Appeared in ***20th ACM-SIAM Symposium on Discrete Algorithms (SODA '10)***

Deletion Without Rebalancing in Balanced Binary Trees[*]

Siddhartha Sen[†] Robert E. Tarjan[‡]

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 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 $O(\log\log m)$ bits of balance information per node, where m is the number of insertions, or $O(\log\log n)$ with periodic rebuilding, where n is the number of nodes. An insertion takes up to two rotations and $O(1)$ amortized time. 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.

1 Introduction

Here is the true story that motivated this work, fictionalized to protect the parties involved. An on-line business contracted with a database provider to build a real-time database to store customer information, to be queried each time a customer interacted with the business, and to be updated on a regular basis. The database provider decided to use a red-black tree [9] to store the database, but implemented rebalancing only after insertions, not after deletions. As a safety check, a height limit of 80 was placed on the allowed height of the tree; in a valid red-black tree, a height bound of 80 would allow storage of 2^{40} records, 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. This process, too, caused the height bound to be exceeded, and this cycle repeated, causing an extended service outage.

Aside from the legal questions this incident raised, it raises a theoretical one as well: does it make any sense to try to maintain balance in a search tree by rebalancing only after insertions, not after deletions? Before considering this question, we review some of the literature concerning deletion in balanced trees. Such a review provides insight into how the unfortunate 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 AVL trees after an insertion, by doing one or two rotations and updating height information in $O(\log n)$ nodes, where n is the total number of nodes. An algorithm for rebalancing after a deletion appeared several years later, in a technical report by a different author [7]; deletion requires $O(\log n)$ rotations rather than $O(1)$. For all existing forms of balanced trees, of which there are many [3, 4, 5, 9, 10, 12, 13, 16], deletion is at least a little more complicated than insertion, although for some kinds of balanced search trees, notably red-black trees [9] and the recently introduced rank-balanced trees [10], rebalancing after a deletion can be done in $O(1)$ rotations. Many textbooks describe algorithms for insertion but not deletion. At least one well-known database system, Berkeley DB [14, 15], uses a B^+ tree with underfilled nodes that is rebalanced after insertions but not deletions. Thus it was perhaps natural to try the same thing for red-black trees. But disaster ensued.

A more precise version of our question is this: can one maintain a 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 [18]), 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.

[*] 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.

[†]Princeton University, `sssix@cs.princeton.edu`.

[‡]Princeton University and HP Labs, `ret@cs.princeton.edu`.

We realize this possibility. 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 O(1) and most of the rebalancing occurs deep in the tree. These results hold for bottom-up rebalancing; we extend them to top-down rebalancing with finite look-ahead as well. To obtain our bounds we use an exponential potential function, an idea we have also used [10] to analyze rank-balanced trees. Perhaps surprisingly, we obtain better constant factors for many of our bounds than the corresponding bounds for rank-balanced trees. Thus not only does rebalancing after deletions complicate the implementation, it makes the performance of the data structure worse in some ways.

It is natural to ask whether one can obtain similar results for multiway trees, in particular B trees or B^+ trees. The answer is yes, and indeed B^+ trees with underfilled nodes have the desired properties, as we show in a companion paper [17].

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 $\lg\lg n$ + O(1) with periodic rebuilding) [1], rather than the one bit per node needed in AVL [2], rank-balanced [10], and red-black trees [9]. Indeed, we provide evidence to suggest that O(1) bits do not suffice. (We leave rigorous resolution of this question as an open problem.) We conclude that the effort 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 defines ravl trees and describes bottom-up rebalancing after an insertion; the rebalancing algorithm is that of AVL trees, extended to ravl trees. Section 4 analyzes the amortized efficiency of bottom-up rebalancing. Section 5 describes and analyzes top-down rebalancing with fixed look-ahead. Section 6 describes a way to rebuild the tree efficiently if it becomes unbalanced. Section 7 examines other ways of handling insertions and deletions, and gives examples showing that natural methods that use one balance bit per node fail. Section 8 explores the pros and cons of rebalancing after deletions. Section 9 gives some preliminary experimental results. Section 10 contains final remarks.

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

2 Tree Terminology

Our tree terminology is the same as in [10]. 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 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. 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 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. 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 the number of insertions in a sequence of intermixed searches, insertions, and deletions.

