# Rank-Balanced Trees

Bernhard Haeupler[2], Siddhartha Sen[1,4], and Robert E. Tarjan[1,3,4]

[1] Princeton University, Princeton NJ 08544, {sssix, ret}@cs.princeton.edu
[2] CSAIL, Massachusetts Institute of Technology, haeupler@mit.edu
[3] HP Laboratories, Palo Alto CA 94304

**Abstract.** Since the invention of AVL trees in 1962, a wide variety of ways to balance binary search trees have been proposed. Notable are red-black trees, in which bottom-up rebalancing after an insertion or deletion takes $O(1)$ amortized time and $O(1)$ rotations worst-case. But the design space of balanced trees has not been fully explored. We introduce the *rank-balanced tree*, a relaxation of AVL trees. Rank-balanced trees can be rebalanced bottom-up after an insertion or deletion in $O(1)$ amortized time and at most two rotations worst-case, in contrast to red-black trees, which need up to three rotations per deletion. Rebalancing can also be done top-down with fixed lookahead in $O(1)$ amortized time. Using a novel analysis that relies on an exponential potential function, we show that both bottom-up and top-down rebalancing modify nodes exponentially infrequently in their heights.

## 1 Introduction

Balanced search trees are fundamental and ubiquitous in computer science. Since the invention of AVL trees [1] in 1962, many alternatives [2–5, 7, 10, 9, 11] have been proposed, with the goal of simpler implementation or better performance or both. Simpler implementations of balanced trees include Andersson's implementation [2] of Bayer's binary B-trees [3] and Sedgewick's related left-leaning red-black trees [4, 11]. These data structures are asymmetric, which simplifies rebalancing by eliminating roughly half the cases. Andersson further simplified the implementation by factoring rebalancing into two procedures, *skew* and *split*, and by adding a few other clever ideas. Standard red-black trees [7], on the other hand, have update algorithms with guaranteed efficiency: rebalancing after an insertion or deletion takes $O(1)$ rotations worst-case and $O(1)$ time amortized [13, 15]. As a result of these developments, one author [12, p. 177] has said, "AVL... trees are now passé."

Yet the design and analysis of balanced trees is a rich area, not yet fully explored. We continue the exploration. Our work yields both a new design and new analyses, and suggests that AVL trees are anything but passé. Our new design is the *rank-balanced* tree, a relaxation of AVL trees that has properties similar to those of red-black trees but better in several ways. If no deletions occur, a rank-balanced tree is exactly an AVL

[4] Research at Princeton University partially supported by NSF grants CCF-0830676 and CCF-0832797 and US-Israel Binational Science Foundation grant 2006204. The information contained herein does not necessarily reflect the opinion or policy of the federal government and no official endorsement should be inferred.

tree; with deletions, its height is at most that of an AVL tree with the same number of insertions but no deletions. Rank-balanced trees are a proper subset of red-black trees, with a different balance rule and different rebalancing algorithms. Insertion and deletion take at most two rotations in the worst case and $\mathrm{O}(1)$ amortized time; red-black trees need three rotations in the worst case for a deletion. Insertion and deletion can be done top-down with fixed look-ahead in $\mathrm{O}(1)$ amortized rebalancing time per update.

Our new analyses use an exponential potential function to measure the amortized efficiency of operations on a balanced tree as a function of the heights of its nodes. We use this method to show that rebalancing in rank-balanced trees affects nodes exponentially infrequently in their heights. This is true of both bottom-up and top-down rebalancing.

This paper contains five sections in addition to this introduction. Section 2 gives our tree terminology. Section 3 introduces rank-balanced trees and presents and analyzes bottom-up rebalancing methods for insertion and deletion. Section 4 presents and analyzes top-down rebalancing methods. Section 5 develops our method of using an exponential potential function for amortized analysis, and with it shows that rebalancing affects nodes with a frequency that is exponentially small in their heights. The concluding Section 6 compares rank-balanced trees with red-black trees.

## 2 Tree Terminology

A *binary tree* is an ordered tree in which each node $x$ has a *left child* $left(x)$ and a *right child* $right(x)$, either or both of which may be missing. Missing nodes are also called *external*; non-missing nodes are *internal*. Each node is the *parent* of its children. We denote the parent of a node $y$ by $p(y)$. 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 node $x$ is an ancestor of node $y$ and $y \neq x$, $x$ is a *proper ancestor* of $y$ and $y$ is a *proper descendant* of $x$. If $x$ is a node, its *left*, respectively *right* subtree is the binary tree containing all descendants of $left(x)$, respectively $right(x)$. The *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 is an item and the items are arranged in *symmetric order*: the key of 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 root. If they are equal, we have found 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 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 the new item. Deletion is a little harder. First we find the item to be deleted by doing a search on its key. If neither child of the item is missing, we find either the next item or the previous item, by walking down through left, respectively right children of the right, respectively left child of the item until reaching a node with a missing left, respectively right child. We swap the item with the item found. Now 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. An access, insertion, or deletion takes $O(h + 1)$ time in the worst case, if $h$ is the tree height.

