DELETION WITHOUT REBALANCING IN BINARY SEARCH TREES*

SIDDHARTHA SEN†, ROBERT E. TARJAN‡, AND DAVID HONG KYUN KIM§

Abstract. We address the vexing issue of deletions in balanced trees. Rebalancing after a deletion is generally more complicated than rebalancing after an insertion. Textbooks neglect deletion rebalancing, and many B-tree-based database systems do not do it. We describe a relaxation of AVL trees in which rebalancing is done after insertions but not after deletions, yet access time remains logarithmic in the number of insertions. For many applications of balanced trees, our structure offers performance competitive with that of classical balanced trees. With the addition of periodic rebuilding, the performance of our structure is theoretically superior to that of many if not all classic balanced tree structures. Our structure needs $\lg\lg m + 1$ bits of balance information per node, where m is the number of insertions and lg is the base-two logarithm, or $\lg\lg n + \mathrm{O}(1)$ with periodic rebuilding, where n is the number of nodes. An insertion takes up to two rotations and $\mathrm{O}(1)$ amortized time, the same as in standard AVL trees. Using an analysis that relies on an exponential potential function, we show that rebalancing steps occur with a frequency that is exponentially small in the height of the affected node. Our techniques apply to other types of balanced trees, notably B-trees, as we show in a companion paper, and in particular red-black trees, which can be viewed as a special case of B-trees.

Key words. balanced trees, database access methods, exponential potential function, amortized complexity, data structure, algorithm

AMS subject classifications. 68P05, 68P10, 68Q25

1. Introduction.

Here is the true story that motivated this work, fictionalized to protect the parties involved. A database provider was contracted to build a real-time database to store customer information, to be queried and updated on a regular basis. The provider decided to use a red-black tree [12] to store the database, but implemented rebalancing only after insertions, not after deletions. As a safety check, a limit of 80 was placed on the allowed height of the tree. This limit would allow storage of 2^{40} records in a valid red-black tree, far exceeding the anticipated number. Exceeding the height bound was interpreted as an error and triggered a recovery process intended to restore the database. Some time after the database was placed in service, the height bound was exceeded, triggering the recovery process. Unfortunately, the recovery process was ill-designed, causing the height bound to be exceeded again, and this cycle repeated. The database provider had replicated the database for scalability and fault tolerance: queries were load-balanced across the replicas, while updates where applied to all of them. However, the failure that occurred was caused by a valid update, and since all replicas ran the same code, the same update caused every replica to fail, resulting in a total service outage and a lawsuit.

---

*A preliminary version of this article, not including the results in Section 7 and some of the results in Sections 9 and 10, appeared in *Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms* (Austin, Texas, Jan. 17-19), SIAM, Philadelphia, 2010 [31]. A preliminary version of the results in Section 7 appeared in the third author's undergraduate junior thesis [17].

†Microsoft Research, New York, NY 10011 (sidsen@microsoft.com). This author's research was partly done as a graduate student at Princeton University.

‡Department of Computer Science, Princeton University, Princeton, NJ 08540 (ret@cs.princeton.edu), and Intertrust Technologies, Sunnyvale, CA 94085. This author's research was partly done while visiting Stanford University, partially supported by an AFOSR MURI grant.

§Department of Computer Science, University of Chicago, Chicago, IL 60637, United States, hongk@cs.uchicago.edu. This author's research was done as an undergraduate student at Princeton University.

Research at Princeton University was 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.

Aside from the unfortunate practical outcome, this incident raised an interesting theoretical question: can one maintain balance in a search tree by rebalancing only after insertions, not after deletions? Even if the recovery process mentioned above had been implemented correctly, the rebalancing algorithm would still have allowed the tree height to grow enormously, severely degrading performance. Before considering whether this can be avoided, we review some of the literature concerning deletion in balanced trees. Such a review provides insight into how the event described above came about.

The original paper on balanced search trees [2], which introduced AVL trees to the world, is only four pages long. It describes how to rebalance an n-node AVL tree after an insertion by doing one or two rotations and updating height information in $\mathrm{O}(\log n)$ nodes. An algorithm for rebalancing after a deletion appeared several years later, in a technical report by a different author [9]. Deletion rebalancing requires $\mathrm{O}(\log n)$ rotations rather than $\mathrm{O}(1)$. In all existing forms of balanced trees, of which there are many (*e.g.*, [3, 4, 5, 12, 13, 14, 22, 25, 29]), deletion is at least a little more complicated than insertion, although for some kinds of balanced search trees, notably red-black trees [12] and the recently introduced weak AVL (wavl) trees [13, 14], rebalancing after a deletion can be done in $\mathrm{O}(1)$ rotations. Many textbooks describe algorithms for insertion but not deletion. If operations on the search tree occur concurrently, as in many database systems that use some form of B-tree as the underlying data structure, the synchronization necessary to do rebalancing on deletion reduces the available concurrency [11]. Several database systems, including Berkeley DB [26, 27], use a B^+ tree with nodes that are allowed to be underfilled, even empty. Nodes that become overfull as the result of an insertion are immediately split, but nodes that become underfull as the result of a deletion are allowed to remain underfull. Restructuring occurs only when a leaf becomes entirely empty, at which time the empty leaf and any connected empty internal nodes are deleted. Thus it was perhaps natural to try to avoid rebalancing on deletion in red-black trees. But disaster ensued.

A more precise version of our question is this: can one maintain a binary search tree so that search time is logarithmic but rebalancing is done only after insertions, not after deletions? To answer this question, we need to ask, "logarithmic in what parameter?" If there is no rebalancing after deletions (and none after accesses, which excludes self-adjusting structures such as splay trees [33]), then the tree can evolve to have arbitrary structure, which means that the search time can become $\Theta(n)$. But such an evolution may take many deletions, and it is still possible that the tree height, and hence the search time, could remain logarithmic in m, the number of insertions.

We show how to do this. We introduce a new kind of binary tree, the *ravl tree* (relaxed AVL tree), which is rebalanced only after insertions, not after deletions, and whose height is at most $\log_\phi m$, where ϕ is the golden ratio. This bound is the same as that for an ordinary AVL tree without deletions. Indeed, without deletions a ravl tree is exactly an AVL tree. Furthermore, rebalancing affects nodes exponentially infrequently in their heights, which means that the amortized rebalancing time per insertion is $\mathrm{O}(1)$ and most of the rebalancing occurs deep in the tree. Mehlhorn and Tsakalidis [20], proved the latter property for standard AVL trees if only insertions are allowed, not deletions. They also proved this property for "weak" B-trees with intermixed insertions and deletions, which include red-black trees as a special case. A result of Larsen and Fagerberg [19] on relaxed balanced B-trees and a related result of Boyar, Fagerberg, and Larsen [6] on chromatic trees, a type of relaxed balanced red-black tree, improve the constants of the Huddleston-Mehlhorn result for standard red-black trees. All these results use a multilevel credit argument. Our analyses use exponential potential functions, a tool that unifies and simplifies the multilevel credit method, and which we also used [13, 14] to analyze wavl trees. Our results hold for bottom-up rebalancing;

we extend them to top-down rebalancing with finite look-ahead as well. We obtain similar results for a relaxed version of red-black trees. Perhaps surprisingly, we obtain better constant factors for many of our bounds than the corresponding bounds for wavl trees. Thus not only does rebalancing after deletions complicate the implementation, it makes the performance of the data structure worse in some ways.

Two of us [30, 32] have also used exponential potential functions to prove that B^+ trees with underfilled nodes have similar properties, thus providing theoretical support for the use of such trees in actual database systems. Eliminating deletion rebalancing in binary trees is noticeably more challenging, as we discuss in Section 3.

The price we pay for our results on binary trees is that each node in the tree must store $\lg\lg m + 1$ bits of balance information (or $\lg\lg n + \mathrm{O}(1)$ with periodic rebuilding)¶, rather than the one bit per node needed in AVL [2], wavl [13, 14], and red-black trees [12]. Indeed, we provide evidence to suggest that $\mathrm{O}(1)$ bits suffice only if one does cascading swaps of items between nodes during deletions. (We leave rigorous resolution of this question as an open problem.) We conclude that the approach used by the unfortunate database provider to keep red-black trees balanced without rebalancing on deletion was theoretically doomed. That this would manifest itself in practice is a wonder to a theoretician.

The body of our paper consists of nine sections. Section 2 contains our tree terminology. Section 3 discusses candidate approaches for avoiding deletion rebalancing and motivates the approach we take. Section 4 defines ravl trees and describes bottom-up rebalancing after an insertion; the rebalancing algorithm is that of AVL trees, extended to ravl trees. Section 5 analyzes the amortized efficiency of bottom-up rebalancing. Section 6 describes and analyzes top-down rebalancing with fixed look-ahead, an alternative rebalancing method that improves concurrency. Section 7 applies the ideas in Sections 4-6 to red-black trees. Section 8 describes a way to rebuild the trees efficiently if they become very unbalanced. Section 9 examines other ways of handling insertions and deletions, and gives examples showing that natural methods that use one balance bit per node fail. Section 10 explores the pros and cons of rebalancing after deletions. Section 11 contains final remarks.

2. Tree Terminology. Our tree terminology is the same as in [13, 14]. We repeat it here (almost verbatim) for completeness. 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. Missing nodes are *external*; non-missing nodes are *internal*. Each node is the *parent* of its children. We denote the parent of a node x by $p(x)$. The *root* is the unique node with no parent. A *leaf* is a node with both children missing. The *ancestor*, respectively *descendant* relationship is the reflexive, transitive closure of the parent, respectively child relationship. If x is a node, its *left*, respectively *right* subtree is the binary tree containing all descendants of $left(x)$, respectively $right(x)$. The *left*, respectively *right spine* of a binary tree is the path from the root down through left, respectively right children to a missing node. 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.

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 a 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

¶ We denote by lg the base-two logarithm.

the desired item. If the search key is less, respectively greater than that of 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 in the tree.

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 node x containing the item to be deleted by doing a search on its key. If neither child of x 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 y with a missing left, respectively right child. We swap the items in x and y. Now the node containing the item to be deleted is either a leaf or has one missing child. In the former case, we replace it by a missing node; in the latter case, we replace it by its non-missing child. If each node has pointers to its children, an access, insertion, or deletion takes $O(h + 1)$ time in the worst case, where 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. Henceforth, unless we explicitly state otherwise, by a binary tree we mean an internal binary search tree. Our results extend to external binary search trees and to other binary tree data structures. We denote by n, m, and d, respectively the current number of nodes, 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 parameters are related: $n = m - d$.

3. Design Candidates. Our goal is to design binary search trees that avoid rebalancing on deletion yet maintain an $\mathrm{O}(\log m)$ height bound at all times. In this section, we discuss some natural candidate solutions and their drawbacks, motivating our approach in the subsequent sections.

A simple way to avoid deletion rebalancing is to do deletions lazily, via the "tombstone" method: in an existing form of balanced tree, replace the deletion algorithm by the following: to delete an item, simply remove it from its node but leave its key, so that searching remains possible. If a new item with the same key is later inserted, store it in the node containing its key. The tombstone method eliminates not only rebalancing during deletions but also swapping of items between nodes. In analyzing the method, one can ignore deletions and consider trees built only by insertions. This immediately yields the same height bound as the underlying tree, except that n is replaced by m. For example, applying the tombstone method to AVL trees with bottom-up rebalancing on insertion yields a height bound of $\log_\phi m$. A similar application to red-black trees yields a height bound of $2 \lg m$. In fact, all the results we present in this paper can be adapted to the tombstone method.

