Journal of Algorithms **30,** 1–18 (1999)
Article ID jagm.1998.0967, available online at http://www.idealibrary.com on IDEAL®

General Balanced Trees

Arne Andersson*

Department of Computer Science, Lund University, Lund S221 00 Sweden

Received November 26, 1990; revised July 16, 1998

We show that, in order to achieve efficient maintenance of a balanced binary search tree, no shape restriction other than a logarithmic height is required. The obtained class of trees, general balanced trees, may be maintained at a logarithmic amortized cost with no balance information stored in the nodes. Thus, in the case when amortized bounds are sufficient, there is no need for sophisticated balance criteria.

The maintenance algorithms use partial rebuilding. This is important for certain applications and has previously been used with weight-balanced trees. We show that the amortized cost incurred by general balanced trees is lower than what has been shown for weight-balanced trees. © 1999 Academic Press

1. INTRODUCTION

One of the fundamental data structures in computer science is the binary search tree. New methods to maintain data in search trees have been developed and thoroughly analyzed all through the history of the discipline. Attention has mainly been focused on trees with bounded height or *balanced trees*. The reason for this is obvious since worst-case access time is proportional to the height of a tree.

The traditional way to maintain balance at a low cost is by means of some more or less sophisticated balance criterion. To illustrate the large variation in the world of balance criteria some examples are given below. We use $|T|$ to denote the number of leaves (weight) of the tree T.

- *AVL-trees*, introduced by Adelson-Velskii and Landis [1], are defined by a balance criterion requiring the heights of the two subtrees of each node to differ by at most 1. AVL-trees have a maximum height of $1.44 \log|T|$.

* E-mail: arne@dna.lth.se.

- *Symmetric binary B-trees,* or SBB-trees, were introduced by Bayer [6]. The very same class of trees often occurs under the name *red-black trees,* due to Guibas and Sedgewick [14]. The edges in an SBB-tree are of two types: horizontal and vertical. Two adjacent edges are never both horizontal and the number of vertical edges on the path from the root to a leaf is the same for all leaves. This criterion guarantees a maximum height of $2 \log|T|$.

- *Weight-balanced trees* were introduced by Nievergelt and Reingold [20]. For each node in the tree, the ratio of the weights of its two subtrees is restricted. Given a parameter α, $0 \leq \alpha < 1/2$, for each node v in the tree, the quotient between the weight of (i.e., the number of leaves in) v's smallest subtree and the weight of v itself must be at least α. The maximum height of a weight-balanced tree is $\log|T|/\log(1/(1-\alpha))$.

- *α-balanced trees* were introduced by Olivie [21]. These trees are *path-balanced.* Given a parameter α, $0 < \alpha < 1$, for each node in the tree, the quotient between the lengths of the shortest and longest outgoing paths must be at least α. This guarantees a maximum height of $(\log|T|)/\alpha$.

- *k-neighbor trees* were introduced by Maurer *et al.* [18]. They are unary−binary trees where all leaves have the same depth. The number of binary nodes between two unary nodes at the same level is at least k. A k-neighbor tree has a height of at most $\log|T|/\log(2 - 1/(k+1))$.

For each of these classes of trees, computing the maximum height from the balance criterion is a nontrivial exercise. Other examples of balanced trees are HB(k)-trees [12], one-sided height-balanced trees [15, 16], and SBB(k)-trees [4]. Note that the *splay trees,* introduced by Sleator and Tarjan [24], are not strictly balanced since the height of a splay tree T may be $\Theta(|T|)$.

A natural question that arises from the study of balanced trees is whether we really need to make a detour over those balance criteria when the only thing we want (mostly) is a logarithmic height. The disadvantage of using a sophisticated criterion is illustrated by the tree in Fig. 1. This tree has the smallest possible height with respect to its weight; however, according to the balance criteria mentioned above it is not well-balanced at all.

In this article we show that, as long as we are concerned with the amortized cost of maintenance, we can replace the criteria above by a weak global criterion; we just have to specify the relation between the size and the maximum height of the tree. Not only is the balance criterion simple; the maintenance algorithms are also simple and they can be implemented without keeping any balance information in the nodes.