An alternative kind of search tree is an *external binary search tree*: the external nodes are the items, the internal nodes contain keys but no items, and all the keys are in symmetric order. Henceforth by a binary tree we mean an internal binary search tree, with each node having pointers to its children. Our results extend to external binary search trees and to other binary tree data structures. We denote by $n$ the number of nodes and by $m$ and $d$, respectively, the number of insertions and the number of deletions in a sequence of intermixed searches, insertions, and deletions that starts with an empty tree. These numbers are related: $d = m - n$.

## 3 Rank-Balanced Trees

To make search, insertion, and deletion efficient, we limit the tree height by imposing a *rank rule* on the tree. A *ranked binary tree* is a binary tree each of whose nodes $x$ has an integer *rank* $r(x)$. We adopt the convention that missing nodes have rank minus one. The *rank* of a ranked binary tree is the rank of its root. If $x$ is a node with parent $p(x)$, the *rank difference* of $x$ is $r(p(x)) - r(x)$. We call a node an $i$*-child* if its rank difference is $i$, and an $i, j$*-node* if its children have rank differences $i$ and $j$; the latter definition does not distinguish between left and right children and allows children to be missing.

Our initial rank rule is that every node is a 1,1-node or a 1,2-node. This rule gives exactly the AVL trees: each leaf is a 1,1-node of rank zero, the rank of each node is its height, and the left and right subtrees of a node have heights that differ by at most one. To encode ranks we store with each non-root node a bit indicating whether it is a 1- or 2-child. This is Brown's representation [5] of an AVL tree; in the original representation [1], each node stores one of three states, indicating whether its left or right subtree is higher or they have equal heights. The rank rule guarantees a logarithmic height bound. Specifically, the minimum number of nodes $n_k$ in an AVL tree of rank $k$ satisfies the recurrence $n_0 = 1, n_1 = 2, n_k = 1 + n_{k-1} + n_{k-2}$ for $k > 1$. This recurrence gives $n_k = F_{k+3} - 1$, where $F_k$ is the $k^{\text{th}}$ Fibonacci number. Since $F_{k+2} \geq \phi^k$ [8], where $\phi = (1 + \sqrt{2})/2$ is the golden ratio, $k \leq \log_\phi n \leq 1.4404 \lg n$[5].

AVL trees support search in $\mathrm{O}(\log n)$ time, but an insertion or deletion may cause a violation of the rank rule. To restore the rule, we change the ranks of certain nodes and do rotations to rebalance the tree. A *promotion*, respectively *demotion* of a node $x$

[5] We denote by lg the base-two logarithm.

increases, respectively decreases its rank 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 1.) A rotation takes $\mathrm{O}(1)$ time.

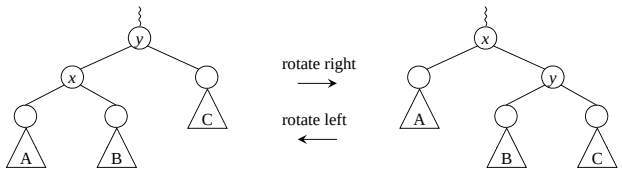


**Fig. 1.** Right rotation at node $x$. Triangles denote subtrees. The inverse operation is a left rotation at $y$.

In the case of an insertion, if the parent of the newly inserted node was previously a leaf, the new node will have rank difference zero and hence violate the rank rule. Let $q$ be the newly added node, and let $p$ be its parent if it exists, null if not. After adding $q$, we rebalance the tree by repeating the following step until a case other than promotion occurs (see Figure 2):

**Insertion Rebalancing Step at $p$**:

*Stop*: Node $p$ is null or $q$ is not a 0-child. Stop.

In the remaining cases $q$ is a 0-child. Let $s$ be the sibling of $q$, which may be missing.

*Promotion*: Node $s$ is a 1-child. Promote $p$. This repairs the violation at $q$ but may create a new violation at $p$. Node $p$ now has exactly one child of rank difference one, namely $q$. Replace $q$ by $p$. Let $p$ be the parent of $q$ if it exists, null if not.

