j+1}$ if it is a rotate or double rotate, since $\phi^2 - \phi - 1 = 0$. Combining these estimates, we find that an insertion of rank j increases the potential by at most $\phi^{j+1} - \phi^{j+1} + O(1) = O(1)$.

Now we truncate the potential function. For k fixed, redefine the potential of all nodes of rank k or greater to be zero. Each rebalancing step has the same effect on the potential as estimated above, with the following exceptions: a non-terminal promote of rank k or greater does not increase the

potential; a terminal promote of rank $k-2$ or greater does not increase the potential; and a rotate or double rotate of rank k or greater increases the potential by at most ϕ^{k-1}. It follows that an insertion of rank $k-1$ or less increases the potential by $\mathrm{O}(1)$ or reduces it, since the estimate above remains valid, but an insertion of rank k or greater reduces the potential by at least $\phi^k - \mathrm{O}(1)$ as a result of the nonterminal promote steps, minus ϕ^{k-1} as a result of the terminal step, totaling $\phi^{k-2} - \mathrm{O}(1)$. Since the potential is always non-negative, there are $\mathrm{O}(m/\phi^k)$ insertion rebalancing steps of rank k, one per insertion of rank k or greater. □

Since insertions in AVL trees are exactly the same as in wavl trees, Theorem 6.1 also holds for AVL trees. A similar result was proved by Mehlhorn and Tsakalidis [Mehlhorn and Tsakalidis 1986], who analyzed bottom-up rebalancing in AVL trees when only insertions are allowed.

Next we consider the general case of arbitrarily intermixed insertions and deletions. As in Section 4, we can analyze deletions separately from insertions, since insertions do not create $2,2$ nodes. Let $b_1 = 1.3247...$ be the *plastic constant* [Laan 1997], the unique real root of $b_1^3 - b_1 - 1 = 0$.

THEOREM 6.2. *With bottom-up rebalancing, the number of deletion rebalancing steps of rank k is $\mathrm{O}(d/b_1^k)$.*

PROOF. We define the potential of a $2,2$ or $2,3$ node of rank j to be b_1^j, and the potential of all other nodes to be zero. Insertions do not increase the potential, since they create no nodes of positive potential. Consider the effect of a deletion of rank j on the potential. Deleting a leaf or unary node increases the potential by $\mathrm{O}(1)$, as does demoting a $2,2$ leaf. A non-terminal single demote step of rank i converts a $2,3$ node of rank i into a $1,2$ node, reducing the potential by b_1^i. A non-terminal double demote step of rank i converts a $2,2$ node of rank $i-1$ into a $1,1$ node, reducing the potential by $b_1^{i-1} < b_1^i$. Successive rebalancing steps differ in rank by 2. Thus the last non-terminal demote step is of rank $j-2$, and the entire sequence of such steps reduces the potential by at least $b_1^{j-3}\sum_{i=0}^{\infty} 1/b_1^{2i} - \mathrm{O}(1) = b_1^{j-1}/(b_1^2-1) - \mathrm{O}(1)$. A terminal rebalancing step of rank j increases the potential by at most $b_1^{j+1} - b_1^{j-1}$ if it is a demote, by at most b_1^{j-1} if it is a rotate, and by at most $b_1^j > \max\{b_1^{j+1} - b_1^{j-1}, b_1^{j-1}\}$ if it is a double rotate. Thus the entire deletion increases the potential by at most $b_1^j - b_1^{j-1}/(b_1^2-1) + \mathrm{O}(1) = b_1^{j-1}(b_1^3 - b_1 - 1)/(b_1^2-1) + \mathrm{O}(1) = \mathrm{O}(1)$, since $b_1^3 - b_1 - 1 = 0$.

Now we truncate the potential function. For k fixed, redefine the potential of all nodes of rank k or greater to be zero. Each rebalancing step has the same effect on the potential as estimated above, with the following exceptions: non-terminal demotes of rank k or greater, terminal demotes of rank $k-1$ or greater, rotates of rank $k+1$ or greater, and double rotates of rank k or greater do not increase the potential; rotates of rank k increase the potential by at most b_1^{k-1}, as estimated above. It follows that a deletion of rank $k-1$ or less increases the potential by $\mathrm{O}(1)$ or reduces it, since the estimate above remains valid, but a deletion of rank k reduces the potential by at least $b_1^{k-1}/(b_1^2-1) - b_1^{k-1} - \mathrm{O}(1) = b_1^{k-1}(2-b_1^2)/(b_1^2-1) - \mathrm{O}(1)$, and a deletion of rank greater than k reduces the potential by at least $b_1^{k-1}/(b_1^2-1) - \mathrm{O}(1)$. Since $b_1^2 - 1 > 0$ and $2 - b_1^2 > 0$, a deletion of rank k or greater reduces the potential by $\Omega(b_1^k) - \mathrm{O}(1)$. Since the potential is always non-negative, there are $\mathrm{O}(m/b_1^k)$ deletion rebalancing steps of rank k, at most one per deletion of rank k or greater. □

We can combine the proofs of Theorems 6.1 and 6.2 to obtain a bound of $\mathrm{O}((m+kd)/b_1^k)$ on the number of bottom-up insertion rebalancing steps of rank k if there are intermixed deletions. To do this, we define the potential of a $1,1$ or $0,1$ node of rank $j < k$ to be b_1^j, and that of all other nodes to be zero. A deletion of rank j increases the potential by $\mathrm{O}(\min\{b_1^j, b_1^{k-1}\})$, since a deletion rebalancing step of rank i can produce a $1,1$ node of rank i or $i-1$. By Theorem 6.2, the total increase in potential caused by deletions is $\mathrm{O}(kd)$. An analysis like that in the proof of Theorem 6.1 but with b_1 in place of ϕ shows that each insertion increases the potential by $\mathrm{O}(1)$ or decreases it, and each insertion of rank k or greater decreases it by $\Omega(b_1^k) - \mathrm{O}(1)$, giving the result.