The drawback of the tombstone method is that if the number of deletions approaches the number of insertions, the space required by the tree can become superlinear in the number of items. One can keep the space usage linear by removing empty nodes when possible: during a deletion, delete the item but not its key if it is in a node with two non-null children, or otherwise delete the node containing the item and replace this node by its non-empty child if it has one [7]. This method ensures that every empty node has two non-empty children, so the number of nodes is at most $2n - 1$. This method fails badly when applied to red-black trees however: in Section 9, we construct sequences of $\mathrm{O}(n)$ intermixed insertions and deletions that produce trees of height $\Omega(n)$. That is, only a linear number of updates, not a superpolynomial number, suffice to make the tree very unbalanced. Similar counterexamples exist for other kinds of balanced trees, such as AVL and wavl trees.

An even more relaxed way to do rebalancing is to avoid rebalancing during both insertions and deletions but maintain a separate thread (or threads, in a multi-threaded implementation) that does rebalancing. This idea has been studied by many authors (*e.g.*, [15, 16, 24, 23, 19, 6]), who call their data structures "relaxed balanced trees" of various kinds. These papers derive bounds on the total number of rebalancing steps that must be done to restore the balance of the tree, as a function of the number of updates (insertions and deletions). In the best of these results, e.g. [19, 6] the number of rebalancing steps is linear in the number of insertions, and the number of steps at a given tree height is exponentially infrequent in the height. The problem with this approach is that a sequence of n insertions of items in increasing order will produce a linear tree, in which searches will be extremely expensive until the balancing process has a chance to do its work. Furthermore, it is still necessary to do deletion rebalancing; it is just delayed. Our goal, on the other hand, is to completely avoid deletion rebalancing, both eager and lazy, while still maintaining, *at all times*, a logarithmic bound on search time.

These approaches fail to maintain $\mathrm{O}(\log m)$ tree height because they rely on standard insertion rebalancing algorithms and, more importantly, on standard ways to store balance information. Specifically, they store only $\mathrm{O}(1)$ balance bits per node, which means they are unable to retain enough information about the structure of the tree after sufficiently many deletions have occurred. The original tombstone method evades this problem at the cost of space, by retaining every node ever inserted. Is there a solution that uses only n nodes? This question is important, because practitioners that wish to avoid deletion rebalancing typically do so by taking an existing balanced tree implementation and removing the deletion rebalancing code. This is precisely what the database provider mentioned in the introduction did, to disastrous effect. In the next two sections, we develop a method that stores sufficient balance information in each node to guarantee $\mathrm{O}(\log m)$ tree height while doing deletions exactly as described in Section 2, with no rebalancing.

To understand our method, it is useful to compare B-trees with binary trees. In a B-tree, every leaf has the same depth. One can avoid rebalancing on deletion by allowing nodes to become underfilled, but a deletion that empties a leaf can trigger a cascade of deletions of empty nodes along a path up toward the root. In a binary tree, no nodes are empty, but nodes can have different depths. We store with each node a non-negative integer rank that is an estimate of the height of the node. Ranks provide a "scaffold" that guarantees a logarithmic height bound, replacing the leaf depth invariant in B-trees.

4. Relaxed AVL Trees. We define balance in a binary tree by giving each node a *rank* and imposing a *rank rule* that constrains the ranks. For a general discussion of this approach, which captures all forms of height balance of which we are aware, see [14].

A *ranked binary tree* is a binary tree in which each node x has an integer rank $r(x)$. Missing nodes have rank -1. The *rank difference* of a node x with parent $p(x)$ is $r(p(x)) - r(x)$. An *i-child* is a node of rank difference i; an *i,j-node* is a node whose children have rank differences i and j. The latter definition does not distinguish between left and right children. An *AVL tree* is a ranked binary tree in which every node is a 1,1-node or a 1,2-node. The leaves of an AVL tree are 1,1-nodes of rank zero. A *relaxed AVL tree*, or *ravl*[‖] *tree*, is a ranked binary tree that obeys the following rank rule: every rank difference is positive.

LEMMA 4.1. *In a ravl tree, each node has height no greater than its rank.*

Proof. Every node has rank greater than the maximum of the ranks of its children. Since missing nodes have rank -1, leaves have non-negative rank. The lemma follows by induction on the node rank. □

[‖] One meaning of "ravel" is "to undo the intricacies of". Ravl trees undo the intricacies of deletions.

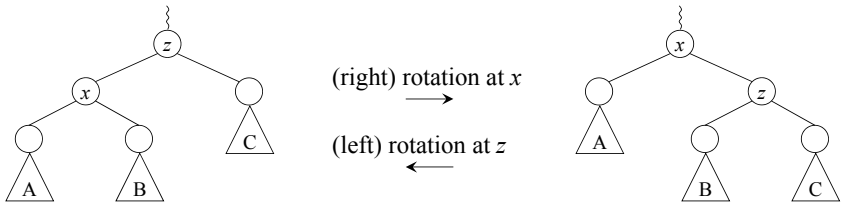


FIG. 4.1. *Rotation. Triangles denote subtrees.*

Any binary tree can be made into a ravl tree by a suitable choice of node ranks; indeed, there are always many ways to do it. The efficiency of ravl trees comes not from their static structure but from the implementation of insertions and deletions and how this affects the tree structure over time. We consider ravl trees built from the empty tree by a sequence of intermixed insertions of leaves and deletions of arbitrary nodes. A new leaf x replaces a missing node and has a rank of zero. If the parent $p(x)$ of x has rank 0 before the insertion, x is a 0-child and violates the rank rule. We restore the rank rule by promoting and demoting nodes and doing rotations. A *promotion* increases the rank of a node by one, a *demotion* decreases it by one. 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. (See Figure 4.1.) The insertion rebalancing algorithm is as follows (see Figure 4.2):

Insertion rebalancing:

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

Promote: Promote $p(x)$; replace x by $p(x)$.

Now either the rank rule holds or x is a 0-child whose sibling is an i-child with $i > 1$. In the latter case, proceed as follows. Assume x is the left child of $z = p(x)$; the other possibility is symmetric. Let y be the right child of x; y may be missing. Do the appropriate one of the following two steps:

Rotate: If node y is missing or a 2-child, rotate at x and demote z.

Double Rotate: Otherwise (node 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.

During rebalancing, there is exactly one violation of the rank rule: x is a 0-child. A rotate or double rotate step restores the rank rule and terminates rebalancing, as does a promote step that promotes the root or results in the new x being an i-child with $i > 0$. In the first rebalancing step, x is a leaf of rank zero and hence a 1,1-node; in each rebalancing step after the first, x is a 1,2-node. The *rank* of a rebalancing step is the rank of $p(x)$ just before the step. Each step has rank one higher than that of the previous step. The *rank* of an insertion is the rank of the last rebalancing step, or zero if there is no rebalancing.

To delete an item in a leaf in a ravl tree, we replace the leaf by a missing node. To delete an item in a node with one child, we remove the node and replace it by its child; this child becomes the left or right child of the old parent of the deleted node if the deleted node was a left or right child, respectively. To delete an item in a node with two children, we swap the item with its symmetric-order predecessor or successor, thereby moving it to a leaf or a node with one child, and proceed as above. In a deletion, no rotations occur and *no ranks change*.

As long as there are no deletions, all nodes remain 1,1- or 1,2-nodes (except in the middle of rebalancing), so the tree remains an AVL tree. Indeed, the rebalancing algorithm is just the

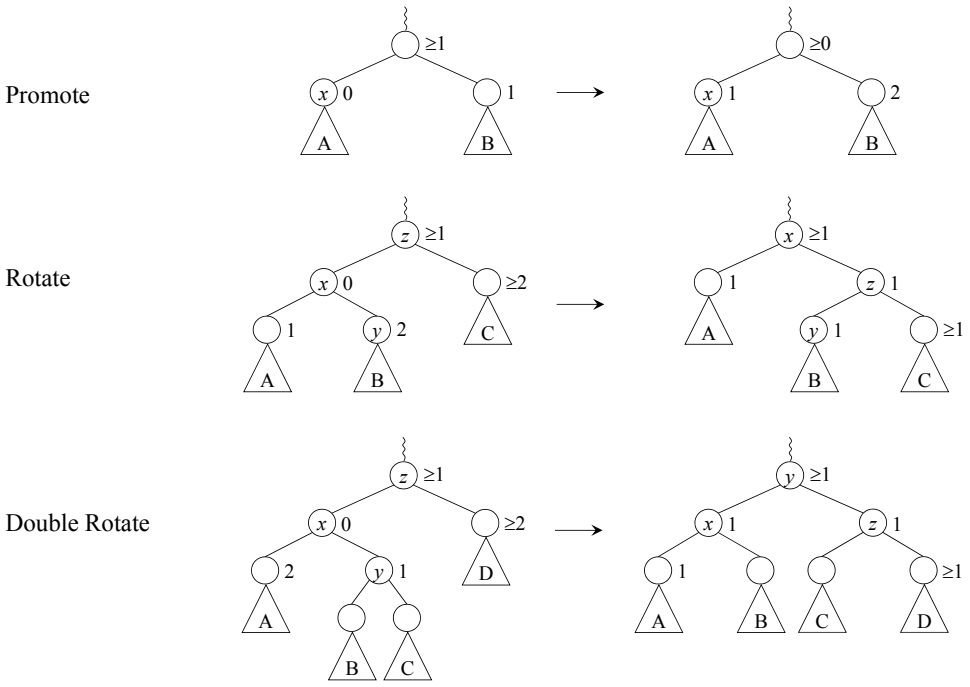


FIG. 4.2. *Bottom-up rebalancing after an insertion in a ravl tree. Numbers are rank differences. The first case is possibly non-terminating.*

standard bottom-up rebalancing algorithm for AVL trees. All the results we shall derive for bottom-up rebalancing hold as a special case for AVL trees built by insertions only. Deletions can create nodes of arbitrary positive rank difference, however, and thus can create trees that are not AVL trees. Indeed, deletions can produce trees of arbitrary structure.

We represent a ravl tree by storing with each node its rank and pointers to its left and right children. An alternative is to store ranks in difference form: the root stores its rank, and every child stores its rank difference. This only works if access to the tree is always via the root, and it requires computing node ranks during an insertion by summing rank differences along the path from the root to the new leaf. In an AVL tree, rank differences are one or two, so one bit per node suffices to store rank differences. But in a ravl tree, rank differences can become arbitrarily large, and storing ranks in difference form offers no advantages and at least one disadvantage. Thus we prefer to store ranks explicitly.

The rebalancing process after an insertion needs access to the affected nodes on the search path. There are several ways to provide such access, as we have discussed previously [14]. One way is to add parent pointers, which requires three pointers per node instead of two and increases the cost of rotations. By using an alternative representation that saves space but costs time, this can be reduced to two pointers per node [10].

Instead of adding or modifying pointers to support parental access, we can store the search path during the search from the root for the insertion position, 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 [21] and Samadi [28] used safe nodes to limit the amount of locking in a concurrent B-tree. 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. During an insertion, the safe node is either the root or the parent of the nearest ancestor of the current node that is not a 1-child and not a 1,1-node. A simpler alternative is to define the safe node to be the parent of the nearest ancestor of the current node that is not a 1,1-node, or the root if there is no such node. The latter definition gives the same node as the former, or its parent. We initialize the safe node to be the root and change it to the parent of the current node each time the current node is not a 1,1-node (or not a 1-child, if we are using the former definition). Once the search reaches the bottom of the tree, we do rebalancing steps (modified appropriately) top-down from the safe node to the new leaf. One advantage of this method is that it extends naturally to support top-down rebalancing with fixed look-ahead, as we discuss in Section 6.

5. Analysis of Bottom-Up Rebalancing. A search in a ravl tree takes $O(h+1)$ time, where h is the tree height. A deletion takes $O(h+1)$ time to find the item to be deleted and the node containing its replacement, if any, plus $O(1)$ time to do the deletion. An insertion takes $O(h+1)$ time to find the location of the new leaf, plus at most $h+1$ rebalancing steps, of which at most one is a single or double rotation. All these bounds are worst-case.