In the remaining cases $s$ is a 2-child. Assume $q$ is the left child of $p$; the other possibility is symmetric. Let $t$ be the right child of $q$, which may be missing.

*Rotation*: Node $t$ is a 2-child. Rotate at $q$ and demote p. This repairs the violation without creating a new one. Stop.

*Double Rotation*: Node $t$ is a 1-child. Rotate at $t$ twice, making $q$ its left child and $p$ its right child. Promote $t$ and demote $p$ and q. This repairs the violation without creating a new one. Stop.

During rebalancing there is at most one violation of the rank rule: node $q$ may be a 0-child. Rebalancing walks up the path from the newly inserted node to the root, doing zero or more promotion steps followed by one non-promotion step. The first step is either a stop or a promotion. After one promotion step, node $q$ is always a 1,2-node. The *rank* of the insertion is the rank of $p$ in the last step, just before the step occurs; if $p$ is null, the rank of the insertion is the rank of $q$ in the last step (a stop).

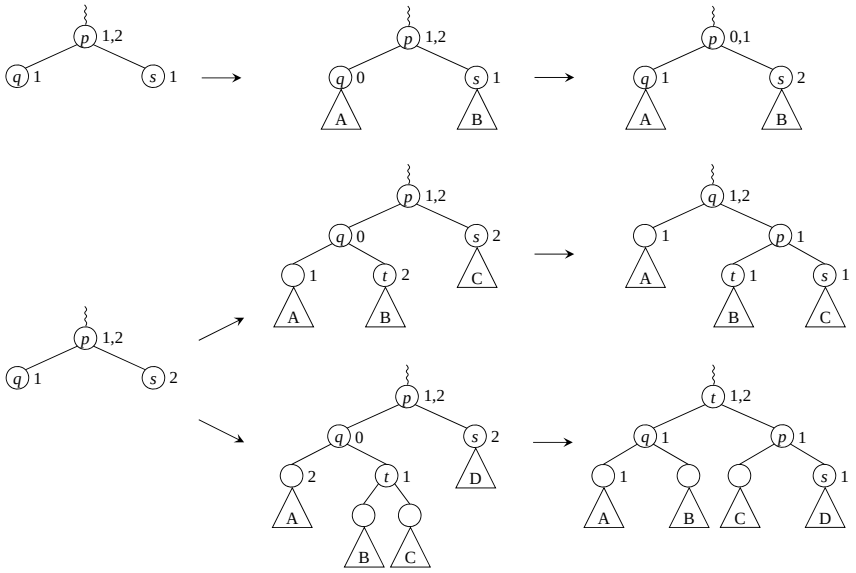


**Fig. 2.** Rebalancing after an insertion. Numbers are rank differences. The first case is non-terminating.

One can do a deletion in an AVL tree similarly [6] [8, pp. 465-468], but the rebalancing may require a logarithmic number of rotations, rather than the one or two needed for an insertion. To reduce this number, we relax the rank rule to allow non-leaf 2,2-nodes as well as 1,1- and 1,2-nodes; leaves must still be 1,1-nodes. We call the resulting trees *rank-balanced trees* or *rb*-trees (not to be confused with red-black trees, which are equivalent to ranked binary trees with a different rank rule). One bit per non-root node still suffices to encode the rank differences. An AVL tree is just an rb-tree with no 2,2-nodes. The rank of an rb-tree is at least its height and at most twice its height.

**Theorem 1.** *The rank and hence the height of an rb-tree is at most* $2 \lg n$.

*Proof.* The minimum number of nodes $n_k$ in an rb-tree of rank $k$ satisfies the recurrence $n_0 = 1, n_1 = 2, n_k = 1 + 2n_{k-2}$ for $k \geq 2$. By induction $n_k \geq 2^{\lceil k/2 \rceil}$. □

Insertion is the same in rb-trees as in AVL trees: insertion rebalancing steps do not create 2,2-nodes (but can destroy them). A deletion in an rb-tree can violate the rank rule by creating a node of rank one with two missing children or a node of rank two with a (missing) 3-child. Let $q$ be the node that replaces the deleted node ($q$ can be a missing node), and let $p$ be its parent if it exists, null if not. We repair the violation by walking up the path to the root, repeating the following step until a case other than demotion or double demotion occurs (see Figure 3):

**Deletion Rebalancing Step at $p$**:

*Stop*: Node $p$ is null, or $q$ is not a 3-child and $p$ is not a 2,2-node of rank 1. Stop.

In the remaining cases node $q$ is a 2- or 3-child. Let $s$ be the sibling of $q$, which may be missing.