By giving positive potential to more nodes, we can eliminate the kd term in this estimate.

THEOREM 6.3. *With bottom-up rebalancing, the number of insertion rebalancing steps of rank k is $O(m/b_1^k)$.*

PROOF. We define the potential of a node of rank j be b^j if it is $1,1$ or $0,1$, and ab^j otherwise, where a and b are constants to be chosen later, such that $b > 1$ and $0 \le a < 1/b$. Consider the effect of an insertion of rank j on the potential. Inserting a leaf increases the potential by one. A promote step of rank i increases the potential by $ab^{i+1} - b^i$. The sequence of non-terminal promote steps ends with one of rank $j-1$ and altogether increases the potential by at most $(ab-1)b^{j-1}\sum_{i=0}^{\infty} 1/b^i + O(1) = (ab-1)b^j/(b-1) + O(1)$. A terminal promote of rank j increases the potential by at most $ab^{j+1} - b^j + b^{j+2} - ab^{j+2}$. A rotate or double rotate of rank j increases the potential by at most $b^j + b^{j-1} - 2ab^j$.

If the last step is a promote, the entire insertion increases the potential by at most

$$\begin{aligned}
&\frac{(ab-1)b^j}{b-1} + (ab-1)b^j + b^{j+2} - ab^{j+2} + O(1)\\
=\ &\frac{(ab-1)b^{j+1}}{b-1} + b^{j+2} - ab^{j+2} + O(1)\\
=\ &\frac{b^{j+1}(ab-1+b(1-a)(b-1))}{b-1} + O(1)\\
=\ &\frac{b^{j+1}(2ab-ab^2+b^2-b-1)}{b-1} + O(1).
\end{aligned}$$

If $a \le (1+b-b^2)/(b(2-b))$, the potential increase is $O(1)$ or negative.

If the last step is a rotate or double rotate, the entire insertion increases the potential by at most

$$\begin{aligned}
&\frac{(ab-1)b^j}{b-1} + b^j + b^{j-1} - 2ab^j + O(1)\\
=\ &\frac{b^{j-1}(2ab-ab^2+b^2-b-1)}{b-1} + O(1).
\end{aligned}$$

This gives us exactly the same constraint as in the case of a terminal promote: if $a \le (1+b-b^2)/(b(2-b))$, the potential increase is $O(1)$ or negative.

Observe that choosing $b = \phi$ and $a = 0$ satisfies the constraint above as well as $b > 1$ and $0 \le a < 1/b$, and gives the potential function we used in the proof of Theorem 6.1.

Now consider the effect of a deletion of rank j on the potential. Deleting a leaf or unary node, or demoting a leaf of rank 1, increases the potential by $O(1)$. A single demote step of rank i increases the potential by $ab^{i-1} - ab^i$. A double demote step of rank i increases the potential by $ab^{i-1} - ab^i + b^{i-2} - ab^{i-1} = b^{i-2} - ab^i > ab^{i-1} - ab^i$ since $a < 1/b$. The sequence of non-terminal demote steps ends with one of rank $j-2$ and altogether increases the potential by at most $(1-ab^2)b^{j-4}\sum_{i=0}^{\infty} 1/b^{2i} + O(1) = b^{j-2}(1-ab^2)/(b^2-1) + O(1)$. A terminal single demote step does not increase the potential, nor does a rotate. A terminal double demote of rank j increases the potential by at most $b^{j-2} - ab^j$. A double rotate of rank j increases the potential by at most $b^{j-2} - ab^{j-1} \ge b^{j-2} - ab^j$. Thus the entire deletion increases the potential by at most

$$\begin{aligned}
&\frac{b^{j-2}(1-ab^2)}{b^2-1} + b^{j-2} - ab^{j-1} + O(1)\\
=\ &\frac{b^{j-2}(1-ab^2+b^2-1-ab^3+ab)}{b^2-1} + O(1)\\
=\ &b^{j-1}(b-ab^2-ab+a)/(b^2-1) + O(1).
\end{aligned}$$

If $a \geq b/(b^2+b-1)$, the potential increase is $\mathrm{O}(1)$ or negative.

Combining the upper and lower bounds on a gives

$$\begin{aligned} & \frac{b}{b^2+b-1} \leq a \leq \frac{1+b-b^2}{b(2-b)} \\ \Longrightarrow \qquad & 2b^2-b^3 \leq (b^2+b-1)(1+b-b^2) \\ & b^4-b^3-b^2+1 \leq 0 \\ & (b-1)(b^3-b-1) \leq 0. \end{aligned}$$

This implies $b^3-b-1 \leq 0$ since $b > 1$. Choosing $b = b_1$ maximizes b subject to this inequality and forces the choice $a = (1+b_1-b_1^2)/(b_1(2-b_1))$. Since $b_1^3 = b_1+1$, $a = b_1(b_1-1)/(2-b_1)$. It is straightforward to verify that $a < 1/b_1$.