3 Relaxed AVL Trees

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*[2] *tree*, is a ranked binary tree that obeys the following *rank rule*: every rank difference is positive.

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 q replaces a missing node and has a rank of zero. If the parent p of the new leaf was itself a leaf before the insertion, the new leaf 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 1.) To restore the rank rule, we let q and p be the new leaf and its parent, respectively; p is null if q is the root. We repeat the following step until a case other than a promotion occurs (see Figure 2):

Rebalancing Step at q:

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

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

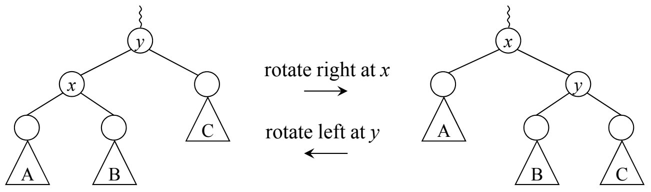

Figure 1: Rotation. Triangles denote subtrees.

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

Promote: Node s is a 1-child. Promote p. Now p is a 1,2-node, but it may be a 0-child.

In the remaining cases s is not a 1-child and q is a 1,2-node. Assume q is the left child of p; the other possibility is symmetric. Let t be the right child of q; t may be missing.

Rotate: Node t is a 2-child. Rotate at q and demote p. Now no node is a 0-child. Stop.

Doubly Rotate: Node t is a 1-child. Rotate twice at t, making q its left child and p its right child. Promote t and demote p and q. Now no node is a 0-child. Stop.

The *rank* of a rebalancing step is the rank of q just before the step. During rebalancing there is at most one violation of the rank rule: q may be a 0-child. The first step is either a stop or a promotion. In each subsequent step, q is a 1,2-node. Each step has rank one higher than that of the previous step. After a rotation, q is a 1,1-node; so is p, if it was a 1,2-node before the insertion. After a double rotation, t and at most one of p and q is a 1,1-node, unless p and q have rank zero, in which case they are both 1,1-nodes.

To delete a leaf in a ravl tree, we replace it by a missing node. To delete a node with one child, we remove it 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 a node with two children, we swap it with its symmetric-order predecessor or successor, making it into a leaf or a node with one child. We swap the ranks of the swapped nodes, and proceed as in the deletion of a leaf or a node with one child. In a deletion no rotations occur, and no ranks change except that swapped nodes swap ranks.

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 standard bottom-up rebalancing algorithm for AVL trees. All the results we shall

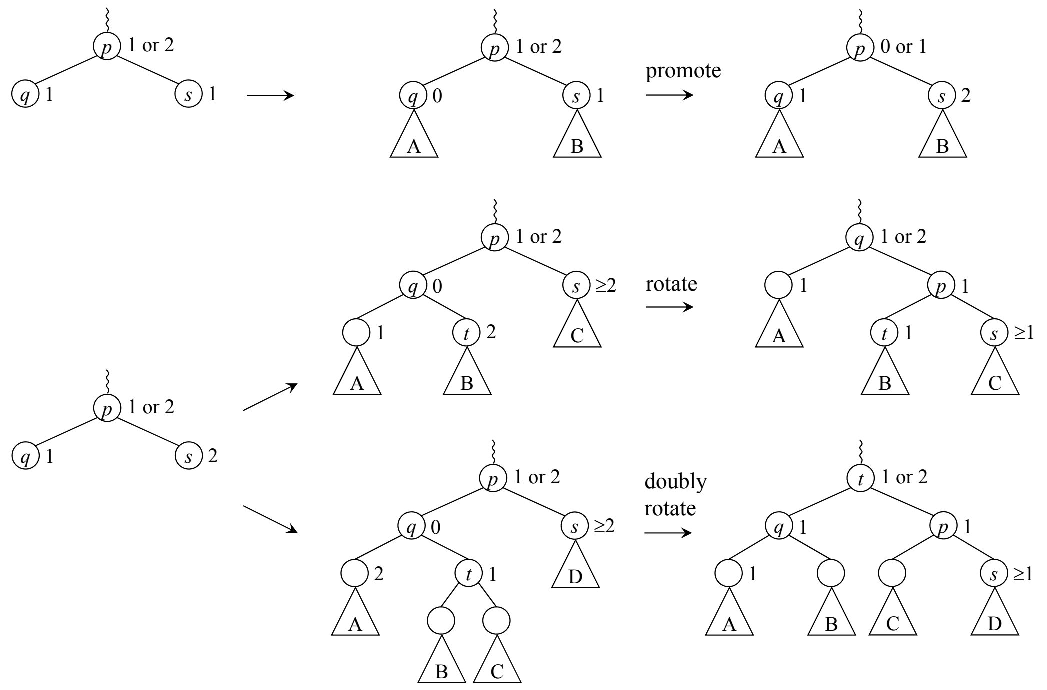

Figure 2: Rebalancing after an insertion. Numbers are rank differences. The first case is non-terminating.