*Demotion*: Node $s$ is a 2-child. Demote $p$. This repairs the violation at $q$ but may create a new violation at $p$. Replace $q$ by $p$. Let $p$ be the parent of $q$ if it exists, null if not.

In the remaining cases $q$ is a 3-child and $s$ is a non-missing 1-child. Assume $q$ is the right child of $p$; the other possibility is symmetric. Let $t$ and $u$ be the right and left children of $s$, either or both of which can be missing.

*Double Demotion*: Nodes $t$ and $u$ are 2-children: Demote $p$ and $s$. This repairs the violation at $q$ but may create a new violation at $p$. Replace $q$ by $p$. Let $p$ be the parent of $q$ if it exists, null if not.

*Rotation*: Node $u$ is a 1-child. Rotate at $s$, promote $s$, and demote $p$. If $t$ is missing, demote $p$ again. (In this case $q$ is also missing, and $p$ is now a leaf, whose rank must be zero.) This repairs the violation without creating a new one. Stop.

*Double Rotation*: Node $t$ is a 1-child and $u$ is a 2-child. Rotate at $t$ twice, making $s$ its left child and $p$ its right child. Promote $t$ twice, demote $s$, and demote $p$ twice. This repairs the violation without creating a new one. Stop.

During deletion rebalancing, there is at most one violation of the rank rule: $p$ is a 2,2-node of rank one or $q$ is a 3-child; after the first step, $q$ must be a 3-child. Rebalancing walks up the path from the node that replaces the deleted node toward the root, doing zero or more demotion and double demotion steps followed by a stop, a rotation, or a double rotation. The *rank* of the deletion is the rank of $p$ in the last step, just before the step occurs; if $p$ is null, the rank of the deletion is the rank of $p$ in the next-to-last step, just before the step occurs, or the rank of the deleted node if there is no next-to-last step (the root is deleted).

Deletion in rb-trees is only slightly more complicated than insertion, with two non-terminal cases instead of one. Deletion takes at most two rotations, the same as insertion.

The rebalancing process needs access to the affected nodes on the search path. To facilitate this, we can either add parent pointers to the tree or store the search path, either in a separate stack or by reversing pointers along the path. A third method is to maintain a *trailing node* during the search. This node is the topmost node that will be affected by rebalancing. In the case of an insertion, it is either the root or the parent of the nearest ancestor of the last node reached by the search that is a 2-child or a 1,2-node. In the case of a deletion, it is either the root or the parent of the nearest ancestor of the current node that is a 1-child or a 1,2-node whose 1-child is not a 2,2-node. In both cases, we initialize the trailing node to be the root and update it as the search proceeds. Once the search reaches the bottom of the tree, we do rebalancing steps (appropriately modified) top-down, starting from the trailing node. This method needs only $\mathrm{O}(1)$ extra space, but it incurs additional overhead during the search and during the rebalancing, to maintain