Now we truncate the growth of the potential function. For k fixed, redefine the potential of a $0,1$ or $1,1$ node of rank k, and of any node of rank greater than k, to be ab^{k+1}. Each insertion rebalancing step has the same effect on the potential as estimated above, with the following exceptions. A promote step of rank k or greater does not change the potential. A promote step of rank $k-1$ increases the potential by $ab^k - b^{k-1}$ whether or not it is terminal. A terminal promote step of rank $k-2$ increases the potential by at most $ab^{k+1} - ab^k + ab^{k-1} - b^{k-2}$, less than the estimate above since $a < 1/b$. A rotate or double rotate of rank $k+1$ or greater does not increase the potential. A rotate or double rotate of rank k increases the potential by at most $ab^{k+1} + b^{k-1} - 2ab^k$, less than the estimate above by $b^k - ab^{k+1} = (1-ab)b^k$. It follows from the analysis above that an insertion increases the potential by $\mathrm{O}(1)$ or decreases it, an insertion of rank k that ends in a rotate or double rotate decreases the potential by at least $(1-ab)b^k - \mathrm{O}(1)$, and an insertion of rank greater than k, or one of rank k that ends in a promotion, decreases the potential by at least $(1-ab)b^k/(b-1) - \mathrm{O}(1)$. Since $a < 1/b$, any insertion of rank k or greater decreases the potential by $\Omega(b^k) - \mathrm{O}(1)$.

Each deletion rebalancing step has the same effect on the potential as estimated above, with the following exceptions: a single demote, a double demote, or a double rotate of rank $k+2$ or greater does not change the potential. It follows from the analysis above that a deletion increases the potential by $\mathrm{O}(1)$ or decreases it.

Since the potential is always non-negative, there are $\mathrm{O}(m/b^k)$ insertion rebalancing steps of rank k, one per insertion of rank k or greater. □

We conclude this section with a rank-based analysis of top-down rebalancing. We use a single potential function and analyze insertions and deletions together. Let $b_2 > 1$ and $a_2 \geq 0$ be constants to be specified later. In the analysis below, setting $a_2 = 2.879...$ and $b_2 = 1.053...$ yields the best result.

THEOREM 6.4. *If rebalancing is top-down with a forced reset during insertion at the fifth* $1,1$ *node in a row and during deletion at the third node in a row that is* $2,2$*, or* $1,2$ *with a 1-child that is* $2,2$*, then the number of rebalancing steps of rank* k *is* $\mathrm{O}(m/b_2^k)$.

PROOF. We define the potential of a node of rank j to be b_2^j if it is $1,1$ or $0,1$; $a_2 b_2^j$ if it is $2,2$ or $2,3$; and zero otherwise. To determine the effect of a forced reset on the potential, we estimate the effect of the topmost step and combine this with the effect of the initial promotion or demotion(s) and the cumulative effect of the non-terminal promote or demote steps. We begin with insertions. Consider a forced reset that begins by promoting a node of rank i and whose topmost rebalancing step is of rank j. Since the forced reset begins by promoting the fifth $1,1$ node in a row, the topmost rebalancing step is either a promote of rank $j = i+4$ (of the topmost $1,1$ node in a row), or a rotate or double rotate of rank $j = i+5$. The initial promotion increases the potential by $a_2 b_2^{i+1} - b_2^i$. A non-terminal promote step of rank k decreases the potential by b_2^k; these steps are of ranks $i+1$ through $j-1$ inclusive. If the topmost step is a promote, it increases the potential by at most $b_2^{j+2} - b_2^j$; the worst case is when the parent of the promoted node is $1,2$ and becomes $1,1$. If the

topmost step is a rotate or double rotate, it increases the potential by at most $b_2^j + b^{j-1}$. Overall the forced reset increases the potential by at most $b_2^{j+2} - b_2^j - b_2^{j-1} - b_2^{j-2} + (a_2 - 1)b_2^{j-3} - b_2^{j-4}$ if the topmost step is a promote, at most $b_2^j - b_2^{j-2} - b_2^{j-3} + (a_2 - 1)b_2^{j-4} - b_2^{j-5}$ if the topmost step is a rotate or double rotate.