Our most important result is that a ravl tree built from an empty tree has height logarithmic in the number of insertions, even if deletions are intermixed arbitrarily. Recall the definition of the Fibonacci numbers F_k and of the golden ratio ϕ: $F_0 = 0, F_1 = 1, F_k = F_{k-1} + F_{k-2}$ for $k > 1$; $\phi = (1+\sqrt{5})/2$. The inequality $F_{k+2} \geq \phi^k$ is well-known [18].

THEOREM 5.1. *If a ravl tree of height h is built from an empty tree by a sequence of m insertions with bottom-up rebalancing intermixed with arbitrary deletions, $m \geq F_{h+3} - 1 \geq \phi^h$. Thus $h \leq \log_\phi m$.*

Proof. We use an exponential potential function of the kind first used by us to analyze wavl trees [13, 14]. Define the potential of a node of rank k to be F_{k+2} if it is a 0,1-node, F_{k+1} if it is a $0,j$-node with $j > 1$, F_k if it is a 1,1-node, and 0 otherwise. We define the potential of a tree to be the sum of its node potentials. We call this the *Fibonacci potential*.

A deletion cannot increase the potential. Inserting a new leaf increases the potential by at most 1: if the parent x of the new leaf was previously a leaf of rank 0, it becomes a 0,1-node of rank 0, if x was previously a 1,2-node of rank 1 (hence unary), it becomes a 1,1-node of rank 1; in either case the potential of x increases from 0 to 1. Consider a rebalancing step of rank k. A promote step that promotes $z = p(x)$ and makes $p(z)$ a 0,1-node does not change the potential: z and $p(z)$, respectively, have potentials F_{k+2} and F_{k+1} before the step and zero and $F_{k+3} = F_{k+2} + F_{k+1}$ after. A promote step that promotes $z = p(x)$ and makes $p(z)$ a $0,i$-node with $i > 1$ also does not change the potential: z and $p(z)$, respectively, have potentials F_{k+2} and zero before the step and zero and F_{k+2} after. Likewise, a rotate step does not increase the potential: nodes x and $z = p(x)$, respectively, have potentials zero and $F_{k+1} = F_k + F_{k-1}$ before the step and F_k and at most F_{k-1} after. Neither does a double rotate step: if $y = right(x)$ is a 1,1-node before the step, the total potential of x, y, and $z = p(x)$ is $F_{k-1} + F_{k+1}$ before the step and at most $F_{k-1} + F_k + F_{k-1}$ after; if y is not a 1,1-node before the step, the total potential of x, y, and z is F_{k+1} before the step and at most $F_k + F_{k-1}$ after. The final possibility is a terminal promote step. If the promotion of $z = p(x)$ makes $p(z)$ a 1,1-node, it does not change the potential: z and $p(z)$, respectively, have potentials F_{k+2} and zero before the step and zero and F_{k+2} after the step. If the promotion of z does not make $p(z)$ a 1,1-node, in particular if z is the root, the step decreases the potential by F_{k+2}.

The potential of an empty tree is 0. The case analysis in the previous paragraph shows that the potential can only increase, by at most 1, each time a new leaf is inserted into a non-empty tree. Thus the total increase in potential caused by a sequence of m insertions into an empty tree, intermixed with arbitrary deletions, is at most $m-1$. If the rank of the root is r,

there was a terminal promote step of rank i that promoted the root, for each i from 0 to $r-1$, inclusive. The total decrease in potential caused by these promotions is $\sum_{i=2}^{r+1} F_i = F_{r+3} - 2$. Since the potential is always non-negative, $m - 1 \geq F_{r+3} - 2$. By Lemma 4.1, $h \leq r$. Thus $m \geq F_{h+3} - 1 \geq F_{h+2} \geq \phi^h$. □

Remark: One can also prove Theorem 5.1 using an alternative definition of potential. Each node in the tree has a positive integer *count*. The potential of a node is the sum of the counts of all its descendants, including itself. We assign counts as follows: each new leaf gets a count of 1. When a leaf is deleted, its count is added to that of its parent, if it has a parent; when a unary node is deleted, its count is added to that of its child. It follows by a case analysis like that in the proof of Theorem 5.1 that the potential of a node x is at least $F_{k+3} - 1$, where $k = \max\{r(x), r(p(x)) - 2\}$, with $r(\text{null}) = 0$. This implies $m \geq F_{h+3} - 1$. This argument was suggested by an anonymous referee of a previous version of this paper; we also used an analogous argument to prove a similar result for wavl trees [13, 14]. We prefer the first proof above: we think it is a little simpler, it uses a potential that depends only on the structure of the tree, not its history, and the proof extends naturally to give inverse-exponential bounds on the amortized number of rebalancing steps, as we show below.

Although the worst-case insertion rebalancing time is $\mathrm{O}(\log m)$, we can obtain much better amortized bounds. Indeed, we can obtain the same amortized bounds as for AVL trees built by insertions, with no deletions. To obtain better bounds for rebalancing, we use the potential method of amortized analysis [35]. As in the proof of Theorem 5.1, we assign a non-negative *potential* to each state of the data structure, zero for an empty structure. We define the *amortized cost* of an operation to be its actual cost plus the net increase in potential it causes. Then, for any sequence of operations on an initially empty structure, the total amortized cost of the operations is an upper bound on their total actual cost.

THEOREM 5.2. *Starting with an empty ravl tree, a sequence of m insertions with bottom-up rebalancing intermixed with arbitrary deletions does at most $3m$ promote steps.*

Proof. Let the potential of a tree be the number of 0,1-nodes plus the number of 1,1-nodes of positive rank. Define the cost of an insertion to be the number of promote steps done during rebalancing. A deletion cannot increase the potential since it cannot create a 0,1-node or a 1,1-node. Consider an insertion. Adding a new leaf increases the potential by at most one, by making the parent of the new leaf into a 0,1-node or a 1,1-node of positive rank. A non-terminal promote step decreases the potential by one: the promoted node changes from a 0,1-node to a 1,2-node; the potential of its parent does not change. Thus such a step has an amortized cost of zero. This is also true of a promote step that promotes the root. A terminal promote step that promotes a node other than the root can leave the potential unchanged (if the parent of the promoted node becomes a 1,1-node), and thus has an amortized cost of one. A rotate or double rotate step can increase the potential by at most two, by creating two 1,1-nodes of positive rank. We conclude that the amortized cost of an insertion is at most three: one for the increase in potential caused by inserting a new leaf, plus zero for each non-terminal promote step, plus at most two for the last rebalancing step. □

By truncating the Fibonacci potential, we can prove the stronger result that rebalancing steps of rank k occur exponentially infrequently in k.

THEOREM 5.3. *Starting from an initially empty tree, a sequence of m insertions with bottom-up rebalancing intermixed with arbitrary deletions does at most $(m-1)/F_k \leq (m-1)/\phi^{k-2}$ rebalancing steps of rank k, for any $k > 0$.*

Proof. Fix $k > 0$. Let the potential of a node be its Fibonacci potential if its rank is less than k, F_{k+1} if its rank is k, it has a 0-child, and it is not a 0,1-node, and zero otherwise. Let the potential of a tree be the sum of the potentials of its nodes. The effect of a rebalancing step on the potential is the same as discussed above, with the following exceptions. A promote

step of rank k or higher does not change the potential. A promote step of rank $j < k$ that promotes a node $z = p(x)$ whose parent $p(z)$ has rank at least k decreases the potential by F_{j+2} unless $j = k - 1$, $p(z)$ has rank k, and $p(z)$ is not a 1,1-node before the step, in which case it does not change the potential. A rotate or double rotate of rank greater than k does not change the potential. A rotate or double rotate of rank k decreases the potential by at least $F_{k+1} - F_{k-1} = F_k$. Thus no rebalancing step increases the potential. Furthermore a rebalancing step of rank k either decreases the potential by at least F_k (if it is a rotate or double rotate) or is preceded by a promote step of rank $k - 1$ that reduces the potential by F_{k+1}. Thus the potential decreases by at least F_k for every rebalancing step of rank k. □

6. Top-Down Rebalancing. Rather than rebalance bottom-up after a new leaf is added, we can rebalance top-down before the leaf is added. Indeed, the safe node method described at the end of Section 4 does rebalancing top-down once it reaches the bottom of the tree. We can modify this method to rebalance more eagerly and thereby keep the look-ahead fixed; that is, keep the safe node within $\mathrm{O}(1)$ nodes of the current node of the search. This improves the worst-case concurrency of the tree, because the critical section of an insertion encompasses only $\mathrm{O}(1)$ nodes at any time. The idea is to force a reset of the safe node after a sufficiently large number of search steps that do not do a reset. A reset occurs at the next search step unless the current node is a 1,1-node. If the current node is a 1,1-node but not a 1-child, or both it and its parent are 1,1-nodes, we can force a reset by promoting the current node and rebalancing from the safe node top-down. This gives us the following top-down insertion algorithm, which we describe in complete detail to make its operation crystal-clear. If the tree is empty, create a new node of rank zero containing the item to be inserted and make it the root, completing the insertion. Otherwise, initialize t and z to be the root, and promote the root if it is 1,1. This establishes the invariant for the main loop of the algorithm: z is a non-null node that is not a 1,1-node; t is the parent of z unless z is the root, in which case $t = z$. Repeat the following step until the item is inserted (see Figure 6.1).

Top-down insertion step:

From z, take one step down the search path, to x. If x is null, replace it by a new node of rank zero containing the item to be inserted, completing the insertion: the new node cannot be a 0-child since z was not a 1,1-node and hence has positive rank. In the remaining cases, x is not null. If x is not a 1,1-node, replace t by z and z by x, completing the step. If x is a 1,1-node but not a 1-child, promote x and replace t by z and z by x, completing the step. In the remaining cases, x is a 1,1-node and a 1-child. From x take one step down the search path, to y. If y is null, replace it by a new node of rank zero containing the item to be inserted; if the new node is a 0-child, promote x and do a single or double rotate step to make all rank differences positive. This completes the insertion. The remaining possibility is y non-null. If y is not a 1,1-node, replace t by x and z by y. Otherwise, promote x and y, making x a 0-child, and do a single or double rotate step to make all rank differences positive. If a single rotation is done, replace t by x and z by y. If a double rotation is done, replace t by y and z by the new node one step down from y along the search path (either x or z). This completes the step.

If the first insertion step does a single or double rotation, it changes the root. Subsequent steps do not affect the root and have $t \neq z$. Each insertion step either finishes the insertion or replaces z by a node of smaller rank, so the number of insertion steps is at most one plus the rank of the root. We define the *rank* of an insertion step to be the rank of x, or zero if x is null.