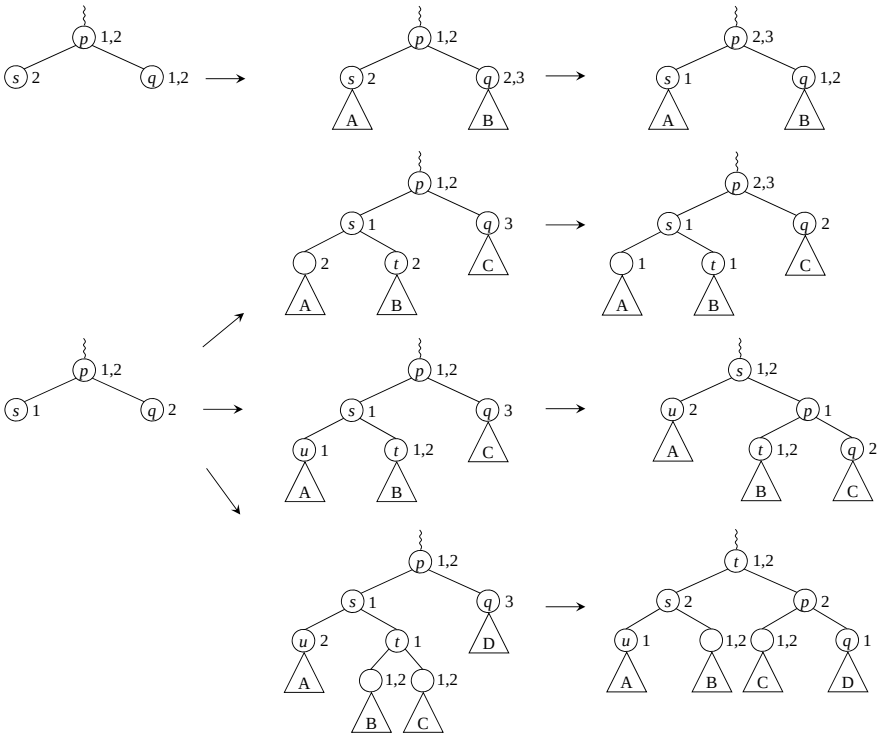


**Fig. 3.** Rebalancing after a deletion. Numbers are rank differences. The first two cases are non-terminating. If $q$ is a 2-child, the first case only applies if $p$ is a leaf. The third case assumes $t$ is not a missing node; if it is, $p$ is a leaf and is demoted.

the trailing node and to determine the next node on the search path, respectively. Its big advantage is that it extends to top-down rebalancing with finite look-ahead, as we discuss in the next section.

With any of these methods, a search, insertion, or deletion takes $\mathrm{O}(\log n)$ time worst-case. The number of rebalancing steps in an insertion or deletion is $\Theta(\log n)$ worst-case but $\mathrm{O}(1)$ amortized. To obtain this bound, we use a standard method of amortized analysis [14]. We assign to each state of the data structure a non-negative *potential* that is zero for an empty (initial) structure. We define the *amortized cost* of an operation to be its actual cost plus the net increase in potential it causes. Then the sum of the amortized costs is an upper bound on the sum of the actual costs for any sequence of operations that begins with an empty structure.

To analyze rb-tree rebalancing, we define the potential of a tree to be the number of non-leaf 1,1-nodes plus twice the number of 2,2-nodes. Each non-terminal insertion rebalancing step decreases the potential by one by converting a 1,1-node into a 1,2-node. Each non-terminal deletion rebalancing step except possibly the first decreases the potential by at least one, by converting a 2,2-node into a 1,2- or 1,1-node. The first deletion rebalancing step can increase the potential by one, by converting a 1,2-node

into a 1,1-node. A terminal insertion or deletion rebalancing step increases the potential by at most two or three, respectively. This gives the following theorem:

**Theorem 2.** *The total number of rebalancing steps is at most* $3m + 6d$.

We conclude this section by deriving a bound on the height of rb-trees that is close to that of AVL trees unless there are almost as many deletions as insertions.

**Theorem 3.** *With bottom-up rebalancing, the height of an rb-tree is at most* $\lg_\phi m$.

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

We noted earlier that $F_{k+3} - 1$ satisfies the recurrence $x_0 = 1, x_1 = 2, x_k = 1 + x_{k-1} + x_{k-2}$ for $k > 1$. This gives $K(x) \geq F_{k+3} - 1$ if $k = 0$; $k = 1$; or $k > 1$, $x$ is a 1,1- or 1,2-node, and the inequality holds for both children of $x$. This implies that the inequality holds for a new leaf and after each rebalancing step of an insertion. In the case of a promotion step, the children of the promoted node satisfy the inequality before the promotion; since the promoted node becomes a 1,2-node, it satisfies the inequality after the promotion. In the cases of rotation and double rotation, the children of the affected nodes satisfy the inequality before the step; since none of the affected nodes becomes a 2,2-node, they all satisfy the inequality after the step.

The inequality holds for the parent of a deleted node before rebalancing, since this node inherits the count of the deleted node. It also holds after each rebalancing step except possibly at a newly created 2,2-node. A 1,2-node that becomes a 2,2-node as a result of the demotion of a child satisfies the inequality because it did before the demotion. Node $s$ in a rotation step and node $t$ in a double rotation step satisfy the inequality after the step because $p$ satisfies the inequality before the step, and $s$, respectively $t$ has the same rank and count after the step as $p$ did before it. The only other case of a new 2,2-node is node $p$ in a rotation step if $p$ is not a leaf. For $p$ to become a 2,2-node, $q$ cannot be missing. Either $q$ was demoted by the previous rebalancing step, or $q$ is a leaf whose parent was deleted. In the former case, $q$ satisfies the inequality at rank $k - 1$ before its demotion, where $k$ is the new rank of $p$. Since $t$, the other child of $p$, satisfies the inequality at rank $k - 2$, $p$ satisfies the inequality as well. In the latter case $p$ has new rank two and has total count at least four, so it satisfies the inequality. □

## 4 Top-Down Rebalancing

The method of rebalancing using a trailing node described in Section 3 does the rebalancing top-down rather than bottom-up. We can modify this method to use fixed look-ahead. If the look-ahead is sufficiently large, the amortized number of rebalancing steps per update remains $\mathrm{O}(1)$. The idea is to force a reset of the trailing node after

sufficiently many search steps. In an insertion, if the current node of the search is a 1,1-node whose parent is a 1,1-node, we can force the next search step to do a reset by promoting the current node and rebalancing top-down from the trailing node. In a deletion, if the current node is a 2,2 node or a 1,2-node whose 1-child is a 2,2-node, we can force the next step to do a reset by demoting the current node in the former case or the current node and its 1-child in the latter case, and rebalancing top-down from the trailing node.