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 store ranks explicitly.

Rebalancing requires traversal of the path from the new leaf toward the root. There are several ways to support this. One is to add parent pointers, or to use another representation that supports parental access; two pointers per node suffice to allow access to the children and parent of each node [8]. Another is to store the access path during the search from the root for the insertion location, either in an auxiliary stack or by reversing the child pointers along the search path. A third, which requires only one extra pointer and does not change the tree structure, is to maintain a *trailing node* during the search for the insertion position. This is the topmost node that will be affected by rebalancing, namely the parent of the nearest ancestor of the current node of the search that is neither a 1-child nor a 1,1-node; if there is no such node, the trailing node is the root. We initialize the trailing node to be the root and change it to the parent of the current node of the search each time the current node is neither a 1-child nor a 1,1-node. Once the search reaches the bottom of the tree, we do rebalancing steps (modified appropriately) top-down from the trailing 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 5.

4 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 two rotations and $O(h+1)$ rebalancing steps. All these bounds are worst-case. To obtain better bounds for rebalancing and to bound the height of the tree, we use the potential method of amortized analysis [19]. To each state of the data structure we assign a non-negative *potential*, 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.

Our first, simple amortization argument shows that each insertion takes O(1) amortized rebalancing steps. We define the potential of a node to be one if it is either a 0,1-node or a 1,1-node of positive rank, and zero otherwise. We define the potential of a tree to be the sum of the potentials of its nodes. We define the cost of a rebalancing step to be one. A deletion cannot increase the potential since it cannot create a 0-child or a 1,1-node. Consider an insertion. Adding a new leaf increases the potential by at most one, by creating a 0,1-node or a 1,1-node of positive rank. A promotion step that makes a node q a 0-child of its parent p decreases the potential by one: q becomes a 1,2-node, decreasing its potential by one, and it becomes a 0-child, which does not change the potential of p. Thus such a promotion step has an amortized cost of zero. This is also true of a promotion step that promotes the root. A promotion step that is followed by a stop does not increase the potential: it reduces the potential of the promoted node by one and can make its parent a 1,1-node, increasing the potential by one. A stop has no effect on the potential. A rotation or double rotation can increase the potential by at most two, by creating two new 1,1-nodes of positive rank. We conclude that the amortized cost of an insertion is at most four: adding the new leaf increases the potential by at most one; if the last step is a stop it and the next-to-last step have an amortized cost of at most two; if the last step is a rotation or double rotation it has an amortized cost of at most three; every promotion step has an amortized cost of zero unless it is followed by a stop. This gives the following theorem:

THEOREM 4.1. *Starting with an empty ravl tree, a sequence of m insertions with bottom-up rebalancing intermixed with arbitrary deletions does at most $4m$ rebalancing steps.*

The proof of Theorem 4.1 also gives the following generalization:

THEOREM 4.2. *Starting with an arbitrary ravl tree containing g 1,1-nodes of positive rank, a sequence of m insertions with bottom-up rebalancing intermixed with arbitrary deletions does at most $4m + g$ rebalancing steps.*

An exponential potential function of the kind first used by us to analyze another kind of balanced tree [10] gives our most important result: 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 [11]. We 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 has a 0-child but is not a 0,1-node, F_k if it is a 1,1-node, and zero otherwise. We define the potential of a tree to be the sum of the potentials of its nodes. We call this the *Fibonacci potential.*