We do a similar analysis of deletions. Consider a forced reset that begins by demoting a node of rank i, and its 1-child if it has one, and whose topmost rebalancing step is of rank j. Since the forced reset begins by demoting the third node in a row that is $2, 2$, or $1, 2$ with a 1-child that is $1, 1$, the topmost rebalancing step is either a demote of rank $j = i + 4$, or a rotate or double rotate of rank $j = i + 6$. If the initialization demotes only one node, it increases the potential by $b_2^{i-1} - a_2 b_2^i$. If it demotes two nodes, it increases the potential by $b_2^{i-2} - a_2 b_2^{i-1} + b_2^{i-1} > b_2^{i-1} - a_2 b_2^i$. A non-terminal demote step of rank k decreases the potential by $a_2 b_2^k$ if it is a single demote. If it is a double demote, it increases the potential by $b_2^{k-2} - a_2 b_2^{k-1}$, more than a single demote. The non-terminal steps are of every other rank from $i + 2$ to $j - 2$ inclusive. If the topmost step is a single demote, it increases the potential by at most $a_2 b_2^{j+1} - a_2 b_2^j$; the worst case is when the parent of the demoted node is $1, 2$ and becomes $2, 2$. The same worst case applies if the topmost step is a double demote, but now the potential increases by at most $b_2^{j-2} - a_2 b_2^{j-1} + a_2 b_2^{j+1} > a_2 b_2^{j+1} - a_2 b_2^j$. If the topmost step is a rotate, it increases the potential by at most $a_2 b_2^{j-1}$. If it is a double rotate, it increases the potential by at most $a_2 b_2^j + b_2^{j-2}$, more than a single rotate. Thus the largest potential increase occurs after an initialization that demotes two nodes, followed by non-terminal double demote steps of every other rank from $i + 2$ to $j - 2$ inclusive, followed by a topmost step that is either a double demote or a double rotate. Overall the forced reset increases the potential by at most $a_2 b_2^{j+1} - a_2 b_2^{j-1} + b_2^{j-2} - a_2 b_2^{j-3} + b_2^{j-4} + (1 - a_2)b_2^{j-5} + b_2^{j-6}$ if the topmost step is a double demote, at most $a_2 b_2^j + b_2^{j-2} - a_2 b_2^{j-3} + b_2^{j-4} - a_2 b_2^{j-5} + b_2^{j-6} + (1 - a_2)b_2^{j-7} + b_2^{j-8}$ if the topmost step is a double rotate.

We choose a_2 and b_2 to maximize b_2 subject to the constraint that no forced reset increases the potential. That is, all four quantities above (two each for a forced reset during an insertion and a deletion, respectively) remain non-positive. Solving numerically gives $a_2 = 2.879...$ and $b_2 = 1.053...$.

There are at most six rebalancing steps of rank less than six per insertion or deletion, including all those outside of forced resets. A forced reset that starts by promoting or demoting a node of rank k does at most six rebalancing steps, all of rank greater than k. We prove that for any k there are $\mathrm{O}(m/b_2^k)$ forced resets that start by promoting or demoting a node of rank k. The theorem follows.

To do this, we truncate the growth of the potential function. For k fixed, redefine the potential of a node of rank k or greater to be b_2^k if it is $1, 1$ or $0, 1$; $a_2 b_2^k$ if it is $2, 2$ or $2, 3$; and zero otherwise. Outside of forced resets, an insertion or deletion increases the potential by $\mathrm{O}(1)$, since it makes $\mathrm{O}(1)$ changes all at nodes of rank $\mathrm{O}(1)$. This totals $\mathrm{O}(m)$ over all insertions and deletions. Consider a forced reset whose topmost step is of rank j. If $j \leq k - 2$, our assignment of a_2 and b_2 above ensures that the forced reset does not increase the potential, since the analysis above remains valid. If $j > k-2$, the truncated potential reduces the higher-order terms of rank k or greater, in decreasing order, in the four quantitites above; checking all four cases shows that the potential increase remains nonpositive. Furthermore, if the forced reset starts by promoting or demoting a node of rank k, then it decreases the potential by $\Omega(b_2^k)$. The worst case is a forced reset during an insertion whose topmost rebalancing step is a rotate or double rotate. In this case, the potential increases by at most $b_2^k - b_2^k - b_2^k + (a_2 - 1)b_2^k - b_2^k = a_2 b_2^k - 3b_2^k = -\Omega(b_2^k)$ by our choice of a_2. Since the total increase in potential is $\mathrm{O}(m + d) = \mathrm{O}(m)$ and the potential is always non-negative, there are only $\mathrm{O}(m/b_2^k)$ such forced resets. □