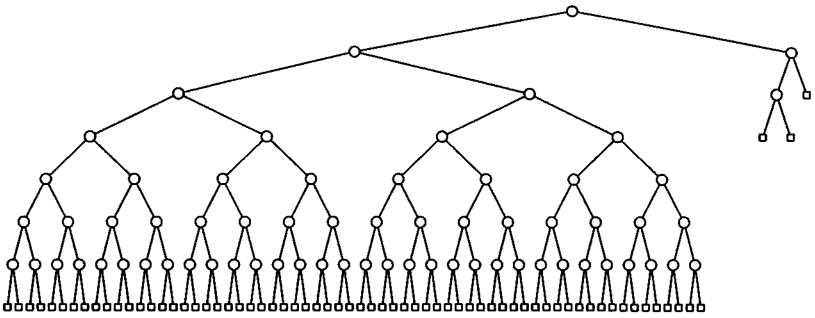

FIG. 1. A well-balanced tree?

This class of trees, called *general balanced trees*, is maintained by *partial rebuilding*. The method of partial rebuilding has been used by Overmars and van Leeuwen [22, 23] to maintain weight-balanced trees. It can also be used to maintain a modified version of α-balanced trees [3]. Partial rebuilding is an attractive method in the sense that it is useful not only for ordinary binary search trees, but also for more complicated data structures, such as multidimensional search trees [7], where other balancing methods do not work. Making a careful analysis we are able to show a lower maintenance cost of general balanced trees than what has been shown for weight-balanced trees. Thus, we improve the method of partial rebuilding in terms of maintenance cost.

A preliminary version of this article is published in Proceedings of Workshop on Algorithms and Data Structures, WADS '89, Ottawa [2]. An independent article has also been presented by Galperin and Rivest at the Fourth Annual ACM–SIAM Symposium on Discrete Algorithms, SODA '93 [13].

2. PRELIMINARIES

In the following we do not distinguish between nodes and subtrees and by the subtree v we mean the subtree rooted at the node v and by T we mean the entire tree (or the root of the tree). We deal with binary search trees where each internal node contains one element and each leaf is empty. The number of edges on the path from the root T to the node v is the *depth* of v. The largest number of edges from a node v to a leaf is the *height* of v, denoted $h(v)$. The *weight* of v equals the number of leaves in v and is denoted $|v|$. Note that a tree containing n internal nodes (or elements) has a weight of $n + 1$.

Let v_H and v_L denote node v's highest and lowest subtrees, respectively, ties are broken arbitrarily. As a measure of the shape of the subtree v we use the *weight difference* at v, $\delta(v)$, defined as

$$\delta(v) = \mathrm{Max}\big(0, |v_H| - |v_L| - 1\big). \tag{1}$$

As the basic restructuring operation, our algorithms use a procedure for rebuilding a tree to perfect balance, as defined below.

DEFINITION 1. A binary tree v is perfectly balanced if and only if $\delta(v) = 0$ and both of v's subtrees are perfectly balanced trees.

In the following we assume that rebuilding a (sub)tree v takes $O(|v|)$ time. Examples of linear algorithms for balancing trees can be found in the literature [10, 11, 13, 17, 25].

We also assume that updates are performed by adding or removing nodes at the lowest level of the tree. For each node v, an update below v changes the value of $\delta(v)$ by at most 1. Hence, at least $\delta(v)$ updates have been made in the subtree v since the last time it was perfectly balanced.

3. MAIN RESULT

The main idea in maintaining a general balanced tree is to let the tree take any shape as long as its height does not exceed $\lceil c \log|T| \rceil$ for some constant $c > 1$. When this criterion is violated, the height can be decreased by partial rebuilding at a low amortized cost. This is due to the following observation on the shape of an unbalanced tree:

LEMMA 1. *Let T be a binary tree, $h(T) > \lceil c \log|T| \rceil$. Let v be the lowest node on (any of) T's longest path(s) such that $h(v) > \lceil c \log|v| \rceil$. Then*

$$\delta(v) > (2^{1-1/c} - 1)|v| - 1. \tag{2}$$

Proof. Since v is the lowest node satisfying $h(v) > \lceil c \log|v| \rceil$ we know that $h(v_H) \le \lceil c \log|v_H| \rceil$. Hence,

$$\lceil c \log|v| \rceil < h(v) = h(v_H) + 1 \le \lceil c \log|v_H| \rceil + 1. \tag{3}$$