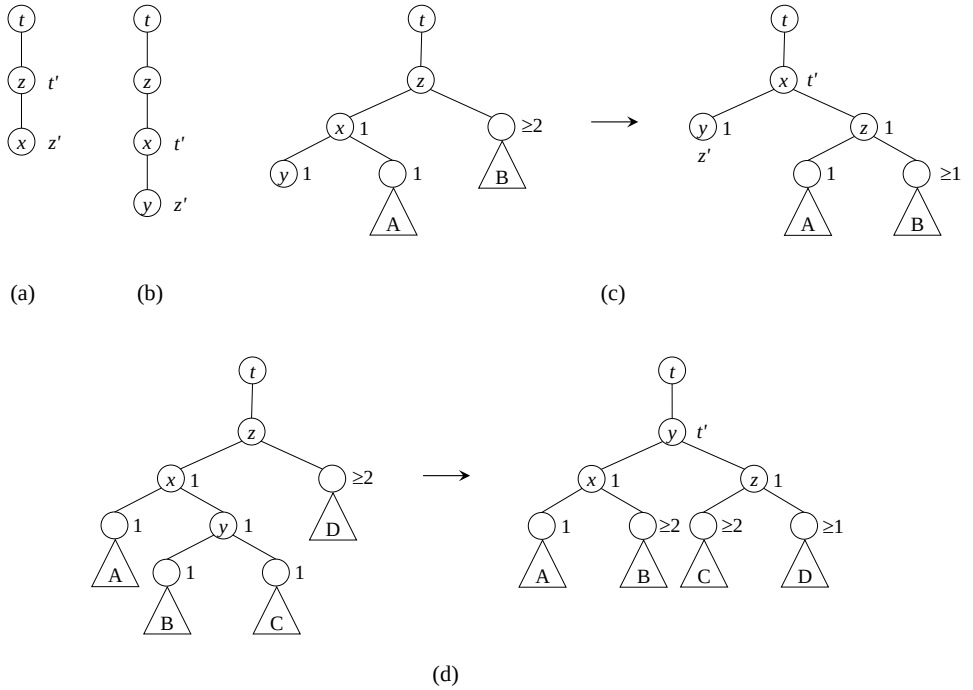


FIG. 6.1. *Top-down rebalancing step during an insertion in a ravl tree. Node z is not 1,1. Terminal cases are omitted. (a) Node x, the child of z along the search path is not 1,1 or not a 1-child. If x is 1,1, promote it; replace z by x and t by z. (b) Node x is 1,1 and a 1-child but y, its child along the search path, is not 1,1. Replace z by y and t by x. (c) Nodes x and y are 1,1, x is a 1-child, and x and y are both left (or both right) children. Promote x and y, demote z, rotate at x, replace z by y, and replace t by x. (d) Nodes x and y are 1,1, x is a 1-child, x is a left child, and y is a right child (or x is a right child and y a left child). Promote y twice, demote z, rotate at y twice, replace t by y, and replace z by the child of y along the search path.*

Every insertion step of positive rank is non-terminal, since in such a step x is non-null and is either a 1,1-node, in which case y must be non-null, or not a 1,1-node, in which case the step finishes without descending to y. We call an insertion step *rebalancing* if it is terminal or it does at least one promotion or rotation; the non-rebalancing steps merely traverse the search path without changing the tree.

We can show by a simple potential argument that the number of rebalancing steps is $O(m)$. Let the potential of a tree be twice the number of 1,1-nodes of positive rank plus the number of $1,i$-nodes with $i > 1$. An insertion into an empty tree does not increase the potential, nor does promoting z in the initialization. An examination of the remaining cases shows that a non-terminal rebalancing step decreases the potential by at least one, and a terminal rebalancing step increases it by at most one. This gives us the following theorem:

THEOREM 6.1. *Starting with an empty ravl tree, a sequence of m top-down insertions intermixed with arbitrary deletions takes at most $2m$ rebalancing steps.*

To obtain analogues of Theorems 5.1 and 5.3, we use an exponential potential function that grows more slowly than the Fibonacci potential. We define the potential of a node of rank $k > 0$ to be $2^{(k+1)/2}$ if it is a 1,1-node, $2^{(k-1)/2}$ if it is a $1,i$-node with $i > 1$, and zero otherwise; we define the potential of a node of rank zero to be zero; and we define the potential of a tree to be the sum of the potentials of its nodes. An insertion into an empty tree does not increase the potential, nor does promoting z in the initialization. Consider a restructuring insertion step of rank k that is not the last insertion step. If x is a 1,1-node that is not a 1-child,

it is promoted. This increases the potential by at most $2^{(k+1)/2} - 2^{(k+1)/2} \leq 0$, since the potential of x decreases from $2^{(k+1)/2}$ to zero, and the potential of z can only increase if it has rank $k+2$, in which case it increases by $2^{(k+1)/2}$. If x is a 1,1-node that is a 1-child, and y is also a 1,1-node, x and y are promoted and a single or double rotation is done. In the former case, the potentials of x, y, and z, respectively, are $2^{(k+1)/2}$, $2^{k/2}$, and $2^{k/2}$ before the step and $2^{(k+2)/2}$, 0, and at most $2^{(k+1)/2}$ after, resulting in no increase in the potential of the tree. In the latter case, the potentials of x, y, and z, respectively, are $2^{(k-1)/2}$, $2^{(k+2)/2}$, and at most $2^{(k-1)/2}$ after the step, again resulting in no increase in the potential of the tree. Consider a terminal insertion step. If the step replaces x by an empty node, it increases the potential of z, and hence of the tree, by at most one. Suppose the step replaces y by an empty node. If it does a single rotation, it increases the potential of x from zero to two and decreases that of z from 1 to zero, for a net increase of one. If it does a double rotation, it increases the potential of y from zero to two and decreases that of z from one to zero, again for a net increase of one. We conclude that a terminal insertion step increases the potential by at most one.

THEOREM 6.2. *A ravl tree built from an empty tree by a sequence of m top-down insertions intermixed with arbitrary deletions has height at most $2 \lg m$.*

Proof. Deletions do not increase the potential. By the discussion above, each insertion other than the first increases the potential by at most one. If the root has rank $k > 0$ and it is promoted, the potential decreases by $2^{(k+1)/2}$. For the root to have height h, it must have rank at least h, which means that root promotions have decreased the potential by at least $\sum_{i=1}^{h-1} 2^{(i+1)/2} = 2^{(h+1)/2}/(\sqrt{2}-1)$. Since the total decrease is at most m, this gives $2^{h/2} \leq m$. □

THEOREM 6.3.

Starting from an empty tree, a sequence of m top-down insertions intermixed with arbitrary deletions does at most $m/2^{k/2}$ rebalancing steps of rank k.

Proof. The lemma is immediate for $k = 0$. Fix $k > 0$. Redefine the exponential potential function used to prove Theorem 6.2 to be zero for all nodes of rank greater than k. It is still true that no insertion step increases the potential, and that each terminal step increases it by at most one. A rebalancing step of rank k is non-terminal. If it does not do a single or double rotation, it decreases the potential by $2^{(k+1)/2}$, by decreasing the potential of y from $2^{(k+1)/2}$ to zero. If it does a single rotation, it decreases the potential by at least $2^{k/2}$: the potentials of x, y, and z, respectively, are $2^{(k+1)/2}$, $2^{k/2}$, and zero before the step and zero, zero, and at most $2^{(k+1)/2}$ after. If it does a double rotation, it also decreases the potential by at least $2^{k/2}$: the potentials of x, y, and z, respectively, are $2^{(k-1)/2}$, zero, and at most $2^{(k-1)/2}$ after the step. The theorem follows. □

By increasing the amount of look-ahead in top-down insertion, we can improve the constants in Theorems 6.2 and 6.3. Specifically, we force a reset after traversing k consecutive 1,1-nodes, of which the top one is not a 1-child, or traversing $k+1$ consecutive 1,1-nodes of which the top one is a 1-child, by promoting the bottom 1,1-node and rebalancing appropriately. Here $k \geq 2$ is an appropriately large constant. To analyze this method, we define the potential of a 1,1-node of rank k to be b^k for some appropriate constant $b > 1$, that of any other node to be zero, and that of a tree to be the sum of the potentials of its nodes. If the parent of the top 1,1-node has rank k just before the forced reset, then the rebalancing increases the potential by at most $b^k - b^{k-2} - b^{k-3} - \ldots - 1$, whether or not the top 1,1-node is a 1-child. By choosing k sufficiently large, we can choose b arbitrarily close to ϕ while guaranteeing that forced resets do not increase the potential, giving an analogue of Theorem 6.2 with b in place of $\sqrt{2}$. By truncating the potential, we obtain an analogue of Theorem 6.3 with b in place of $\sqrt{2}$. Choosing $k = 3$ is sufficient to give $b > \sqrt{2}$. Interestingly, for minimum

look-ahead ($k = 1$), this potential function is not useful; for $k > 2$, giving positive potential to the $1, i$-nodes for $i > 1$ makes the analysis worse, not better.

7. Relaxed Red-Black Trees. In this section we apply our ideas to red-black trees to obtain relaxed red-black trees. Although the results of this section follow from our results on relaxed B-trees [32], we sketch them here for completeness, to show that they can be derived directly, without using multiway trees as an intermediary, and to enlarge our study in Section 9 of what can go wrong with alternative deletion methods.

A *red-black* tree [12] is a binary tree is which each node is either red or black, with the node colors satisfying the following constraints:

Black Rule: Every path from the root to a missing node contains the same number of black nodes.

Red Rule: The parent of a red node is black.

Red-black trees are equivalent to 2,4-trees, which are multiway trees in which all leaves have the same depth, each internal node has 2, 3, or 4 children, each internal node contains one less item than its number of children, and each leaf contains 1, 2, or 3 items. To obtain the 2,4-tree equivalent of a given red-black tree, contract each red node into its parent. To obtain the red-black tree equivalent of a given 2,4-tree, split each node with two items into a black parent and a red child, each with one item, and split each node with three items into a black parent and two red children. Since there are two ways to split a node with two items, the latter mapping is one-to-many.

Red-black trees are also equivalent to ranked binary trees satisfying the *red-black rank rule*: every node has a non-negative rank, all rank differences are zero or one, every leaf has rank zero, and every 0-child has a parent that is not a 0-child [14]. Given a ranked binary tree satisfying the red-black rank rule, we can color its nodes to satisfy the red and black rules by coloring the root black and coloring each child black if it is a 1-child or red if it is a 0-child. Given a red-black tree, we can assign ranks to the nodes to satisfy the red-black rank rule by giving each node a rank equal to the number of black nodes on every path from it to a missing node, not counting the node itself.

Red-black trees were invented by Bayer [5], who called them *symmetric binary B-trees*, and popularized by Guibas and Sedgewick [12], who invented the red-black representation. A red-black tree of n nodes has height at most $2 \lg n$. Rebalancing a red-black tree after an insertion or deletion takes at most two rotations worst-case for an insertion, at most three rotations worst-case for a deletion, and $\mathrm{O}(1)$ amortized color flips for an insertion or deletion [34]. The insertion rebalancing algorithm is like that of AVL trees (and ravl trees): there are three rebalancing cases (ignoring symmetries); the first does only color flips but need not terminate, the second does a rotation and some color flips and terminates, and the third does two rotations and some color flips and terminates. For further results and discussion about red-black trees see [5, 12, 34, 36].

We develop a relaxed version of red-black trees in which rebalancing occurs only during insertions, not deletions, with properties like those of ravl trees. To obtain such trees we relax the red-black rank rule to allow arbitrary positive ranks. A *relaxed red-black tree* is a ranked binary tree such that all ranks and rank differences are non-negative and no 0-child has a 0-child as a parent. To insert an item into a relaxed red-black tree using bottom-up rebalancing, follow the search path until reaching a missing node. Replace the missing node by a node x of rank zero containing the item to be inserted. Then rebalance as follows (see Figure 7.1):

Insertion rebalancing:

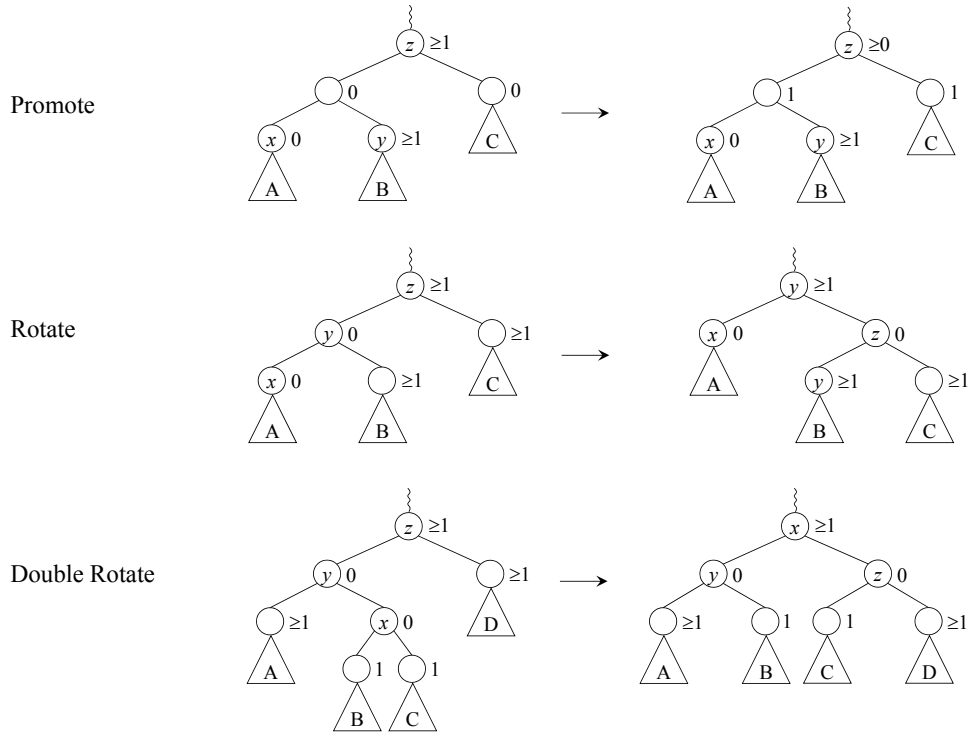


FIG. 7.1. *Bottom-up rebalancing after an insertion in relaxed red-black trees. Numbers are rank differences. The first case is possibly non-terminating.*

While x is a 0-child with a non-null grandparent z that is a 0,0-node, repeat the following step:

Promote: Promote z; replace x by z.