A deletion cannot increase the potential. Adding a new leaf increases the potential by at most one, either by creating a new 1,1-node of rank one or by creating a new 0,1-node of rank zero, since $F_1 = F_2 = 1$. Consider a rebalancing step of rank k. A promotion of p that makes its parent x a 0,1-node does not change the potential: the potential of x and p, respectively, is F_{k+1} and F_{k+2} before the step, and $F_{k+3} = F_{k+1} + F_{k+2}$ and zero after. A promotion that makes p a 0-child but does not make its parent x a 0,1-node also does not change the potential: the potential of x and p, respectively, is zero and F_{k+2} before the step, and F_{k+2} and zero after. Likewise, a rotation does not increase the potential: nodes p and q have potential $F_{k+1} = F_k + F_{k-1}$ and zero before the step and F_k and at most F_{k-1} after. Neither does a double rotation: the total potential of p, q, and t is F_{k+1} before the step and at most $F_k + F_{k-1} = F_{k+1}$ after. (Here we use $F_0 = 0$.) The final possibility is a promotion of a node p followed by a stop. If the promotion makes the parent x of p a 1,1-node, it does not change the potential: the total potential of x and p is F_{k+2} both before and after the promotion. Otherwise the promotion decreases the potential by F_{k+2}.

THEOREM 4.3. *If a ravl tree 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. Let the potential be the Fibonacci potential. The first insertion leaves the potential at zero. Each subsequent insertion increases the potential by at most one, not counting decreases resulting from promotions followed by stops. If the rank of the root is r, there was a promotion of rank i that did not create a 1,1-node followed by a stop, 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, $m-1 \geq F_{r+3}-2$. Since all rank differences are positive and the leaves have non-negative rank, $h \leq r$. Thus $m \geq F_{h+3} - 1 \geq F_{h+2} \geq \phi^h$.

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

THEOREM 4.4. *Starting from an initially empty tree, a sequence of 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. Deletions do not increase the potential. The addition of a new leaf other than the first increases the potential by at most one. No rebalancing step increases the potential. A step of rank k is preceded by a promotion of rank $k-1$. If the step of rank k is a stop or a promotion, the promotion of rank $k-1$ reduces the potential by F_{k+1}. If the step of rank k is a rotation or double rotation, it reduces the potential by at least $F_{k+1} - F_{k-1} = F_k$. Thus the potential decreases by at least F_k for each rebalancing step of rank k.

5 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 trailing node method described at the end of Section 3 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 trailing node within O(1) nodes of the current node of the search. The idea is to force a reset of the trailing 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 and its parent are 1,1-nodes, we can force a reset by promoting the current node and rebalancing from the trailing node top-down. By doing this often enough but not too often, we obtain a top-down rebalancing method with fixed look-ahead that has the properties established in Section 4 for bottom-up rebalancing (with worse constant factors). As the look-ahead increases, the efficiency of top-down rebalancing approaches that of bottom-up rebalancing.

We shall analyze the version of this method that forces a reset every time the search during an insertion visits three 1,1-nodes in a row without reaching a leaf; this is the smallest amount of look-ahead that gives the results we seek. Let the potential be four times the simple potential used to prove Theorems 4.1 and 4.2; namely, four times the number of 0,1-nodes and 1,1-nodes of positive rank. A forced reset takes four rebalancing steps and decreases the potential by at least four, since it destroys three 1,1-nodes and creates at most two, so the amortized cost of a forced reset is at most zero. Deletions do not increase the potential. Adding a leaf increases the potential by at most four. The rebalancing after a leaf is added consists of a stop, or of one, two, or three promotions followed by a terminal step. The potential increase in each case is zero, at most four, at most zero, and at most minus four, respectively. The amortized cost of an insertion is thus at most ten; the second case is the worst. This gives the following analogue of Theorem 4.2:

THEOREM 5.1. *Starting with an arbitrary tree containing g 1,1-nodes of positive rank, a sequence of m insertions with top-down rebalancing intermixed with arbitrary deletions takes at most $10m+4g$ rebalancing steps.*

To obtain analogues of Theorems 4.3 and 4.4, we use an exponential potential that grows more slowly than the Fibonacci potential. For a base $b > 1$ to be chosen later, we define the potential of 1,1-nodes of rank k to be b^k, the potential of all other nodes to be zero, and the potential of a tree to be the sum of the potentials of its nodes; nodes with a 0-child do not need potential since we shall analyze the effect of a forced reset as a unit. We want to guarantee that a forced reset does not increase the potential. Consider a forced reset whose topmost rebalancing step is of rank k. The net increase in potential caused by the forced reset is at most $b^{k+1} - b^{k-1} - b^{k-2} - b^{k-3}$ if the topmost step is a stop, at most $b^k - b^{k-2} - b^{k-3}$ if this step is a rotation or double rotation. Choose b to be the largest real root of $b^3 - b - 1 = 0$. Then a forced reset does not increase the potential. Adding a new leaf and rebalancing at the bottom of the tree increases the potential by O(1). A promotion step of rank k that does not create a 1,1-node and is followed by a stop decreases the potential by b^k. We conclude that $b^h \leq cm$ for some positive constant c, giving the following theorem:

THEOREM 5.2. *A ravl tree built from an empty tree by a sequence of m insertions with top-down rebalancing intermixed with arbitrary deletions has height at most $\log_b m + \mathrm{O}(1)$, where $b = 1.325+$ is the largest real root of $b^3 - b - 1$.*

THEOREM 5.3. *Starting from an initially empty tree, a sequence of m insertions with top-down rebalancing intermixed with arbitrary deletions does $\mathrm{O}(m/b^k)$ rebalancing steps of rank k for any $k \geq 0$, where b has the same value as in Theorem 5.2.*

Proof. The theorem is immediate for $k < 4$. Suppose $k \geq 4$. Let the potential of a node be b^j if it is a 1,1-node of rank $j < k$ and zero otherwise; let the potential of a tree be the sum of the potentials of its nodes. Deletions do not increase the potential. Adding a new leaf increases the potential by $O(1)$. Neither a forced reset nor the rebalancing after a new leaf is added can increase the potential. Consider a rebalancing step of rank k. This step must be part of a forced reset whose topmost rebalancing step is of rank at least k, and which begins with the promotion of a node of rank less than k. This forced reset reduces the potential by at least b^{k-2}.

By increasing the amount of look-ahead in top-down rebalancing, we can improve the constants in Theorem 5.1 to be arbitrarily close to those in Theorem 4.2, and we can make the base b in Theorems 5.2 and 5.3 arbitrarily close to ϕ. Indeed, if we force a reset every time the search during an insertion visits four 1,1-nodes in a row without reaching a leaf, then Theorems 5.2 and 5.3 hold for b the largest positive root of $b^4 - b^2 - b - 1$, which exceeds $\sqrt{2}$.

6 Rebuilding the Tree

As the ratio of the number of deletions to the number of insertions approaches one, the height of a ravl tree can become $\omega(\log n)$, although it remains $O(\log m)$. For many applications this is not a concern, but for those in which it is, we can keep the height $O(\log n)$ by periodically rebuilding the tree. How to do the rebuilding, and how often, are interesting questions that deserve careful study. Here we offer a simple rebuilding method and some thoughts on how often to rebuild.

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 $O(n)$ time. To facilitate building the new tree, we store the nodes on its right spine on a stack, bottommost node on top. Each insertion into the new tree takes $O(1)$ amortized time, and rebuilding the entire tree takes $O(n)$ time. The new tree has height at most $\lg n + 1$: every child is a 1- or 2-child, and the 2-children have parents on the right spine. The new tree also has potential $O(n)$ for any of the potential functions we have considered.

To decide when to rebuild the tree, we keep track of n and of the rank r of the root. If we are using bottom-up rebalancing, we rebuild the tree whenever $r > \log_\phi n + c$, where c is a small positive constant. Then the rebuilding time is $O(1/(\phi^c - 1))$ per deletion. The larger we choose c, the smaller the overhead for rebuilding, but the larger the height can become as a function of n. If we allow c to grow as a function of n, we can make the rebuilding time $o(1)$ per deletion while still maintaining a height bound of $\log_\phi n$ plus a lower-order term. If we are using top-down rebalancing, we rebuild the tree whenever $r > \log_b n + c$, where b is the base in Theorem 5.2 and c is larger than the additive constant in Theorem 5.2.

We can also make the rebuilding incremental. For example, we can start the rebuilding when the height bound is violated and move two nodes from the old to the new tree after each insertion or deletion. During rebuilding, we do each insertion in the old or new tree as appropriate: such an operation is in the new tree if the key of the new item is at most that of the last item moved, in the old tree otherwise. We store the left spine of the old tree and the right spine of the new tree in stacks, so that the next node to be deleted from the old tree, and its insertion location in the new tree, can be found in $O(1)$ time. We must update these stacks during insertions and deletions, but this takes $O(1)$ amortized time per insertion or deletion. If n is the number of items in the old tree when rebuilding starts, the number in the new tree will be between $n/2$ and $2n$ when rebuilding stops.