This gives that

$$\begin{gathered} \lceil c \log|v| \rceil < \lceil c \log|v_H| \rceil + 1, \\ |v_H| > 2^{-1/c}|v|. \end{gathered} \tag{4}$$

From the fact that $|v_L| = |v| - |v_H|$ we can compute the value of $\delta(v)$:

$$\begin{aligned}\delta(v) &= |v_H| - \big(|v| - |v_H|\big) - 1\\ &= 2|v_H| - |v| - 1\\ &> (2^{1-1/c} - 1)|v| - 1. \qquad (5)\end{aligned}$$

▮

By a straightforward application of Lemma 1 we can maintain a balanced tree efficiently during insertions.

THEOREM 1. *If no deletions are made, a binary search tree T with maximum height $\lceil c \log|T|\rceil$, $c > 1$, can be maintained at an amortized cost of $O(\log|T|)$ per insertion. No balance information is needed in the nodes, only one global integer is needed.*

Proof. We let the global integer contain the value of $|T|$. During insertion, we keep track of the depth of the insertion path. Whenever the depth of a new leaf exceeds $\lceil c \log|T|\rceil$ we back up along the path until we find the lowest node v, $h(v) > \lceil c \log|v|\rceil$. We make a partial rebuilding at v.

In order to locate the node v, we need to keep track of the heights and weights of the visited nodes as we follow the path upward. The weights are computed by explicitly counting the nodes in all subtrees along the path. The total cost of this counting is $O(|v|)$.

After the rebuilding, $h(v) = \lceil \log|v|\rceil$. This implies that the height of v, and therefore also the height of T, does not exceed the height before the insertion. Hence, if $h(T) \le \lceil c \log|T|\rceil$ held before the insertion, it will also hold after. Since the height of an empty tree is zero, the height condition holds by induction.

The cost of the rebalancing, including the cost of locating v, is $O(|v|)$. According to Lemma 1, $\delta(v)$ has been changed $\Omega(|v|)$ times since the last time v was involved in a rebuilding. Hence, by reserving a constant amount of extra time each time the weight difference is changed at a node, enough time will be saved to cover the cost of rebuilding. Since an update affects the weight difference of $O(\log n)$ nodes, the amortized cost of insertions will be $O(\log n)$. ▮

In order to handle deletions efficiently, we make a simple extension to the algorithm above; when enough deletions have been made to cover the cost, we rebuild the entire tree to perfect balance.

THEOREM 2. *Given constants $c > 1$, and $b > 0$, a balanced tree T with maximum height $\lceil c \log|T| + b\rceil$ may be maintained without any balance information stored in the nodes, using two global integers, at an amortized cost of $O(\log|T|)$ per update.*

Proof. We let the two global integers contain $|T|$, the number of leaves in $|T|$, and $d(T)$, the number of deletions made since the last time T was globally rebuilt. Updates are performed in the following way:

- *Insertion*: If the depth of the new leaf exceeds $\lceil c \log(|T| + d(T))\rceil$ we back up along the insertion path until we find the lowest node v, $h(v) > \lceil c \log|v|\rceil$, where a partial rebuilding is made.
- *Deletion*: $d(T)$ increases by one. If $d(T) \geq (2^{b/c} - 1)|T|$ we rebuild T to perfect balance and set $d(T) = 0$.

A deletion does not increase $h(T)$. Furthermore, after an insertion we are guaranteed that $h(T) \leq \lceil c \log(|T| + d(T))\rceil$, provided that this relation holds before the insertion. Hence, by induction, $h(T) \leq \lceil c \log(|T| + d(T))\rceil$. Since $d(T) < (2^{b/c} - 1)|T|$, we conclude that

$$h(T) \leq \left\lceil c \log\left(|T| + d(T)\right)\right\rceil \leq \left\lceil c \log\left((2^{b/c})|T|\right)\right\rceil = \lceil c \log|T| + b\rceil. \quad (6)$$

Rebuildings are made on two occasions: when the number of deletions gets too large and when the tree gets too high during insertion. The amortized cost for the first type of rebuilding is constant for each deletion. For the second case we use the same argument as in the proof of Theorem 1. By reserving a constant time each time an update (insertion or deletion) changes the weight difference of a node, we save enough time to cover partial rebuildings. ▮

Since the trees maintained in Theorems 1 and 2 are allowed to take any shape as long as their height is low enough, we call them *general balanced trees*. We use the notation GB-trees or GB(c)-trees, where c is the height constant used in the theorems. (The constant b in Theorem 2 is omitted in this notation.)