Now either the rank rule holds or x is a 0-child whose grandparent z is a $0, i$-node with $i > 0$. In the latter case, proceed as follows. Let y be the parent of x. Do the appropriate one of the following two steps:

Rotate: If nodes x and y are both left or both right children, rotate at y.

Double Rotate: Otherwise (node x is a left child and y a right child or vice-versa), rotate at x twice.

The *rank* of an insertion step is the rank of x just before the step.

To delete an item, we proceed exactly as in ravl trees: we find the node containing the item to be deleted and swap this item with its predecessor or successor if it is in a node with two non-null children. Now the item is in a leaf or a node with only one non-null child. If it is in a leaf, we replace the leaf by a missing node; if it is in a node with one non-null child, we replace this node by its non-null child. No nodes change ranks.

Instead of rebalancing bottom-up after an insertion, we can rebalance top-down during the search for the insertion position. If the tree is empty, create a new node of rank zero containing the item to be inserted and make it the root, completing the insertion. Otherwise, initialize t and z to be the root, and promote the root if it is 0,0. This establishes the invariant for the main loop of the algorithm: z is a non-null node that is not a 0,0-node and not a 0-

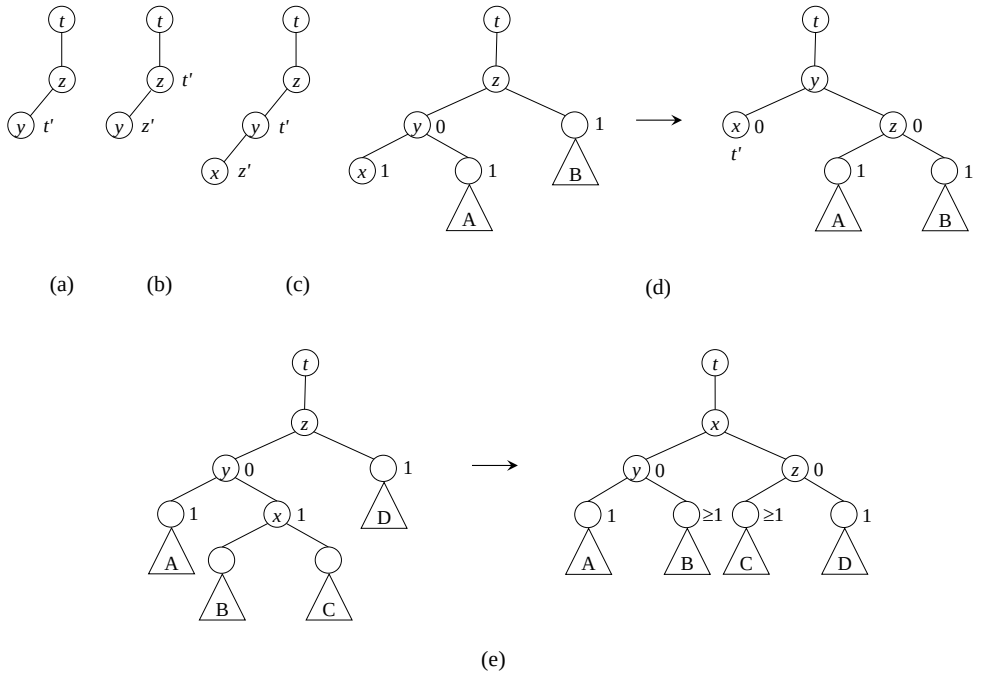


FIG. 7.2. *Top-down rebalancing step during an insertion in a relaxed red-black tree. Node z is not 0,0 and not a 0-child. Terminal cases are omitted. (a) Node y, the child of z along the search path, is 0,0. Promote y; replace t by y and z by the child of y along the search path. Since y is originally 0,0, the new z cannot be 0,0. (b) Node y is a 1-child but not 0,0. Replace t by z and z by y. (c) Node y is a 0-child and its child along the search path, x, is not 0,0. Replace t by y and z by x. (d) Node y is a 0-child, x is 0,0, and x and y are both left or both right children. Promote x, rotate at y, replace t by x, and replace z by the child of x along the search path. (e) Node y is a 0-child, x is 0,0, and x is a right child and y a left child (or x a left child and y a right child). Promote x, rotate at x twice, replace t by the child of x along the search path (y or z), and replace z by the child of t along the search path.*

child; t is the parent of z unless z is the root, in which case $t = z$. Repeat the following step until the item is inserted (see Figure 7.2):

Top-down insertion step:

From z, take one step down along the search path, to y. If y is null, replace it by a new node of rank zero containing the item to be inserted. This completes the insertion: the new node may be a 0-child, but z is not. In the remaining cases, y is non-null. If y is a 0,0-node, promote y, replace y by t, and replace z by the child of y along the search path; this child cannot be null since it has non-negative rank. This completes the step. If y is a 1-child that is not a 0,0-node, replace t by z and z by y, completing the step. In the remaining cases y is a 0-child and hence a 1,1-node. From y take one step down the search path, to x. If x is null, replace x by a new node of rank zero containing the item to be inserted; if the new node is a 0-child, do a single or double rotate step to restore the rank rule. This completes the insertion. The remaining possibility is x non-null. If x is not a 0,0-node, replace z by x and t by y, completing the step. Otherwise (x is a 0,0-node), promote x and do a single or double rotate step to restore the rank rule. If a single rotation is done, replace t by x and z by the child of x along the search path; if a double rotation is done, replace t by whichever of y and z is along the search path from x after the rotations, and replace z by the child of the new t along the search path. This completes the step.

Each top-down insertion step either finishes the insertion or replaces z by a node of smaller rank, so the number of insertion steps is at most one plus the rank of the root. We define the *rank* of a top-down insertion step to be the rank of y, or zero if y is null. Every such step of positive rank is non-terminal, since if y is a 1,1-node of positive rank, both of its children are non-null. We call a top-down insertion step *rebalancing* if it is terminal or it does at least one promotion or rotation.

We can analyze both bottom-up and top-down rebalancing using the same potential function. To get an amortized constant bound on the number of rebalancing steps, we define the potential of a tree to be twice the number of 0,0-nodes plus the number of $0, i$-nodes with $i > 0$. Deletions do not increase the potential. Insertion of a new node increases the potential by at most one. Each promote step in bottom-up rebalancing and each non-terminal rebalancing step in top-down rebalancing decreases the potential by at least one. Each single or double rotate step in bottom-up rebalancing, and each terminal insertion step in top-down rebalancing does not change the potential. This we obtain the following theorem:

THEOREM 7.1. *In a relaxed red-black tree built by m insertions, each with either bottom-up or top-down rebalancing, intermixed with arbitrary deletions, the number of rebalancing steps is at most $2m$.*

To bound the root rank and hence the height of the tree, we use an exponential potential function. We define the potential of a node of rank k to be 2^k if it is a $0, i$-node with $i > 0$, 2^{k+1} if it is a 0,0 node, and zero otherwise; we define the potential of a tree to be the sum of the potentials of its nodes. Insertion of a new node increases the potential by at most one. No insertion step, whether bottom-up or top-down, can increase the potential. If the root has rank k and it is promoted, the potential decreases by 2^{k+1}.

THEOREM 7.2. *In a relaxed red-black tree built by m insertions, each with either bottom-up or top-down rebalancing, intermixed with arbitrary deletions, the rank of the root is at most $\lg(m+1) - 1$, and the height of the root is at most $2\lg(m+1)$.*

Proof. The only increase in potential is caused by insertions of new nodes and totals at most $m - 1$. (An insertion into an empty tree does not increase its potential.) If the root has rank k, there must have been a promotion of the root for each rank between 0 and $k - 1$ inclusive, decreasing the potential by $2^{k+1} - 2$. Thus $2^{k+1} \leq m + 1$, which implies $k \leq \lg(m+1) - 1$. The rank rule implies that the rank of the grandparent of a node is greater than the rank of the node, which implies that the height of the root is at most $2\lg(m+1)$. □

By truncating the exponential potential function, we can show that the number of rebalancing steps of rank k is exponentially small in k.

THEOREM 7.3. *In a relaxed red-black tree built from an empty tree by m insertions, each with either bottom-up or top-down rebalancing, intermixed with arbitrary deletions, the number of rebalancing steps of rank k is at most $m/2^{k-1}$.*

Proof. The theorem is immediate for $k = 0$. Fix $k > 0$. Redefine the exponential potential to be zero for all nodes of rank k or greater. Inserting a new node still increases the potential by at most one, and no insertion rebalancing step can increase it, but a non-terminal bottom-up step of rank k, which must be a promotion, decreases it by 2^k, as does a top-down rebalancing step of rank k. Since each bottom-up step of rank k is preceded by a non-terminal bottom-up step of rank $k - 1$, the theorem follows. □

We conclude this section with a few comments about ravl trees versus relaxed red-black trees. Rebalancing does fewer promotions in the latter than in the former, at least locally. On the other hand, the height bound is smaller by a constant factor for ravl trees with bottom-up rebalancing than for relaxed red-black trees, and by increasing the look-ahead we can also

make it smaller for ravl trees with top-down rebalancing. In a ravl tree the height of a node is at most its rank, but in a relaxed red-black tree the height of a node is at most twice its rank plus one. Thus to compare Theorems 5.3 and 6.3 with 7.3, we need to compare ϕ (the base in Theorem 5.3) or $\sqrt{2}$ (the base in Theorem 6.3) with $\sqrt{2}$ (the square root of the base in Theorem 7.3). If rebalancing is bottom-up, or if rebalancing is top-down and the look-ahead is at least 3, the comparison favors ravl trees. Determining which variant of which of these data structures is best under what actual circumstances is a subject for experimental investigation.

8. Rebuilding the Tree. As the ratio of the number of deletions to the number of insertions approaches one, the height of a ravl tree or a relaxed red-black tree can become $\omega(\log n)$, although it remains $\mathrm{O}(\log m)$. For many applications this is not a concern, but for those in which it is, we can keep the height $\mathrm{O}(\log n)$ by periodically rebuilding the tree. How to do the rebuilding, and how often, are interesting questions that deserve study. Here we offer several versions of a simple rebuilding method and a rule for when to rebuild. We discuss the rebuilding of ravl trees; one can rebuild relaxed red-black trees in the same way.

To rebuild the tree, we initialize a new tree to empty. Then we traverse the old tree in symmetric order, deleting each visited node and inserting it into the new tree. Traversing the old tree takes $\mathrm{O}(n)$ time. Doing the insertions in the new tree takes $\mathrm{O}(n \log n)$ time, most of it spent searching for the position to insert the next item. We can reduce the rebuilding time to $\mathrm{O}(n)$ by maintaining the right spine of the new tree in a stack, bottommost node on top. Then each insertion takes $\mathrm{O}(1)$ amortized time, and rebuilding the entire tree takes $\mathrm{O}(n)$ time.