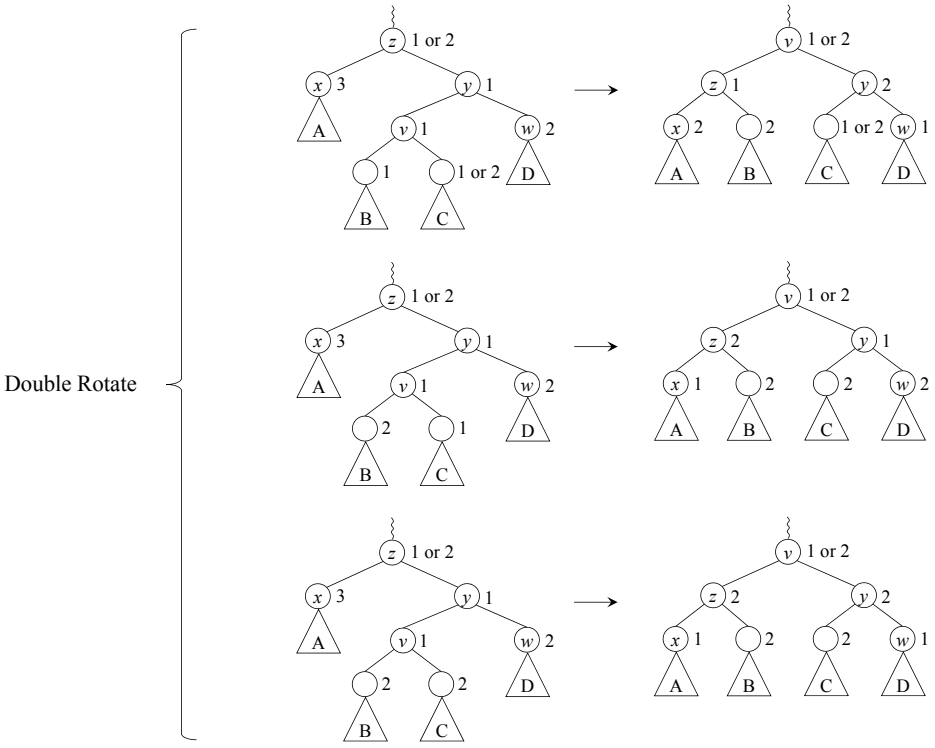

Fig. 4. The modified Double Rotate step for deletion rebalancing with promotion. Numbers next to nodes are their rank differences.

7. REBALANCING WITH PROMOTION

An inspection of our results suggests that the worst-case rebalancing step is a double rotate during deletion. We can improve the constants in several of our theorems by modifying this step so that it reduces the number of $1,1$ nodes if this is possible. Specifically, at the end of a double rotate step during deletion, if node z is $1,1$ we promote it; otherwise, if y is $1,1$, we promote it. (See Figure 4.) The effect of such a promotion is to convert a $2,2$ node of some rank k into a $1,2$ node, and to convert a $1,1$ node of rank $k-2$ into a $2,2$ node. We call this variant *rebalancing with promotion*. We can do rebalancing with promotion either bottom-up or top-down.

This variant has at least two drawbacks: it is slightly more complicated than the original, and it invalidates the proof of Theorem 4.3: the induction does not apply to node y if it becomes $2,2$. (Of course, the proof of Theorem 4.3 is not valid for the original method if rebalancing is top-down with fixed look-ahead.) On the other hand, Theorems 4.1 and 5.1 hold for rebalancing with promotion, by the same proofs. (In the proof of Theorem 5.1 we can reduce the potential of $2,2$ and $2,3$ nodes from $8/3$ to $7/3$, which reduces the implied constant factor in the bound, but the look-ahead remains the same.) We also obtain the following improvements of Theorems 4.2, 6.2, 6.3, and 6.4.

THEOREM 7.1. *If rebalancing is bottom-up with promotion, there are at most* $3m + d \leq 4m$ *promote steps.*

PROOF. The same as the proof of Theorem 4.2, except that a double rotate step during a deletion does not increase the potential, so d deletions increase the potential by at most d. □

THEOREM 7.2. *If rebalancing is bottom-up with promotion, there are* $O(d/\sqrt{2}^k)$ *deletion rebalancing steps of rank* k.

PROOF. Like that of Theorem 6.2, but with $\sqrt{2}$ as the base in place of b_1. For the untruncated potential, a double rotate of rank j with promotion during a deletion increases the potential by at most $\sqrt{2}^{j-1}$, the same as a rotate. A terminal demote of rank j increases the potential by at most $\sqrt{2}^{j+1} - \sqrt{2}^{j-1} = \sqrt{2}^{j-1}$. Summing as in the proof of Theorem 6.2 shows that a sequence of non-terminal demote steps of which the last is of rank $j-2$ decreases the potential by $\sqrt{2}^{j-1}$. Thus a deletion increases the potential by $\mathrm{O}(1)$ or reduces it. Truncating the potential and arguing as in the proof of Theorem 6.2 gives the theorem. □

THEOREM 7.3. *If rebalancing is bottom-up with promotion, there are* $\mathrm{O}(m/\sqrt{2}^k)$ *insertion rebalancing steps of rank* k*.*

PROOF. Like that of Theorem 6.3. A double rotate with promotion does not increase the potential, so a deletion of rank j increases the potential by at most

$$\begin{aligned} & \frac{b^{j-2}(1-ab^2)}{b^2-1} + b^{j-2} - ab^j + \mathrm{O}(1) \\ = \; & \frac{b^{j-2}(1-ab^2+b^2-1-ab^4+ab^2)}{b^2-1} + \mathrm{O}(1) \\ = \; & b^j(1-ab^2)/(b^2-1) + \mathrm{O}(1), \end{aligned}$$

which occurs when the last step is a double demote. If $a \geq 1/b^2$, this is $\mathrm{O}(1)$. Combining this with the upper bound on a that comes from the analysis of insertion gives $1/b^2 \leq a \leq (1+b-b^2)/(b(2-b))$, which implies $b^3 - b^2 - 2b + 2 \leq 0$. The left-hand side factors into $(b^2-2)(b-1)$, giving $b^2 \leq 2$ since $b > 1$. The choice $b = \sqrt{2}$ is the maximum that satisfies the constraint; this choice forces the choice $a = 1/2$. The rest of the proof is the same as that of Theorem 6.3. □

Let $a_3 = 2.589...$ and $b_3 = 1.150....$

THEOREM 7.4. *If rebalancing is top-down with promotion and with a forced reset during insertion at the fifth* $1,1$ *node in a row and during deletion at the third node in a row that is* $2,2$*, or* $1,2$ *with a 1-child that is* $2,2$*, then the number of rebalancing steps of rank* k *is* $\mathrm{O}(m/b_3^k)$*.*