EXAMPLES. Figure 2 shows a GB(1.2)-tree maintained as in the proof of Theorem 2. We assume that two deletions have been made since the last global rebuilding and that node U is being inserted. The path to U is too long since $h(T) = 6 > \lceil 1.2 \log(15 + 2)\rceil = \lceil 1.2 \log(|T| + d(T))\rceil$. We have to make a partial rebuilding at one of the nodes on that path. Making a depth-first search we find that $h(Q) = 5 > \lceil 1.2 \log 10\rceil = \lceil 1.2 \log|Q|\rceil$. A partial rebuilding is made at Q and the insertion is completed. The resulting tree is shown in Fig. 3.

4. ANALYSIS II: THE CONSTANT FACTOR

Above we proved our main result: a GB-tree can be maintained at $O(\log n)$ amortized cost per update. In this section, we make a more detailed study. The purpose of this study is to show a better constant factor

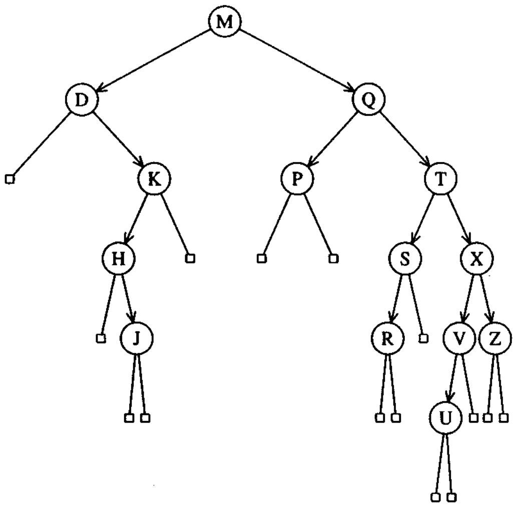

FIG. 2. A GB(1.2)-tree which requires rebalancing.

for GB-trees than what has previously been shown for weight-balanced trees.

When analyzing the constant factor, by cost we mean the amount of restructuring work needed per update. To be more precise, we let the cost of a partial rebuilding equal the number of internal nodes involved. Thus, the cost of a partial rebuilding at node v is $|v| - 1$. Other costs, such as

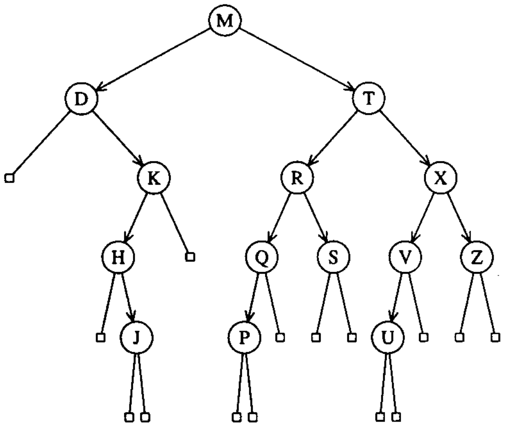

FIG. 3. The tree in Fig. 2 after a partial rebuilding.

counting sizes of trees, creating new nodes or removing nodes, etc. are ignored. The comparison with weight-balanced trees is made in Section 5.

Although this was not made in Theorem 2, the result of Lemma 1 allows us to compute a constant factor, telling an upper bound on the amount of rebalancing work spent per update. We chose to express this work in terms of the number of internal nodes involved in a rebuilding. Thus, the cost of a partial rebuilding at node v is $|v| - 1$. According to Lemma 1, when a partial rebuilding is to be made at node v, we have that

$$\delta(v) > (2^{1-1/e} - 1)|v| - 1. \tag{7}$$

Thus, the amortized cost of changing the δ-value of a node during an update is $1/(2^{1-1/e} - 1)$. Since an update is made below at most $\lceil c \log|T|\rceil$ nodes, the amortized cost per update in the entire tree is $\lceil c \log|T|\rceil/(2^{1-1/e} - 1)$.