We do not need to rebuild the tree all at once; we can do it incrementally, Once we decide to rebuild, we do $1 + \epsilon$ deletions from the old tree and reinsertions into the new tree for every actual insertion or deletion of an item, where $\epsilon > 0$ is an appropriately chosen constant, such as $\epsilon = 1$. To facilitate the rebuilding, we maintain a *current item* x that is the next item to be moved from the old tree to the new one. To begin rebuilding, we initialize x to be the item of minimum key in the tree. After moving an item from the old tree to the new one, we update x to be the item of minimum key in the old tree. Rebuilding stops when the old tree is empty. We insert a new item into the new tree if its key is less than that of the current item, into the old tree if it is greater. Similarly, deletions and searches are performed in the new tree if the item's key is less than that of the current item, in the old tree otherwise. To facilitate the rebuilding process, we can maintain the right spine of the new tree (which is the path to the position of the next insertion) and the left spine of the old tree (which is the path to the current item). These paths must be updated during insertions and deletions, but such updating takes $\mathrm{O}(1)$ amortized time per insertion or deletion during rebuilding.

There remains the question of when to rebuild. We want to do this often enough to guarantee that the tree height is $\mathrm{O}(\log n)$ at all times but the rebuilding time is small. To decide when to rebuild, we maintain the current number of items n and the number of insertions m' since the last rebuilding started, including reinsertions of items that were in the tree when the rebuilding started. A newly inserted item whose key is greater than that of the current item only counts once, not twice. For such an item we increment m' when it is reinserted, not when it is inserted. When $m'/n > \alpha$, we begin rebuilding, where α is a parameter. If rebuilding is all at once, we choose $\alpha > 1$. If rebuilding is incremental, we choose $\alpha \geq 1 + 1/\epsilon$. As we shall prove, this choice guarantees that each rebuilding completes before the next one begins. Only a deletion can trigger rebuilding, since an insertion increases both m' and n by 1 and hence cannot cause m'/n to exceed α, since $\alpha > 1$. When this deletion occurs, we do the deletion and then either rebuild the tree completely or delete and reinsert $1 + \epsilon$ items. The rebuilding process is deemed to start just after the deletion that triggers it.

THEOREM 8.1. *If rebuilding is done all at once, there are at most* $d/(\alpha-1)$ *reinsertions over all rebuildings,* $O(1/(\alpha-1))$ *per deletion.*

Proof. For a given rebuilding, consider the interval of time from the beginning of the previous rebuilding, or the beginning of the operation sequence if there is no previous rebuilding, to the beginning of the given rebuilding. Let n be the number of items in the tree at the beginning of the interval. Let Δm and Δd be the number of insertions of new items and the number of deletions done during the interval. The number of items in the tree at the beginning of the given rebuilding is $n + \Delta m - \Delta d$, as is the number of reinsertions during this rebuilding. For this rebuilding to occur, it must be the case that $m' = n + \Delta m > \alpha(n + \Delta m - \Delta d)$. Subtracting $n + \Delta m - \Delta d$ from both sides of this inequality and dividing by $\alpha - 1$, we get $\Delta d/(\alpha-1) > n + \Delta m - \Delta d$. Summing over all the rebuildings gives the theorem. □

THEOREM 8.2. *Suppose rebuilding is incremental. If the tree contains* n *items at the beginning of a rebuilding, at most* $n/\epsilon - 1$ *insertions of new items and* $n/(1+\epsilon) - 1$ *deletions occur during the rebuilding, and the rebuilding does at most* $n(1 + 1/\epsilon) - 1$ *reinsertions. Furthermore* $m'/n' \leq \alpha$ *throughout the rebuilding, where* n' *is the current number of items.*

Proof. The deletion that triggers the rebuilding results in $1 + \epsilon$ reinsertions, after which $n - 1 - \epsilon$ additional reinsertions are needed. Each insertion reduces the number of needed reinsertions by at least ϵ, each deletion by at least $1 + \epsilon$. The bounds on the numbers of insertions, deletions, and reinsertions during the rebuilding follow. (The bounds are not as tight as possible; we have dropped some subtractive terms.) Let Δm and Δd be the number of insertions of new items and the number of deletions, respectively, from the beginning of rebuilding up to some time t during rebuilding. Then $m' < n + \Delta m$, and the number of items n' at time t satisfies $n' = n + \Delta m - \Delta d$, so $m'/n' < (n + \Delta m)/(n + \Delta m - \Delta d) \leq n/(n - \Delta d) < n/(n - n/(1 + \epsilon))$ by the bound on Δd. This ratio is $1 + 1/\epsilon \leq \alpha$. □

THEOREM 8.3. *If rebuilding is incremental, there are at most* $d(1 + 1/\epsilon)/(\alpha - 1)$ *reinsertions over all rebuildings.*

Proof. We apply the argument in the proof of Theorem 8.1. For a given rebuilding, consider the interval of time from the beginning of the preceding rebuilding, or from the beginning of the operation sequence if there is no previous rebuilding, to the beginning of the given rebuilding. Let n, Δm, and Δd be the number of items in the tree at the beginning of the interval, the number of insertions of new items during the interval, and the number of deletions during the inteval, respectively. By the argument in the proof of Theorem 8.1, $\Delta d/(\alpha-1) > n + \Delta m - \Delta d$. Thus the number of items in the tree at the beginning of the rebuilding is at most $\Delta d/(\alpha - 1)$. By Theorem 8.2, the number of reinsertions during the rebuilding is at most $\Delta d(1 + 1/\epsilon)/(\alpha - 1)$. Summing over all rebuildings gives the theorem. □

THEOREM 8.4. *If rebuilding is done all at once, the tree height is always at most* $\log_\phi n + \log_\phi \alpha$ *if insertion rebalancing is bottom-up, at most* $2 \lg n + 2 \lg \alpha$ *if insertion rebalancing is top-down.*

Proof. The bound on tree height follows immediately from Theorem 5.1 for bottom-up rebalancing and from Theorem 6.2 for top-down rebalancing: if n' is the current number of items in the tree, $m'/n' \leq \alpha$ at all times except just after a deletion that triggers a rebuilding. □

THEOREM 8.5. *If the rebuilding process is incremental, both the old tree and the new tree have height at most* $\log_\phi n + \log_\phi(\alpha(1 + 1/\epsilon))$ *if insertion rebalancing is bottom-up, at most* $2 \lg n + 2 \lg(\alpha(1 + 1/\epsilon))$ *if insertion rebalancing is top-down.*

Proof. We prove the bound for bottom-up rebalancing using Theorem 5.1. The bound for top-down rebalancing follows similarly from Theorem 6.2. Consider an interval of time from the beginning of a rebuilding, or the beginning of the operation sequence, to the end of

the next rebuilding, or the end of the operation sequence if there is no next rebuilding. The tree associated with this interval is the new tree built by the rebuilding that starts the interval, or the original tree if the interval starts at the beginning of the operation sequence. We prove that the height bound holds for this tree throughout the interval, which gives the theorem. Let t be a time during the interval, and let m' and n' be the number of insertions into the tree during the interval up to time t and the number of items in the tree at time t, respectively. If the interval does not end in a rebuilding, or if it does but t precedes the deletion that triggers this rebuilding, then $m'/n' \leq \alpha$, so the height bound holds (without the factor of $1 + 1/\epsilon$) by Theorem 5.1. Suppose the interval ends with a rebuilding and t is during this rebuilding. Let n be the number of items just before this rebuilding, and let Δm and Δd be the number of insertions of new items and the number of deletions from the beginning of the rebuilding to t, respectively. Then $(m' - \Delta m)/(n + 1) \leq \alpha$. Also,

$$\begin{aligned} m'/n' &= m'/(n + \Delta m - \Delta d) \\ &\leq (m' - \Delta m)/(n + 1 - n/(1 + \epsilon)) \text{ by Theorem 8.2} \\ &\leq (m' - \Delta m)/(n(\epsilon/(1 + \epsilon)) + 1) \\ &\leq (m' - \Delta m)/((n + 1)(\epsilon/(1 + \epsilon))) \\ &\leq \alpha(1 + 1/\epsilon) \end{aligned}$$

The height bound follows by Theorem 5.1. □

By choosing ϵ fixed and α sufficiently large, even growing as a function of n, we can make the rebuilding time arbitrarily small compared to the time for insertions and deletions, at the cost of only an additive term in the height bound. It is straightforward to verify that the inverse exponential upper bounds on insertion rebalancing steps (Theorems 5.2, 5.3, 6.1, and 6.3) hold even with rebuilding, if the constant factors are adjusted to account for the reinsertions.

An alternative approach to rebalancing is to use local rebalancing steps, as in relaxed balanced trees. For example, one can use the rebalancing steps of chromatic trees [6] to rebalance a relaxed red-black tree. The rebalancing steps corresponding to insertion rebalancing (blacking, rb1, and rb2) are not needed. A similar set of local rebalancing steps suffices for ravl trees. If one does a sufficiently large constant number of rebalancing steps per deletion, one can keep the tree height logarithmic in n at all times [38]. There are at least two disadvantages of this approach: it restores balance much less efficiently than reinsertion, and it is more complicated, since it reintroduces the complexity of deletion rebalancing that we set out to eliminate.

9. Using fewer balance bits. So far, the only method we have discussed that achieves an $\mathrm{O}(\log m)$ height bound using only $\mathrm{O}(1)$ bits of balance information per node is the tombstone method without node deletions (Section 3). This method produces a data structure with $\Theta(m)$ nodes, not $\Theta(n)$, however. This leaves the question of whether there is a method that avoids rebalancing on deletion while using only n nodes and $\mathrm{O}(1)$ balance bits per node. We show that several approaches to this question fail, including the approach used by the database provider in the episode mentioned in the introduction. We give one method that succeeds, but it requires storing the rank of the root as well as doing a cascade of node swaps during a deletion. Our conclusion, based not on a proof but on consideration of several alternative methods, is that any method for which bounds like ours hold must rebalance on insertion and must either store $\Omega(\log \log m)$ bits of balance information per node or spend more than $\mathrm{O}(1)$ restructuring time per deletion. We now attempt to justify this conclusion.