PROOF. Almost the same as the proof of Theorem 6.4, with a_3 and b_3 in place of a_2 and b_2, respectively. A double rotate of rank j with promotion during a deletion increases the potential by at most $a_3 b_3^{j-1}$, the same as a rotate. Overall a forced reset during a deletion whose topmost step of rank j is a rotate or double rotate with promotion increases the potential by at most $a_3 b_3^{j-1} - a_3 b_3^{j-3} + b_3^{j-4} - a_3 b_3^{j-5} + b_3^{j-6} + (1-a_3) b_3^{j-7} + b_3^{j-8}$. Setting this quantity to be non-positive has the same effect as a forced reset whose topmost step is a double demote. This gives the improvement. □

8. RANK-BALANCED TREES VERSUS AVL TREES AND RED-BLACK TREES

In Section 3 we claimed that wavl trees combine the good properties of AVL trees and red-black trees. In this section we justify this claim by comparing the properties of these three types of trees.

Ignoring ranks, wavl trees are a proper subset of red-black trees, as the following results show.

THEOREM 8.1. *Given a ranked binary tree such that all rank differences are 1 or 2, its nodes can be assigned new ranks to make it a red-black tree.*

PROOF. Given a ranked binary tree with rank function r such that all rank differences are 1 or 2, assign to each node a new rank $r'(x) = \lfloor r(x)/2 \rfloor$. Since the original ranks are non-negative, so are the new ones. We claim that the red-black rank rule holds for the new ranks. If x is a leaf, $r(x) \leq 1$, since missing nodes have rank difference at most 2. Thus $r'(x) = 0$, and all missing nodes have new rank difference 1. Let x be a child. Since $r(p(x)) - r(x) \leq 2$, $r'(p(x)) - r'(x) \leq 1$. Let x be a

grandchild. Then $r(p(p(x))) \geq r(x) + 2$, which implies $r'(p(p(x))) - r'(x) \geq 1$. In particular, no parent of a 0-child is a 0-child. □

COROLLARY 8.2. *Every wavl tree can be assigned new node ranks to make it a red-black tree.*

THEOREM 8.3. *Given a red-black tree, its nodes can be assigned new ranks such that all rank differences are 1 or 2, and all red leaves have rank zero.*

PROOF. Given a red-black tree with rank function r, assign to each node x a new rank $r'(x) = 2r(x)$ if x is red, $2r(x) + 1$ if x is black. Then all ranks are non-negative. If x is a leaf, $r(x) = 0$, so $r'(x) \leq 1$, and all missing nodes have rank difference 1 or 2. Let x be a child. If x is red, $r(p(x)) = r(x)$ and x is black, so $r'(p(x)) = r(x) + 1$. If x is black, $r(p(x)) = r(x) + 1$, so $r'(p(x)) = r'(x) + 1$ if $p(x)$ is red, $r'(p(x)) = r'(x) + 2$ if x is black. □

We conclude from Theorems 8.1 and 8.3 that, ignoring ranks, red-black trees are exactly the ranked binary trees with rank differences 1 or 2, and a red-black tree with all red leaves can be converted into a wavl tree. A similar mapping converts a red-black tree with all black leaves into a wavl tree. A red-black tree with leaves of both colors may or may not be convertible into a wavl tree, however. We give a necessary and sufficient condition for conversion to be possible. We call a node x in a binary tree *lopsided* if for some k there is a path of length k from x to a leaf and another path of length $2k$ from x to a leaf. In the next lemma and theorem we adopt the convention that the root of a red-black tree is black.

LEMMA 8.4. *A node x in a red-black tree is lopsided if and only if there is a path of black nodes from x to a leaf, and a path of nodes alternating in color from x to a red leaf.*

PROOF. Let x be a lopsided node, with paths of lengths k and $2k$ to leaves. The rank of x is at most k by the length of the short path and at least k by the length of the long path, hence exactly k. It follows that the path of length k is all black and the path of length $2k$ alternates in color and ends at a red leaf. Conversely, let x be a node in a red-black tree with a path of k black nodes from x to a leaf, and a path alternating in color from x to a red leaf. Since all paths from x to leaves contain the same number of black nodes, the alternating-color path must have length $2k$. □

THEOREM 8.5. *A red-black tree can be assigned new node ranks to make it a wavl tree if and only if it does not contain a lopsided node.*

PROOF. Let x be a lopsided node in a red-black tree, with paths of lengths k and $2k$ to leaves. Because of the long path, new node ranks that make the tree a wavl tree must give x a rank of at least $2k$. But then one of the nodes other than x on the short path must have rank difference at least 3, since there are $k - 1$ such nodes and their rank differences sum to at least $2k$. Thus there is no such rank assignment.