Forcing a reset as often as possible minimizes the look-ahead, but if we force a reset less often we can guarantee $\mathrm{O}(1)$ amortized rebalancing steps per update. To demonstrate this, we use the same potential function as in Section 3. In an insertion, if a search step does not do a reset, every node along the search path from the grandchild of the trailing node to the parent of the current node is a 1,1-node. Thus if we force a reset after five search steps that do not do a reset (by promoting the fifth 1,1-node in a row), each rebalancing to force a reset decreases the potential: the potential of the current node increases by one, each of the four non-terminal rebalancing steps decreases the potential by one, and the last rebalancing step increases it by at most two. A forced reset takes $\mathrm{O}(1)$ time including rebalancing. If we scale this time to be at most one, the amortized time of a forced reset is non-positive. In a deletion, if a search step does not do a reset, every node along the search path from the grandchild of the trailing node to the parent of the current node is a 2,2-node or a 1,2-node whose 1-child is a 2,2-node. If we force a reset after five search steps that do not do a reset (by doing a demotion or a double demotion at the fifth node in a row that is a 2,2-node or a 1,2-node whose 1-child is a 2,2-node), each rebalancing to force a reset decreases the potential: decreasing the rank of the current node and possibly that of its child does not increase the potential, each of the four non-terminal rebalancing steps decreases the potential by one, and the last step increases it by at most three. In either an insertion or deletion, any rebalancing at the bottom of the search path takes $\mathrm{O}(1)$ amortized time. This gives the following theorem:

**Theorem 4.** *Top-down rebalancing with sufficiently large fixed look-ahead does* $\mathrm{O}(1)$ *amortized rebalancing steps per insertion or deletion.*

Theorem 4 remains true as long as every forced reset reduces the potential. One disadvantage of top-down rebalancing is that the proof of Theorem 3 breaks down: the induction does not apply to the 2,2-nodes created by forced resets during insertions.

## 5 Rank-Based Analysis

The amortized analysis of bottom-up rebalancing in Section 3 implies that most rebalancing steps are low in the tree: in a sequence of $m$ insertions and $d$ deletions, there are $\mathrm{O}((m+d)/k)$ insertions and deletions of rank $k$ or greater. Something much stronger is true, however: for some $b > 1$, there are only $\mathrm{O}((m+d)/b^k))$ insertions and deletions of rank $k$ or greater. That is, the frequency of rebalancing steps decreases exponentially with height. This is true (and easy to prove) for weight-balanced trees if one ignores the need to update size information, but to our knowledge ours is the first such result for trees that use some form of height balance, and it covers rank changes as well as

rotations. The result also holds for top-down rebalancing with sufficiently large fixed look-ahead, for a value of $b$ that depends on the look-ahead.

It is convenient to assign potential to 1,2-nodes as well as to 1,1- and 2,2-nodes. We assign to a node of rank $k$ a potential of $\Phi_k$ if it is a 1,1- or 2,2-node, or $\Phi_{k-2}$ if it is a 1,2-node, where $\Phi$ is a non-decreasing function such that $\Phi_0 = \Phi_{-1} = 0$, to be chosen later. The potential of a tree is the sum of the potentials of its nodes.

With this choice of potential, the potential change of a sequence of non-terminal rebalancing steps telescopes. Specifically, a non-terminal insertion rebalancing step at a node of rank $k$ decreases the potential by $\Phi_k - \Phi_{k-1}$ and promotes the node to rank $k+1$. Consecutive insertion rebalancing steps are at nodes that differ in rank by one. Thus a sequence of non-terminal insertion rebalancing steps starting at a node of rank 0 and ending at a node of rank $k$ decreases the potential by $\Phi_k - \Phi_{-1} = \Phi_k$, since $\Phi_{-1} = 0$. A non-terminal deletion rebalancing step at a node of rank $k$ decreases the potential by $\Phi_k - \Phi_{k-3}$ if it is a demotion of a non-2,2-node and by $\Phi_{k-1} + \Phi_{k-2} - \Phi_{k-2} - \Phi_{k-3} = \Phi_{k-1} - \Phi_{k-3}$ if it is a double demotion. Since $\Phi$ is non-decreasing, the potential decrease is at least $\Phi_{k-1} - \Phi_{k-3}$ in either case. Consecutive deletion rebalancing steps are at nodes that differ in rank by two. If the first rebalancing step is a demotion of a 2,2-node of rank one, the step does not change the potential, because the demoted node was a 1,2-node of rank one before the deletion. Thus a sequence of non-terminal deletion rebalancing steps starting at a node of rank 1 or 2 and ending at a node of rank $k$ decreases the potential by at least $\Phi_{k-1} - \Phi_0 = \Phi_{k-1}$.