Suppose we store rank differences instead of ranks in the nodes, and we want to keep the rank differences bounded (to 1 or 2 in a ravl tree, 0 or 1 in a relaxed red-black tree). How do

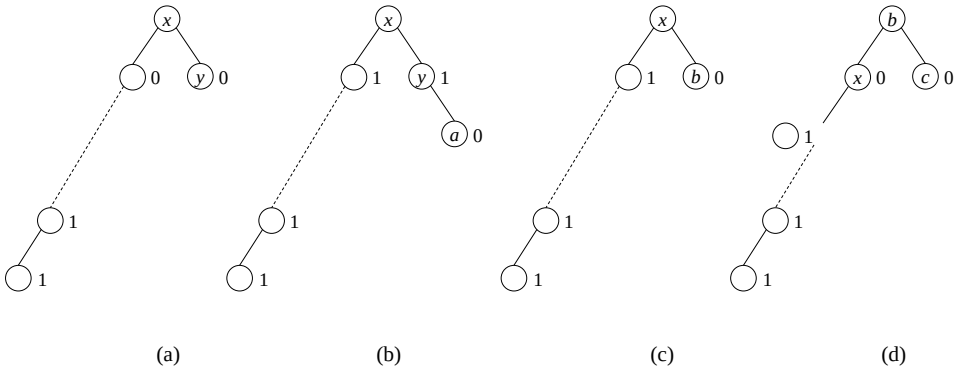


FIG. 9.1. *Counterexample for relaxed red-black trees with one balance bit per node. Rank differences are to the right of nodes. (a) Root has two red children. (b) After insertion of new node a. Children of root become black. (c) After deletion of two biggest items and insertion of new node b. (d) After insertion of new node c, causing a left rotation that increases the tree height by 1 and leaves the root with two red children.*

we do insertions? The most naïve idea is to avoid computing ranks while walking along the search path, merely inserting each new node with a fixed rank difference. If this difference is positive, we immediately run into the problem that a sequence of insertions of items in sorted order will produce a linear-depth tree, making search times linear. An alternative for relaxed red-black trees is to give new nodes a rank difference of zero. This is equivalent to coloring new nodes red, which is what the insertion algorithm for standard red-black trees does, and was the method used by the database provider discussed in Section 1. By mixing insertions with deletions, we can build a linear-depth tree, however. Insert three items in sorted order into an initially empty relaxed red-black tree. The third insertion creates a 0-child of a 0-child, causing a rotation and resulting in a tree whose root has two 0-children. Now repeat the following sequence of updates indefinitely: insert an item bigger than all previous ones, delete the two biggest items, and insert two items each bigger than all previous ones. Each sequence of an insertion, two deletions, and two insertions adds a 1-child to the left spine of the tree and produces a root with two red children; the tree consists of the left spine of the root and the right child of the root. (See Figure 9.1.) Thus an intermixed sequence of insertions and deletions can build a tree of linear depth. This example is a counterexample both to what the database provider did and to the space-efficient version of the tombstone method discussed in Section 3.

It is easy to construct a similar counterexample for the variant of ravl trees in which each new leaf has a rank difference of 0, and for the variant in which a new leaf that becomes a child of an old leaf has rank difference 0, but a new leaf that becomes a child of a unary node has rank difference 1. The latter variant corresponds to the way insertions are done in standard AVL trees.

It is not surprising that these variants fail, since insertions can increase our rank-based potential functions by an arbitrarily large amount, destroying our analyses. We can avoid this by maintaining the rank of the tree root. Then we can compute the rank of each node visited during a search by summing rank differences. In a ravl tree, if we give a new node a rank equal to the maximum of zero and two less than the rank of its parent (thereby giving it a rank difference of 0, 1, or 2), then adding such a node increases the potential by at most one, and all our analyses still hold. The equivalent method in a relaxed red-black tree is to give a new node a rank equal to the maximum of zero and one less than the rank of its parent, thereby giving it a rank difference of 0 or 1. Interestingly, the method of giving a new node

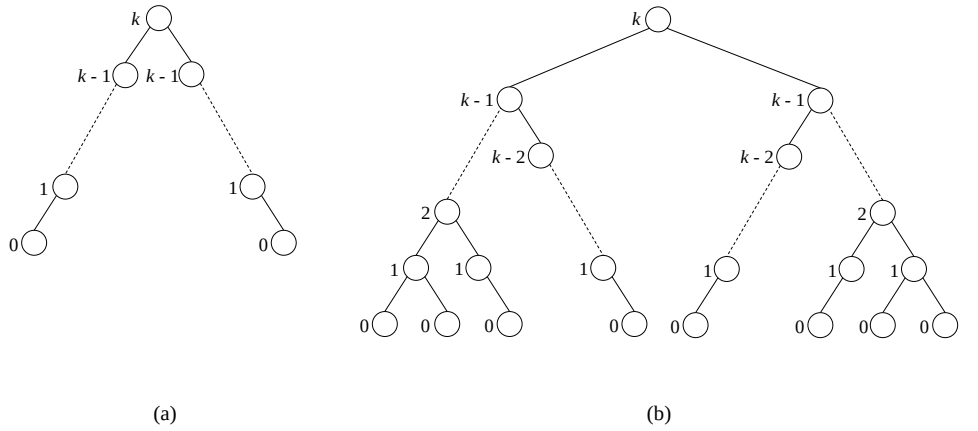


FIG. 9.2. *Counterexamples for two alternative methods of insertion and deletion in ravl trees that use one balance bit per non-root node. Node ranks are shown to the left. (a) New leaves get a maximum rank difference of 1. (b) New children get a maximum rank difference of 2.*

in a ravl tree a rank equal to the maximum of zero and one less than the rank of its parent (thereby giving it a rank difference of 0 or 1) fails, as the following counterexample shows. (See Figure 9.2(a).) For arbitrary $k \geq 1$, in $\mathrm{O}(k^2)$ insertions and deletions build a tree T_k of height k consisting of a root of rank k and a left and right spine, with each child having a rank difference of 1 and the two leaves having rank zero, as follows. For $k = 1$, do one insertion into an empty tree followed by one insertion of an item smaller than the one in the root and one insertion of an item larger than the one in the root. For $k > 1$, start with T_{k-1}. Do $2(k-1)$ insertions to give every non-leaf a second child of rank difference one. Then insert an item smaller than all those in the tree followed by an insertion of an item larger than all those in the tree. The first of these will increase the rank of the root; the second will make the root a 1,1-node. Finally, delete all the leaves that now have rank difference two. The result is T_k. Thus $\mathrm{O}(n)$ updates suffice to build a tree of height $\Theta(\sqrt{n})$.

So far our counterexamples have only involved deletions of leaves. Such deletions only reduce the tree potential. On the other hand, deletion of a unary node can increase the potential if the node replacing it (its old child) increases in rank. The following counterexample shows that this problem is not just hypothetical. (See Figure 9.2(b).) Consider the variant of ravl trees in which we maintain the rank of the root, insert each new leaf with a rank equal to the maximum of 0 and two less than the rank of its new parent, and when deleting a unary node give its replacing node a rank difference of 2. This method guarantees that all rank differences are 1 or 2. For arbitrary $k \geq 1$, in $\mathrm{O}(k^3)$ insertions and deletions build a tree T'_k of height k consisting of a root of rank k in which every child is a 1-child, every leaf has rank zero, every node on the left and right spines has two children, and the other non-leaves have one child. For $k = 1$, build $T'_1 = T_1$. For $k > 1$, start with T'_{k-1}. Insert an item less than all items in the tree and an item greater than all items in the tree. This increases the length of both spines by one, promotes all the items on the old spines including the root, and leaves the non-spine children of nodes on the spine with rank difference 2. To each of the leaves of rank zero and rank difference two, add a child via an insertion. This promotes the parent and results in a path of two nodes, each of rank difference one. For each of the remaining 2-children, proceed as follows. Delete the 2-child, replacing it by its only child, which increases in rank by one and becomes a 2-child. The leaf at the bottom of the path descending from this node now has rank one. Do two insertions to give this bottom node two 1-children.

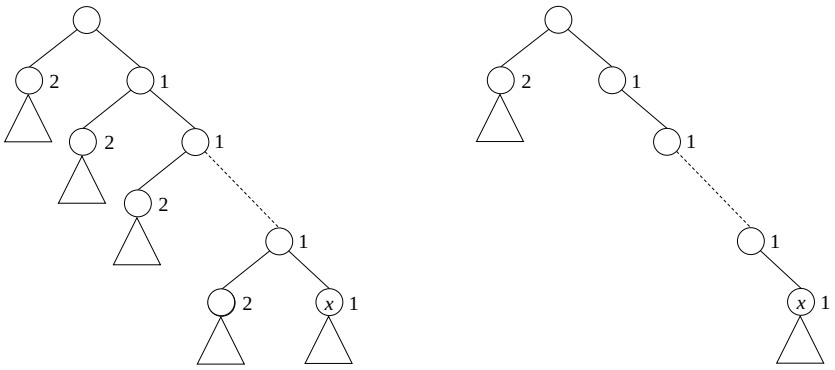
Delete the new 2-child, increasing the rank of the two new leaves to one. Choose one of these new leaves and do two insertions to give it two 1-children. Continue in this way until every node on the path down from the spine has two 1-children, and the topmost node on the path is a 2-child. Do one more insertion to add a child to one of the rank-0 leaves at the bottom of the path. This promotes every node along the path, creating a path of 1-children all the way to the spine. Now delete all the 2-children of nodes on this path. Repeating this construction for every 2-child of a node on the left or right spine produces T'_k. The number of updates to build T'_k from T'_{k-1} is $\mathrm{O}(k^2)$, so the total number of updates to build T'_k from an empty tree is $\mathrm{O}(k^3)$. Thus $\mathrm{O}(n)$ updates suffice to build a tree of height $\Theta(n^{1/3})$. Changing the insertions so that they add nodes of non-negative rank but rank difference 0, 1, or 2 does not affect this counterexample, since all insertions add nodes of rank 0. The counterexample also works if deletions use the tombstone method, since all deletions are of leaves or nodes with one child. A similar counterexample exists for relaxed red-black trees in which deletions are modified to keep all rank differences 0 or 1.

The counterexample in the previous paragraph suggests (but does not prove) that keeping the height logarithmically bounded using $\mathrm{O}(1)$ balance bits per node requires a deletion method that does not increase any ranks. We can obtain such a method by making sure that only leaves are deleted. This gives a valid method, but it requires arbitrarily long sequences of item swaps during deletions. There are three cases of deletion. To delete an item e, if e is in a leaf, merely delete the leaf. If e is in a node with a right child, swap e with the first item in the right subtree of the node containing e, and repeat this step until e is in a leaf; then delete the leaf. If e is in a node with a left child but no right child, proceed symmetrically: swap e with the last item in the left subtree of the node containing e, and repeat this step until e is in a leaf; then delete the leaf. The time for a deletion is $\mathrm{O}(h+1)$. If we use this deletion method and modify insertions so that a leaf added by a deletion has a rank equal to the maximum of zero and the rank of its parent minus two, we obtain a variant of ravl trees in which every node has rank difference 1 or 2 and the bounds we have derived hold. To represent ranks we need to store the rank of the root plus one bit per node. The same idea applies to relaxed red-black trees.

Since successive swaps during a deletion are at nodes of strictly decreasing ranks, the number of swaps during a deletion is at most $\log_\phi m$ by Theorem 5.1. The following example shows that this bound is tight to within a constant factor even in the amortized sense. Let k be an integer large enough that $k < \lfloor k/\lg\phi \rfloor - 1 = h$. Do a sequence of insertions that build an AVL tree of height h in which a child has rank difference 1 if it is a right child or it is a descendant of the node x of height k on the right spine, and all other (left) children have rank difference 2. (See Figure 9.3(a).) It is easy to prove by induction that such a tree can be built by an appropriate sequence of insertions using the insertion method of Section 3. Indeed, *any* tree satisfying the AVL rank rule can be so built.

Node x has $2^{k+1}-1$ descendants including itself. The tree contains at most $F_{h+3}-1 \leq \phi^{h+1} \leq 2^k$ non-descendants of x. Leaf by leaf, delete all nodes except the left child of the root, the nodes on the right spine, and the descendants of x. (See Figure 9.3(b).) Now repeatedly delete the item in the root. Each of the first $2^{k+1}-1$ such deletions will cause item swaps along the right spine from the root to x (and possibly some additional swaps). Thus each of these $\Omega(n)$ deletions does at least $\Omega(h-k) = \Omega(\log m)$ swaps.