Conversely, consider a red-black tree with no lopsided nodes. Every unary node is black with a red child that is a leaf; every red node is either a leaf or binary. Recolor the tree to move the red nodes toward the leaves by applying the following transformation until it no longer applies: given a binary red node x whose children are both leaves or whose grandchildren are all black, color x black and color its children red. This transformation preserves the red-black rule. Now every red node is a leaf or has a red grandchild, which implies that the parent of a red node has an alternating-color path to a red leaf. Let r be the rank function implied by the revised coloring. Give each node x a new rank $r'(x) = 2r(x)$ if x is red or there is a path of black nodes from x to a leaf, $2r(x) + 1$ otherwise. Then every leaf is either red or has an all-black path to a leaf (itself), so every leaf has new rank zero. If x is red, its parent, which is black, cannot have a black path to a leaf, or the parent would be lopsided. Since $r(x) = r(p(x))$, $r'(p(x)) = 2r(x) + 1 = r'(x) + 1$. If x is black with a red parent, $r(p(x)) = r(x) + 1$, so $2r(x) \leq r'(x) \leq 2r(x) + 1$ and $r'(p(x) = 2r(x) + 2$, giving x a rank difference of 1 or 2. If x is black with a black parent, either x has an all-black path to a leaf, in which case so does $p(x)$, and $r'(p(x)) = 2r(x) + 2 = r'(x) + 2$, or it does not, in which case

$2r(p(x)) + 2 \leq r'(p(x)) \leq 2r(p(x)) + 3$ and $r'(x) = 2r(x) + 1$, again giving x a rank difference of 1 or 2. □

AVL trees are a proper subset of wavl trees. Indeed, our bottom-up insertion algorithm for wavl trees is exactly the original insertion algorithm for AVL trees. AVL trees have a height bound of $\log_\phi n$, better than the $2 \lg n$ bound of red-black trees. The height bound of wavl trees, $\min\{\log_\phi m, 2 \lg n\}$ (Theorems 3.1 and 4.3) degrades gracefully from the AVL-tree bound to the red-black tree bound as the number of deletions increases. The height bound of red-black trees does not degrade gracefully. Indeed, a sequence of n insertions in increasing order into an empty red-black tree produces a tree of height $2 \lg n - \mathrm{O}(1)$, whereas the same sequence of insertions into a wavl tree produces a tree of height $\lg n + \mathrm{O}(1)$. Furthermore the total length of the insertion search paths is $2n \lg n - \mathrm{O}(n)$ in the red-black tree but $n \lg n + \mathrm{O}(n)$ in the wavl tree.

AVL trees require at most two rotations and $\mathrm{O}(\log n)$ rank changes per insertion but $\Omega(\log n)$ rotations per deletion, worst-case. Alternating insertions and deletions in an AVL tree can cause each deletion to do $\Omega(\log n)$ rotations, so the amortized number of rotations is $\Theta(\log n)$. Top-down insertion or deletion with fixed look-ahead in an AVL tree is problematic. (We do not know of an algorithm; we think there is none.)

Relaxing the AVL rank rule improves rebalancing efficiency. Bottom-up rebalancing after an insertion or deletion in a wavl tree takes at most two rotations and $\mathrm{O}(\log n)$ rank changes worst-case, $\mathrm{O}(1)$ rebalancing steps amortized. The same result holds for red-black trees, except that deletions can take up to three rotations. Top-down rebalancing can be done in wavl trees with fixed look-ahead in $\mathrm{O}(\log n)$ worst-case and $\mathrm{O}(1)$ amortized rebalancing steps per insertion or deletion. The same result holds for red-black trees [Tarjan 1985b]. In wavl trees, both bottom-up and top-down rebalancing with fixed look-ahead does $\mathrm{O}(m/b^k)$ rebalancing steps of rank k, where the base b depends on the rebalancing method. Such a result also holds for bottom-up rebalancing in red-black trees. Huddleston and Mehlhorn [Huddleston and Mehlhorn 1981; 1982] showed that bottom-up rebalancing in $2,4$-trees does $\mathrm{O}(m/(5/3)^h))$ rebalancing steps of height h, a bound improved to $\mathrm{O}((m+hd)/2^h)$ by Sen and Tarjan [Sen and Tarjan 2013]. These bounds give bounds for red-black trees by the standard mapping from $2,4$-trees to red-black trees (see Section 3); the latter bound implies that bottom-up rebalancing in red-black trees does $\mathrm{O}((m + dk)/2^k)$ rebalancing steps of rank k. Since the height of a red-black tree is at most twice its rank, the number of rebalancing steps of height h is $\mathrm{O}((m + dh)/\sqrt{2}^h)$. This is better than the bound in Theorem 6.2 for wavl trees, but not quite as good as the bounds in Theorems 7.2 and 7.3, for wavl trees in which rebalancing is bottom-up with promotion. No doubt the choice of a suitable potential function will yield a result like Theorem 6.4 for red-black trees with top-down rebalancing, but we have not (yet) worked out the details.

9. REMARKS

We have presented a framework that uses ranks and rank differences to define height-based balance in binary trees. Our framework gives natural definitions of classical balanced trees, including AVL trees and various forms of red-black trees. Using our framework, we have defined a new-type of height-balanced binary tree, the *weak AVL tree* or *wavl tree*, and shown that it has many of the good properties of both AVL trees and red-black trees. We have introduced exponential potential functions and used them to obtain inverse-exponential rank-based bounds on rebalancing in wavl trees. Such functions unify and simplify the height-based credit analysis of Huddleston and Mehlhorn. Our analysis handles rotations directly, avoiding a detour into multiway trees and back.