Whether the tree is rebuilt incrementally or all at once, the tree height is always at most $\log_\phi n + O(1)$ with bottom-up rebalancing or $\log_b n + O(1)$ with top-down rebalancing, and Theorems 4.2 and 4.4, or 5.1 and 5.3 hold, respectively.

7 Good and Bad Alternatives

Each node in a ravl tree stores $\Omega(\log \log n)$ bits of balance information, rather than the one bit per node needed in other kinds of balanced search trees. It is natural to ask whether so many bits are necessary in ravl trees; or, stated differently, whether there exists a variant of ravl trees that stores only one bit per node.

All the results we have presented rely on three properties: if a new leaf is a 0-child before rebalancing, it has rank zero, if the parent of a new leaf is a 1,1-node, it has rank 1, and deletions do not increase any rank. Thus all the results still hold if we modify insertions so that a leaf added by an insertion has rank at least zero and at most the maximum of zero and the rank of its parent minus two, or the rank of its parent minus one if the other child of its parent has rank difference at least two. We can also modify deletions so that the rank of a new leaf (created by deleting its only child) is decreased by any non-negative amount up to its old rank, giving it a new rank between zero and its old rank, inclusive.

As we noted earlier, one balance bit can encode a rank difference of 1 or 2 in non-root nodes and requires storing the rank of the root explicitly to determine actual ranks along a path originating at the root. The modification to insertions described above is compatible with this storage scheme, but the modification to deletions is not. During an insertion, if we give a new leaf

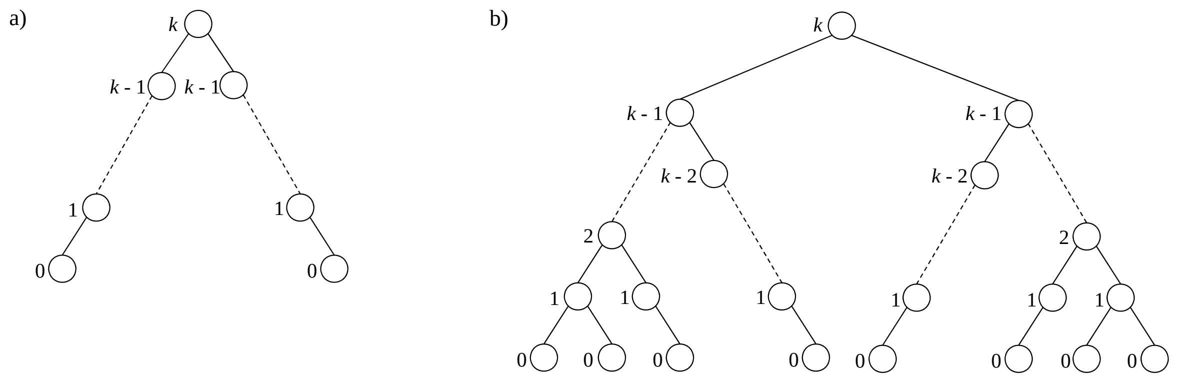

Figure 3: Counterexamples for two alternative methods of insertion and deletion that use one balance bit per non-root node.

the maximum rank allowed by the modification above, then one bit suffices to store the rank difference of the new leaf, and the parent of the new leaf can only be a 1,1-node if its rank is 1. During a deletion, if we decrease the rank of a new leaf, however, then its rank difference may exceed 2; at the same time, allowing ranks to increase may create 1,1-nodes of higher rank. In fact, modifications of the insertion method that create a 0-child or a 1,1-node of arbitrary rank before rebalancing, or modifications of the deletion method that increase ranks, invalidate the proofs of Theorems 4.3 and 5.2. Indeed, we describe two such modifications and construct a sequence of updates for each that produces trees of height a fractional power of m.

Suppose that we leave deletion unmodified but modify insertion to give the new leaf a rank difference of one unless that would make its rank negative, in which case the leaf gets a rank difference of zero. The following class of sequences builds trees of height $\Omega(\sqrt{m})$. (See Figure 3a.) Suppose we have built a tree of rank k consisting of a left and right spine, with each child having rank difference one and the two leaves having rank zero. Do $2(k-1)$ insertions to give every non-leaf a second child of rank difference one. Then do two insertions of children of the two leaves. The first such insertion 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 a tree of rank $k+1$ of the same type.