10. To Rebalance on Deletion or Not?. Let us compare ravl trees and relaxed red-black trees to standard kinds of balanced trees. Deletion is much simpler in the former than in the latter. The price we pay for this simplicity is that the height bound is logarithmic in the number of insertions rather than the number of nodes, and each node needs to store $\lg\lg m + \mathrm{O}(1)$ bits of balance information rather than $\mathrm{O}(1)$. Rebalancing in ravl trees and relaxed red-



(a) (b)

FIG. 9.3. *Counterexample showing that $\Omega(\log m)$ swaps per deletion can occur. (a) Tree built by insertions. (b) Left children have rank difference 2 except those that are descendants of x. Right children and proper descendants of x have rank difference 1. Node x has $2^{k+1} - 1$ descendants. There are at most $F_{h+3} - 1$ non-descendants of x. (b) Tree after initial deletions. As long as the subtree rooted at x is non-empty, deletion of the root will cause swaps along the path from the root to x.*

black trees affects node exponentially infrequently in their height. The same is true in some kinds of standard balanced trees, including wavl trees and red-black trees [14, 32], but the bounds are not as good, because deletion rebalancing interferes with insertion rebalancing. Using periodic rebuilding as discussed in Section 8, we can reduce the height bound in ravl and relaxed red-black trees to logarithmic in the number of nodes. Using the multi-swapping deletion method discussed in Section 9, we can reduce the amount of balance information per node to $\mathrm{O}(1)$ in ravl trees and relaxed red-black trees, as long as we store the rank of the root as well. Storing the ranks explicitly, however, is only a small space penalty, and it reduces the context needed in rebalancing steps, so it seems a small price to pay. Indeed, at least some authors [3, 37] have advocated storing ranks in standard balanced trees, since it simplifies rebalancing. Doing this eliminates the space advantage of standard balanced trees. Our tentative conclusion is that our theoretical results favor ravl trees and relaxed red-black trees over their standard cousins.

We have also done some preliminary experiments in which we compared ravl trees and relaxed red-black trees (without periodic rebuilding) to standard red-black trees and wavl trees, on typical input sequences. Our results show that ravl trees and relaxed red-black trees perform significantly fewer rotations and balance information updates than the other trees, at the cost of slightly greater average and maximum path lengths. All balanced tree implementations were written in C; all reported quantities are machine-independent.

We generated five tree operation sequences, each performing a total of 2^{26} operations on a tree of size $n = 2^{13}$. To isolate the effect of rebalancing, only insertions and deletions were performed; the expected cost of interspersed accesses can be inferred from the average and maximum path lengths of the tree after each operation. Table 10.1 summarizes our results; the average and maximum path lengths reported are the average values over all operations. The first, fourth, and fifth operation sequences perform insertions and deletions on randomly selected items, chosen uniformly at random in the first sequence and according to a Zipf distribution [8, 39] with rank exponent $\alpha = 0.9346$ in the fourth and fifth sequences. (This value of α is based on a classic measurement study of the number of unique visitors seen by America Online on December 1, 1997 [1].) The fifth sequence simulates a dynamic Zipf dis-

Test	Red-black trees				Wavl trees			
	# rots $\times 10^6$	# bals $\times 10^6$	avg plen	max plen	# rots $\times 10^6$	# bals $\times 10^6$	avg plen	max plen
1. Random	26.44	116.07	10.47	15.63	29.55	133.74	10.39	15.09
2. Queue	50.32	285.13	11.38	22.50	50.33	184.53	11.20	14.00
3. Working set	41.71	185.35	10.51	16.18	43.69	159.69	10.45	15.35
4. Static Zipf	25.24	112.86	10.41	15.46	28.27	130.93	10.34	15.05
5. Dynamic Zipf	23.18	103.47	10.48	15.66	26.04	125.99	10.40	15.16

Test	Relaxed red-black trees				Ravl trees			
	# rots $\times 10^6$	# bals $\times 10^6$	avg plen	max plen	# rots $\times 10^6$	# bals $\times 10^6$	avg plen	max plen
1. Random	11.45	38.91	11.30	18.90	14.32	80.61	11.11	16.75
2. Queue	33.56	67.10	11.94	23.50	33.55	134.22	11.38	14.00
3. Working set	22.38	50.25	11.61	19.36	28.00	119.92	11.20	16.64
4. Static Zipf	10.78	38.47	12.73	24.69	13.48	78.03	11.12	17.68
5. Dynamic Zipf	10.24	37.83	11.36	18.86	12.66	74.28	11.11	16.84

TABLE 10.1

Performance comparison of red-black, wavl, relaxed red-black, and ravl trees on typical input sequences (rots = rotations, bals = balance information updates, avg plen = average path length, max plen = maximum path length).

tribution by randomly selecting an item and promoting it to the most popular rank after each operation (this simulates the "flash crowd" or "slashdot" effect often seen in websites). The second operation sequence simulates a queue by inserting the items in order and repeatedly deleting the smallest item in the tree and inserting an item larger than all other items in the tree. The third operation sequence randomly selects an item and inserts or deletes the $\lg n$ items centered around this item in symmetric order.

The results in Table 10.1 show that ravl trees performed significantly fewer rotations and balance information updates—over 42% and 35% fewer, respectively, on average—than red-black trees and wavl trees on the tested sequences. Similarly, relaxed red-black trees performed over 51% and 68% fewer rotations and balance information updates, respectively, on average. The price for this improvement is a slight increase in the average and maximum path length of the resulting trees: under 5.6% and 4.3% greater, respectively, for ravl trees, and under 11.3% and 33.4% greater, respectively, for relaxed red-black trees, on average. Wavl trees performed more rotations and balance information updates than red-black trees, but maintained better average and maximum path lengths. The same comparison holds for ravl trees and relaxed red-black trees.

We plan to conduct more thorough experiments on these and other balanced tree implementations, such as left-leaning red-black trees [5, 29]. In particular, we are investigating the performance of the trees on worst-case sequences, for which periodic rebuilding in ravl trees may be required to provide competitive performance.

11. Remarks. We have shown that one can obtain logarithmic worst-case search time in binary search trees that are rebalanced only after insertions, not after deletions. The resulting data structures are simpler than standard balanced search trees. Our results seem to require either that $\lg \lg m + \mathrm{O}(1)$ balance bits be stored per node, or that deletion be modified to delete only leaves, which seems to require cascaded swapping of items. Whether this can be proved or disproved is an open question. Also open is the best way to do incremental rebuilding to overcome the cumulative effect of deletion without rebalancing. On the experimental side, it would be valuable to do a systematic evaluation of the practical performance of ravl trees and relaxed red-black trees as compared to red-black trees, wavl trees, and other standard kinds of trees. Ravl trees and relaxed red-black trees combine simplicity with efficiency and may

well be very useful in practice.

REFERENCES

[1] LADA A. ADAMIC AND BERNARDO A. HUBERMAN, *Zipf's law and the Internet*, Glottometrics, 3 (2002), pp. 143–150.

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

[3] ARNE ANDERSSON, *Balanced search trees made simple*, in WADS, vol. 709, 1993, pp. 60–71.

[4] RUDOLF BAYER, *Binary B-trees for virtual memory*, in SIGFIDET, 1971, pp. 219–235.

[5] ———, *Symmetric binary B-trees: Data structure and maintenance algorithms*, Acta Inf., 1 (1972), pp. 290–306.

[6] JOAN BOYAR, ROLF FAGERBERG, AND KIM S. LARSEN, *Amortization results for chromatic search trees, with an application to priority queues*, J. Comput. System Sci., 55 (1997), pp. 504–521.

[7] NATHAN GRASSO BRONSON, JARED CASPER, HASSAN CHAFI, AND KUNLE OLUKOTUN, *A practical concurrent binary search tree*, in PPoPP, 2010, pp. 257–268.

[8] J. B. ESTOUP, *Gammes stenographiques.*, 1916.

[9] CAXTON C. FOSTER, *A study of AVL trees*, Tech. Report GER-12158, Goodyear Aerospace Corp., 1965.

[10] M. L. FREDMAN, R. SEDGEWICK, D. D. SLEATOR, AND R. E. TARJAN, *The pairing heap: a new form of self-adjusting heap*, Algorithmica, 1 (1986), pp. 111–129.

[11] J. GRAY AND A. REUTER, eds., *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, San Mateo, California, 1993.

[12] LEO J. GUIBAS AND ROBERT SEDGEWICK, *A dichromatic framework for balanced trees*, in FOCS, 1978, pp. 8–21.

[13] BERNHARD HAEUPLER, SIDDHARTHA SEN, AND ROBERT E. TARJAN, *Rank-balanced trees*, in WADS, 2009, pp. 351–362.

[14] ———, *Rank-balanced trees*, 2012. In submission.

[15] SABINE HANKE, THOMAS OTTMANN, AND ELJAS SOISALON-SOININEN, *Relaxed balanced red-black trees*, in CIAC, 1997, pp. 193–204.

[16] JOEP L. W. KESSELS, *On-the-fly optimization of data structures*, Commun. ACM, 26 (1983), pp. 895–901.

[17] DAVID HONG KYUN KIM, *Deletions without rebalancing in red black trees*, 2010. Undergraduate junior thesis, Department of Mathematics, Princeton University.

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

[19] K. S. LARSEN AND R. FAGERBERG, *Efficient rebalancing of B-trees with relaxed balance*, Internat. J. Found. Comput. Sci., 7 (1996), pp. 169–186.

[20] KURT MEHLHORN AND ATHANASIOS TSAKALIDIS, *An amortized analysis of insertions into AVL-trees*, SIAM J. on Comput., 15 (1986), pp. 22–33.

[21] J. METZGER, *Managing simultaneous operations in large ordered indexes*, tech. report, Technische Universität München, Institut für Informatik, TUM-Math, 1975.

[22] J. NIEVERGELT AND E. M. REINGOLD, *Binary search trees of bounded balance*, SIAM J. on Comput., 2 (1973), pp. 33–43.

[23] OTTO NURMI AND ELJAS SOISALON-SOININEN, *Chromatic binary search trees: A structure for concurrent rebalancing*, Acta Informatica, 33 (1996), pp. 547–557.

[24] O. NURMI, E. SOISALON-SOININEN, AND D. WOOD, *Concurrency control in database structures with relaxed balance*, in PODS, 1987, pp. 170–176.

[25] HENK J. OLIVIÉ, *A new class of balanced search trees: Half balanced binary search trees*, ITA, 16 (1982), pp. 51–71.

[26] MICHAEL A. OLSON, KEITH BOSTIC, AND MARGO I. SELTZER, *Berkeley DB*, in USENIX Annual, FREENIX Track, 1999, pp. 183–191.

[27] ———, *Berkeley DB*, 2000.

[28] BEHROKH SAMADI, *B-trees in a system with multiple users*, Inf. Proc. Lett., 5 (1976), pp. 107–112.

[29] ROBERT SEDGEWICK, *Left-leaning red-black trees*. www.cs.princeton.edu/~rs/talks/LLRB/LLRB.pdf, 2008.

[30] SIDDHARTHA SEN AND ROBERT ENDRE TARJAN, *Deletion without rebalancing in multiway search trees*, in ISAAC, 2009, pp. 832–841.

[31] SIDDHARTHA SEN AND ROBERT E. TARJAN, *Deletion without rebalancing in balanced binary trees*, in SODA, 2010, pp. 1490–1499.

[32] SIDDHARTHA SEN AND ROBERT E. TARJAN, *Deletion without rebalancing in multiway search trees*, ACM Transactions on Database Systems, (2013). To appear.

[33] DANIEL DOMINIC SLEATOR AND ROBERT ENDRE TARJAN, *Self-adjusting binary search trees*, J. ACM, 32 (1985), pp. 652–686.

[34] ROBERT ENDRE TARJAN, *Updating a balanced search tree in* $O(1)$ *rotations*, Inf. Proc. Lett., 16 (1983), pp. 253–257.
[35] R. E. TARJAN, *Amortized computational complexity*, SIAM J. Algebraic and Disc. Methods, 6 (1985), pp. 306–318.
[36] ROBERT ENDRE TARJAN, *Efficient top-down updating of red-black trees*, Tech. Report TR-006-85, Department of Computer Science, Princeton University, 1985.
[37] MARK A. WEISS, *Data Structures and Algorithm Analysis in Java*, Addison Wesley, 3rd ed., 2011.
[38] S. YANG AND R. E. TARJAN, *Eager and lazy deletion rebalancing in binary search trees*, 2014. Unpublished manuscript.
[39] G. K. ZIPF, *Selected studies of the Principle of Relative Frequency in Language.*, Harvard Univ. Press, 1932.