The improved analysis is based on the fact that when a node becomes unbalanced there is not only a difference in weight between its two subtrees but also a certain imbalance at the lower levels of the tree.

In order to measure this imbalance we define a function $\sigma(v)$ as follows:

$$\sigma(v) = \begin{cases} 0, & \text{if } v \text{ is a leaf,} \\ \delta(v) + \sigma(v_H) + \sigma(v_L), & \text{otherwise,} \end{cases} \tag{8}$$

where $\delta(v)$ is defined as in the previous section. The imbalance along the longest path from a node v can be expressed using the function $\beta(v)$, defined below:

$$\beta(v) = \begin{cases} 0, & \text{if } v \text{ is a leaf,} \\ \delta(v) + \beta(v_H), & \text{otherwise.} \end{cases} \tag{9}$$

From the definitions it follows that

$$\delta(v) \leq \beta(v) \leq \sigma(v). \tag{10}$$

Below, in Lemma 5, we compute the value of $\sigma(v)$ when v is about to be rebuilt. In order to do that, we first show which configuration gives the lowest possible value of $\beta(v)$ in Lemmas 3 and 4. We say that a node v has *minimal weight* if decreasing its weight by one without changing its height would make it out of balance, i.e., if $h(v) > \lceil c \log(|v| - 1)\rceil$.

LEMMA 2. *If u has minimal or less weight and u_H has minimal or larger weight*, *then* $|u_H| - |u_L| \geq 0$.

Proof. The conditions for the lemma give

$$h(u) > \lceil c \log(|u| - 1)\rceil \tag{11}$$

and

$$h(u_H) \leq \lceil c \log|u_H| \rceil. \tag{12}$$

Hence,

$$\lceil c \log(|u| - 1) \rceil < h(u) = h(u_H) + 1 \leq \lceil c \log|u_H| \rceil + 1. \tag{13}$$

Due to the strict inequality, we can remove the ceilings. Hence,

$$c \log(|u| - 1) < c \log|u_H| + 1. \tag{14}$$

This gives that

$$|u_H| > 2^{-1/c}(|u| - 1) > (|u| - 1)/2 \tag{15}$$

and hence

$$|u_H| - |u_L| = 2|u_H| - |u| > 2(|u| - 1)/2 - |u| > -1. \tag{16}$$

The lemma follows since $|u_H| - |u_L|$ is an integer.

LEMMA 3. *Let v be a node that is about to be rebuilt during an insertion. Furthermore, assume that $|v|$ is fixed. Then the smallest possible value of $\beta(v)$ occurs when each descendant $w \neq v$ along the longest path from v satisfies at least one of the following*:

- *w has minimal weight*;
- $|w_L| = 1$.

Proof. We prove the lemma by a contradiction. Assume that the smallest value of $\beta(v)$ occurs only when there exists some nodes on v's longest path which does not satisfy either of the two statements above. Let w be the highest such node and let u be the parent of w. That is, w is the same node as u_H. The node u will either be v itself, or it may have minimal weight or an empty subtree.

Thus, u has minimal or less weight, w has more than minimal weight, and $w_L > 1$.

Consider moving a node from w_L to u_L. Since $|w_L| > 1$, there is a node that can be moved. After the move, w will have minimal or larger weight, while u's weight will be the same as before. The overall condition, that a rebuilding is to be made at v, is not affected by the move.

Lemma 2 implies that $|u_H| - |u_L| \geq 0$ after the move. This, in turn, implies that $|u_H| - |u_L|$ was at least 2 before the move. Thus, the move caused $\delta(u)$ to decrease by at least 1, while $\delta(w)$ has changed by at most 1. Altogether, the move will not cause $\beta(u)$, and hence $\beta(v)$, to increase.