In our study of top-down rebalancing, we have analyzed a method that looks ahead five nodes (five ranks) on insertion and three nodes (six ranks) on deletion. Other choices are possible: there is a trade-off between insertion look-ahead length and deletion look-ahead length. In particular, for the original deletion method, one can obtain analogues of Theorems 5.1 and 6.4 if the look-ahead is seven nodes on insertion and two nodes on deletion. For deletion with promotion, one can obtain analogues of Theorems 5.1 and 7.4 if the look-ahead is four nodes on both insertion and deletion,

and also if the look-ahead is six nodes on insertion and two nodes on deletion. Instead of minimizing the look-ahead, if one increases it by a sufficiently large but fixed amount, one can get arbitrarily close to the plastic constant as a base for the original deletion method and arbitrarily close to $\sqrt{2}$ as a base if deletion is with promotion.

Some refinements and extensions of our results may be possible; we leave these for future work. Open questions include the following: (1) Can the "count" argument used in the proof of Theorem 4.3 be modified so that the potential is history-independent? (In all our other potential-based arguments, the potential is a function only of the current state of the tree, not of its history.) (2) Can the base in any of our rank-based analyses be improved? What are the bases for other choices of look-ahead in top-down rebalancing? (3) Can results like ours be derived for top-down rebalancing in red-black trees? (4) The main difficulty in our potential-based analyses is the number of inequalities that must be satisfied, corresponding to the number of insertion and deletion cases. Is there a systematic way to derive such results, perhaps using linear or non-linear programming, that would guarantee optimal constants?

REFERENCES

ADEL'SON-VEL'SKII, G. M. AND LANDIS, E. M. 1962. An algorithm for the organization of information. *Sov. Math. Dokl. 3*, 1259–1262.

AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. 1983. *Data Structures and Algorithms*. Addison-Wesley.

ANDERSSON, A. 1993. Balanced search trees made simple. In *WADS*. Vol. 709. 60–71.

BAYER, R. 1971. Binary B-trees for virtual memory. In *SIGFIDET*. 219–235.

BAYER, R. 1972. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Inf. 1*, 290–306.

BAYER, R. AND MCCREIGHT, E. M. 1972. Organization and maintenance of large ordered indexes. *Acta Informatica 1,* 3, 173–189.

BLUM, N. AND MEHLHORN, K. 1980. On the average number of rebalancing operations in weight-balanced trees. *Theoretical Computer Science 11,* 3, 303–320.

BROWN, M. R. 1978. A storage scheme for height-balanced trees. *Inf. Proc. Lett. 7,* 5, 231–232.

GUIBAS, L. J. AND SEDGEWICK, R. 1978. A dichromatic framework for balanced trees. In *FOCS*. 8–21.

HAEUPLER, B., SEN, S., AND TARJAN, R. E. 2009. Rank-balanced trees. In *WADS*. 351–362.

HUDDLESTON, S. AND MEHLHORN, K. 1981. Robust balancing in B-trees. In *GI-Conference on Theoretical Computer Science*. LNCS Series, vol. 104. 234–244.

HUDDLESTON, S. AND MEHLHORN, K. 1982. A new data structure for representing sorted lists. *Acta Informatica 17,* 2, 157–184.

KNUTH, D. E. 1973. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley.

LAAN, H. V. D. 1997. *Le Nombre Plastique: Quinze Lecons Sur L'Ordonnance Architectonique*. Brill Academic.

MCCREIGHT, E. M. 1985. Priority search trees. *SIAM J. on Comput. 14,* 2, 257–276.

MEHLHORN, K. 1984. *Data Structures and Algorithms 1: Sorting and Searching*. Vol. 1. Springer-Verlag.

MEHLHORN, K. AND TSAKALIDIS, A. 1986. An amortized analysis of insertions into AVL-trees. *SIAM Journal on Computing 15,* 1, 22–33.

METZGER, J. 1975. Managing simultaneous operations in large ordered indexes. Tech. rep., Technische Universität München, Institut für Informatik, TUM-Math.

NIEVERGELT, J. AND REINGOLD, E. M. 1973. Binary search trees of bounded balance. *SIAM J. on Comput. 2,* 1, 33–43.

OLIVIÉ, H. J. 1982. A new class of balanced search trees: Half balanced binary search trees. *ITA 16,* 1, 51–71.

SAMADI, B. 1976. B-trees in a system with multiple users. *Inf. Proc. Lett. 5,* 4, 107–112.

SEDGEWICK, R. 2008. Left-leaning red-black trees. http://www.cs.princeton.edu/ rs/talks/LLRB/LLRB.pdf.

SEN, S. AND TARJAN, R. E. 2013. Deletion without rebalancing in multiway search trees. In submission.

SKIENA, S. S. 1998. *The Algorithm Design Manual*. Springer-Verlag.

TARJAN, R. E. 1983. Updating a balanced search tree in $O(1)$ rotations. *Inf. Proc. Lett. 16,* 5, 253–257.

TARJAN, R. E. 1985a. Amortized computational complexity. *SIAM J. Algebraic and Disc. Methods 6*, 306–318.

TARJAN, R. E. 1985b. Efficient top-down updating of red-black trees. Tech. Rep. TR-006-85, Department of Computer Science, Princeton University.