Suppose that we leave insertion unmodified but modify deletion so that when a node with one child is deleted, its child (which replaces it) gets a rank difference of two. This delays the problem illustrated in the previous construction but does not avoid it: the following class of sequences builds trees of height $\Omega(m^{1/3})$. (See Figure 3b.) Suppose we have built a tree of rank k in which every child is a 1-child, every leaf has rank zero, every node on the left or right spine has two children, and the other non-leaves have one child. Do two insertions of children of the leaves on the left and right spines. Now the tree has rank $k+1$, every child on the left and right spines is a 1-child, the leaves on the spines have rank zero, and every non-leaf on the spine of rank two or more has a child of rank difference two from which a path of 1-children descends to a leaf of rank zero. Do two insertions that add 1-children to the two nodes on the spines of rank one. For each of the two nodes of rank zero and rank difference two, do an insertion to add a child; 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, whose rank increases by one. The leaf at the bottom of the path descending from this node now has rank one. Do two insertions to make this node a 1,1-node with two leaves of rank zero as children. Delete the new 2-child; choose one of the new leaves of rank one and by two insertions make it a 1,1-node with two leaves of rank zero as children. Continue in this way until the remaining 2-child is a 1,1-node. Add one leaf at the bottom of the path, causing promotions all along the path and making the path into a path of 1-children descending to a leaf of rank zero. Delete all the 2-children of nodes on this path. Repeat this construction for every 2-child of a node on the left or right spine. The result is a tree of rank $k+1$ of the same type. The number of insertions and deletions needed to increase the rank by one is $O(k^2)$.

The latter counterexample suggests (but does not prove) that obtaining a height bound logarithmic in the number of insertions, while using one bit per non-root node, requires a deletion method that does not increase any ranks. Thus if we guarantee that all deletions occur at leaves, then we can evade the above counterexample

Test	Red-black trees				Rank-balanced 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	# rots $\times 10^6$	# bals $\times 10^6$	avg plen	max plen
1. Random	26.443	116.070	10.472	15.627	29.553	133.737	10.390	15.092	14.315	80.606	11.114	16.753
2. Queue	50.317	285.129	11.375	22.501	50.325	184.527	11.195	13.999	33.550	134.218	11.376	13.999
3. Working set	41.714	185.348	10.510	16.181	43.686	159.689	10.445	15.345	28.003	119.924	11.202	16.641
4. Static Zipf	25.238	112.858	10.413	15.458	28.272	130.927	10.338	15.045	13.480	78.029	11.117	17.683
5. Dynamic Zipf	23.176	103.472	10.477	15.661	26.038	125.985	10.404	15.158	12.656	74.275	11.110	16.841

Table 1: Performance comparison of red-black, rank-balanced, and ravl trees on typical input sequences (rots = rotations, bals = balance information updates, avg plen = average path length, max plen = maximum path length).

because such a deletion does not affect the rank of any other node in the tree. We can in fact do all the deletions at the leaves, but a single deletion may require a number of swaps, rather than just one. To delete an item with two children, swap it with its successor. Now the item has at most one child, a right child. If it has a right child, swap it with the item at the bottom of the left spine of this child (the successor of the successor). Repeat such swaps until the item to be deleted is a leaf; then delete it. To delete an item with only a right child, proceed in the same way; to delete an item with only a left child, proceed symmetrically. The time for a deletion is $O(h + 1)$. The structure of the tree remains the same except for the loss of a leaf. If we use this deletion method and we modify insertions so that a leaf added by an insertion has rank the maximum of zero and the rank of its parent minus two, or the rank of its parent minus one if its sibling has rank difference at least two, we obtain a variant of ravl trees in which every node has rank difference 1 or 2, every node is a 1,1-node or a 1,2-node or has a missing child, and the bounds we have derived for ravl trees still hold. To represent ranks, we need one bit per node plus the rank of the root. We have no good bound on the number of swaps during deletions, however, which could be proportional to $\log m$ per deletion. Whether this can be reduced to $O(1)$ amortized per deletion via some periodic rebuilding scheme is a question we leave for further study.