We can compute the total potential change caused by an insertion or deletion by combining the effect of the sequence of non-terminal rebalancing steps with that of the initialization and the terminal step. In an insertion, the initialization consists of adding a new leaf, which has potential $\Phi_0 = 0$. Let $k$ be the rank of the insertion. Consider the last rebalancing step. (See Figure 2.) Suppose this step is a stop. If $p$ is null, then either the insertion promotes the root and decreases the potential by at least $\Phi_k$, or $k = 0$, the insertion is into an empty tree, and it does not change the potential. If $p$ is not null but becomes a 1,2-node, the insertion decreases the potential by at least $\Phi_k$; if $p$ is not null but becomes a 1,1-node, the insertion increases the potential by at most $\Phi_k - 2\Phi_{k-2}$. Suppose the last step is a rotation. Then the insertion increases the potential by at most $\Phi_k - 2\Phi_{k-2}$. Finally, suppose the last step is a double rotation. Then the insertion increases the potential by at most $\Phi_k - 2\Phi_{k-2}$: node $t$ is a 1,1- or 1,2-node before the last step. In all cases the potential increase is at most $\max\{-\Phi_k, \Phi_k - 2\Phi_{k-2}\}$.

In a deletion, the initialization consists of deleting a leaf, or deleting a node with one child and replacing it by its child. The deleted node has potential zero before it is deleted. Let $k$ be the rank of the deletion. Consider the last rebalancing step. (See Figure 3.) Suppose this step is a stop. If $p$ is null, then either the deletion demotes the root and decreases the potential by at least $\Phi_{k-1}$, or $k \leq 1$ and the deletion deletes the root and does not change the potential. If $p$ is not null but becomes a 1,2-node, the deletion decreases the potential by at least $\Phi_k$; if $p$ is not null but becomes a 2,2-node, the deletion increases the potential by at most $\Phi_k - 2\Phi_{k-2}$. Suppose the last step is a rotation. Then the deletion increases the potential by at most $\Phi_{k-3} - \Phi_{k-1} - \Phi_{k-3} \leq 0$ if node $s$ is a 1,1-node, by at most $\Phi_{k-1} - 2\Phi_{k-3}$ if $s$ is a 1,2-node of rank at least two, and by $\Phi_2$ if $s$ is a 1,2-node of rank one; in the third case, node $p$ is a leaf after

the rotation and is demoted. Finally, suppose the last step is a double rotation. Then the deletion increases the potential by at most $\Phi_k - 2\Phi_{k-3}$ if node $u$ is a 1,1- or 1,2-node before the last step or by at most $\Phi_k + 2\Phi_{k-4} - 2\Phi_{k-2} - 2\Phi_{k-3} \leq \Phi_k - 2\Phi_{k-2}$ if node $u$ is a 2,2-child before the rotation. In all cases the potential increase is at most $\max\{-\Phi_{k-1}, \Phi_k - 2\Phi_{k-3}\}$.

For $i \geq 1$, let $\Phi_i = b^i$, where $b = 2^{1/3}$. An insertion or deletion of rank at most 3 increases the potential by $\mathrm{O}(1)$. Since $b^2 - 2 < 0$ and $b^3 - 2 = 0$, an insertion or deletion of rank 4 or more does not increase the potential. We prove that insertions and deletions of a given rank occur exponentially infrequently by stopping the growth of the potential at a corresponding rank. Specifically, for a fixed rank $k \geq 4$ and arbitrary $i \geq 1$, let $\Phi_i = b^{\min\{i,k-3\}}$. Then an insertion or deletion of rank at most 3 still increases the potential by $\mathrm{O}(1)$, and an insertion or deletion of rank greater than 3 and less than $k$ still does not increase the potential, but an insertion or deletion of rank $k$ or greater decreases the potential by at least $b^{k-3}$. This gives the following theorem:

**Theorem 5.** *In a sequence of $m$ insertions and $d$ deletions with bottom-up rebalancing in an initially empty rank-balanced tree, there are $\mathrm{O}((m+d)/2^{k/3})$ insertions and deletions of rank $k$ or more, for any $k$.*

The base of the exponent in Theorem 5 can be increased to $1.32+$ by separately analyzing insertions and deletions (proof omitted). A result similar to Theorem 5 holds for top-down rebalancing (proof omitted):

**Theorem 6.** *A sequence of $m$ insertions and $d$ deletions with top-down rebalancing in an initially empty tree does $\mathrm{O}((m+d)/b^k)$ rebalancing steps at nodes of rank $k$ if forced resets occur after six search steps in an insertion, four in a deletion, where $b = 1.13+$. The base $b$ can be increased arbitrarily close to $2^{1/3}$ by increasing the fixed lookahead.*

It is possible to improve the base in both Theorem 5 and Theorem 6 at the cost of making deletion rebalancing a little more complicated, specifically by changing the double rotation step of deletion rebalancing to promote $p$ if it is a 1,1-node after the step, or promote $s$ if it but not $p$ is a 1,1-node after the step. With this change Theorem 5 holds for a base of $2^{1/2}$, and Theorem 6 holds for a base of $b = 1.17+$ even if deletion does a forced reset after only three search steps. By increasing the fixed lookahead, the base in Theorem 6 can be increased arbitrarily close to $2^{1/2}$. Unfortunately this change in deletion rebalancing invalidates the proof of Theorem 3.

## 6 Rank-Balanced Trees versus Red-Black Trees

Rank-balanced trees have properties similar to those of red-black trees but better in several respects. Rank-balanced trees are a proper subset of red-black trees (proof omitted):

**Theorem 7.** *The nodes of an rb-tree can be assigned colors to make it a red-black tree. The nodes of a red-black tree can be assigned ranks to make it an rb-tree if and only if it does not contain a node $x$ such that there is a path of all black nodes from $x$ to a leaf and another path of nodes alternating in color from $x$ to a red leaf.*

The height bound for AVL trees holds for rb-trees as long as there are no deletions, and holds in weakened form even with deletions (Theorem 3) if rebalancing is bottom-up. On the other hand, the height of a red-black tree can be $2 \lg n - \mathrm{O}(1)$ even without deletions. Red-black trees need up to three rotations per deletion, rb-trees only two.

We conclude that the differences between rb-trees and red-black trees favor rb-trees, especially the height bound of Theorem 3. Guibas and Sedgewick, in their classic paper on red-black trees [7], considered in passing the alternative of allowing rank differences 1 and 2 instead of 0 and 1, but said, "We have chosen to use zero weight links because the algorithms appear somewhat simpler." Our results demonstrate the advantages of the alternative. We think that rank-balanced trees will prove efficient in practice, and we intend to do experiments to investigate this hypothesis.

## References


1. G. M. Adel'son-Vel'skii and E. M. Landis. An algorithm for the organization of information. *Sov. Math. Dokl.*, 3:1259–1262, 1962.
2. A. Andersson. Balanced search trees made simple. In *WADS*, volume 709, pages 60–71, 1993.
3. R. Bayer. Binary B-trees for virtual memory. In *SIGFIDET*, pages 219–235, 1971.
4. R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Inf.*, 1:290–306, 1972.
5. M. R. Brown. A storage scheme for height-balanced trees. *Inf. Proc. Lett.*, pages 231–232, 1978.
6. C. C. Foster. A study of avl trees. Technical Report GER-12158, Goodyear Aerospace Corp., 1965.
7. L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *FOCS*, pages 8–21, 1978.
8. D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 1973.
9. J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. *SIAM J. on Comput.*, pages 33–43, 1973.
10. H. J. Olivié. A new class of balanced search trees: Half balanced binary search trees. *ITA*, 16(1):51–71, 1982.
11. R. Sedgewick. Left-leaning red-black trees. www.cs.princeton.edu/ rs/talks/LLRB/LLRB.pdf.
12. S. S. Skiena. *The Algorithm Design Manual*. Springer-Verlag, 1998.
13. R. E. Tarjan. Updating a balanced search tree in $O(1)$ rotations. *Inf. Proc. Lett.*, 16(5):253–257, 1983.
14. R. E. Tarjan. Amortized computational complexity. *SIAM J. Algebraic and Disc. Methods*, 6:306–318, 1985.
15. R. E. Tarjan. Efficient top-down updating of red-black trees. Technical Report TR-006-85, Department of Computer Science, Princeton University, 1985.