8 To Rebalance on Deletion or Not?

Let us compare ravl trees to standard kinds of balanced search trees. Deletion is much simpler in ravl trees. Ravl trees need $O(\log \log m)$ balance bits per node instead of the one bit per node needed by AVL trees, rank-balanced trees, and red-black trees. Their height is at most $\log_\phi m$, versus $\log_\phi n$ for AVL trees, $2 \lg n$ for red-black trees, and $\min\{\log_\phi m, 2 \lg n\}$ for rank-balanced trees. With periodic rebuilding, at a cost of $O(1)$ per update, a height bound of $\log_\phi n + O(1)$ can be maintained, and only $O(\log \log n)$ balance bits per node are needed. Rebalancing after an insertion takes $O(1)$ rotations worst-case and $O(1)$ amortized time, and nodes are affected by rebalancing with a frequency exponentially small in their height. These results hold for rank-balanced trees and red-black trees, but the constant factors are worse [10]; they do not hold for AVL trees subject to intermixed insertions and deletions. All these comparisons favor ravl trees, except for the extra space needed for balance information. At least some authors, however, have suggested storing complete rank information for balanced trees, claiming that it simplifies rebalancing [3]. Doing this eliminates the space advantage of balanced trees.

9 Experimental Results

In our preliminary experiments, we compared ravl trees (without periodic rebuilding) to standard red-black trees [9] and rank-balanced trees [10] on typical input sequences. Our results show that ravl 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 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 [6, 20] 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 distribution 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 1 show that ravl trees performed significantly fewer rotations and balance information updates—over 42% and 35% fewer, on average, respectively—than red-black trees and rank-balanced trees on the tested sequences. 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, on average, respectively. Rank-balanced trees performed slightly more rotations and balance information updates than red-black trees, but maintained better average and maximum path lengths.

We are in the process of conducting more thorough experiments on these and other balanced tree implementations, such as left-leaning red-black trees [5, 16]. 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.

10 Remarks

We have shown that one can obtain logarithmic worst-case search time in a binary tree without rebalancing after deletions, but this seems to require storing $\Omega(\log\log n)$ balance bits per node. Proving this (or disproving it) would be an interesting theoretical result. Our new data structure, the ravl tree, shows promise as a simple and efficient implementation of binary search trees, and we plan to do further experiments to evaluate its performance in practice.

References

[1] L. A. Adamic and B. A. Huberman. Zipf's law and the Internet. *Glottometrics*, 3:143–150, 2002.

[2] G. M. Adel'son-Vel'skii and E. M. Landis. An algorithm for the organization of information. *Sov. Math. Dokl.*, 3:1259–1262, 1962.

[3] A. Andersson. Balanced search trees made simple. In *WADS*, volume 709, pages 60–71, 1993.

[4] R. Bayer. Binary B-trees for virtual memory. In *SIGFIDET*, pages 219–235, 1971.

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

[6] J. B. Estoup. Gammes stenographiques., 1916.

[7] C. C. Foster. A study of AVL trees. Technical Report GER-12158, Goodyear Aerospace Corp., 1965.

[8] M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan. The pairing heap: a new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986.

[9] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *FOCS*, pages 8–21, 1978.

[10] B. Haeupler, S. Sen, and R. E. Tarjan. Rank-balanced trees. In *WADS*, pages 351–362, 2009.

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

[12] J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. *SIAM J. on Comput.*, pages 33–43, 1973.

[13] H. J. Olivié. A new class of balanced search trees: Half balanced binary search trees. *ITA*, 16(1):51–71, 1982.

[14] M. A. Olson, K. Bostic, and M. I. Seltzer. Berkeley DB. In *USENIX Annual, FREENIX Track*, pages 183–191, 1999.

[15] M. A. Olson, K. Bostic, and M. I. Seltzer. Berkeley DB, 2000.

[16] R. Sedgewick. Left-leaning red-black trees. www.cs.princeton.edu/˜rs/talks/LLRB/LLRB.pdf.

[17] S. Sen and R. E. Tarjan. Deletion without rebalancing in multiway search trees. In *ISAAC*, 2009. To appear.

[18] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.

[19] R. E. Tarjan. Amortized computational complexity. *SIAM J. Algebraic and Disc. Methods*, 6:306–318, 1985.

[20] G. K. Zipf. *Selected studies of the Principle of Relative Frequency in Language.* Harvard Univ. Press, 1932.