We can now continue moving nodes until w satisfies one of the two conditions in the lemma. This gives the contradiction. ∎

LEMMA 4. *For each node v with an empty subtree or minimal weight the following is true*:

$$\delta(v) > (2^{1-1/c} - 1)|v| - 3. \tag{17}$$

Proof. If v has an empty subtree then we are done since $\delta(v) = |v| - 1$. Otherwise, since v has its smallest possible weight we have

$$h(v) > \lceil c \log(|v| - 1) \rceil. \tag{18}$$

This and the fact that $\lceil c \log|v_H| \rceil \geq h(v_H)$ imply that

$$\lceil c \log|v_H| \rceil + 1 \geq h(v_H) + 1 = h(v) > \lceil c \log(|v| - 1) \rceil. \tag{19}$$

Thus,

$$\begin{aligned} c \log|v_H| + 1 &> c \log(|v| - 1), \\ 2^{1/c}|v_H| &> |v| - 1, \\ |v_H| &> 2^{-1/c}(|v| - 1) \\ &> 2^{-1/c}|v| - 1. \end{aligned} \tag{20}$$

From this it follows that

$$\begin{aligned} \delta(v) &= |v_H| - (|v| - |v_H|) - 1 \\ &= 2|v_H| - |v| - 1 \\ &> 2(2^{-1/c}|v| - 1) - |v| - 1 \\ &\geq (2^{1-1/c} - 1)|v| - 3. \end{aligned} \tag{21}$$

▮

Given the result of Lemmas 3 and 4 we can compute the value of $\sigma(v)$ when v is about to be rebuilt.

LEMMA 5. *Let v be a node where a rebuilding is to be made during an insertion. Then*

$$\sigma(v) > \frac{2 - 2^{1/c}}{2^{1/c} - 1} \cdot (|v| - 1) - 3\lceil c \log|v| \rceil. \tag{22}$$

Proof. Let v_d be v's dth descendant on the longest path from v. We have that

$$h(v_d) + d = h(v), \tag{23}$$

$$h(v) > \lceil c \log|v| \rceil, \tag{24}$$

and, for $d \geq 1$,

$$h(v_d) \leq \lceil c \log|v_d| \rceil. \tag{25}$$

Combining Eqs. (23), (24), and (25) gives

$$\begin{aligned} \lceil c \log|v_d| \rceil + d \geq h(v_d) + d = h(v) > \lceil c \log|v| \rceil, \\ 2^{d/c}|v_d| > |v|, \\ |v_d| > 2^{-d/c}|v|. \end{aligned} \tag{26}$$

Assume, hypothetically, that v has a minimal value of β. Combining Lemmas 3 and 4 with Eq. (26) gives that, for every $d \geq 1$,

$$\delta(v_d) > (2^{1-1/c} - 1)2^{-d/c}|v| - 3. \tag{27}$$

For $d = 0$, the same inequality holds as shown by Lemma 1. Adding the weight differences on v's longest path gives

$$\begin{aligned} \sigma(v) &\geq \beta(v) \\ &\geq \sum_{d=0}^{\lceil c \log|v| \rceil - 1} \delta(v_d) \\ &> (2^{1-1/c} - 1) \sum_{d=0}^{\lceil c \log|v| \rceil - 1} 2^{-d/c}|v| - \sum_{d=0}^{\lceil c \log|v| \rceil - 1} 3 \\ &= (2^{1-1/c} - 1) \cdot \frac{1 - 2^{-\lceil c \log|v| \rceil / c}}{1 - 2^{-1/c}}|v| - 3\lceil c \log|v| \rceil \\ &\geq \frac{2 - 2^{1/c}}{2^{1/c} - 1} \cdot \left(1 - \frac{1}{|v|}\right)|v| - 3\lceil c \log|v| \rceil \\ &= \frac{2 - 2^{1/c}}{2^{1/c} - 1} \cdot (|v| - 1) - 3\lceil c \log|v| \rceil. \end{aligned} \tag{28}$$

Note that the hypothesis on $\beta(v)$ is not required. ∎

Finally, we are able to give an upper bound on the amount of restructuring work needed to maintain a general balanced tree. In the analysis we assume that the cost of rebuilding a subtree v equals the number of elements in v, which is $|v| - 1$.

THEOREM 3. *Provided that the cost of rebuilding a subtree v is $|v| - 1$ and that $c > 1$ and $b > 0$, a binary search tree T of height at most $\lceil c \log|T| + b \rceil$ can be maintained at an amortized restructuring cost of $((2^{1/c} - 1)/(2 - 2^{1/c}))c \log|T| + o(\log|T|)$ per update.*

Proof. We use the algorithm from Theorem 2. First, we note that the rebuildings made to compensate for deletions (i.e., when $\mathrm{d}(T)$ becomes too large) only require a constant amortized cost per deletion; the constant depends on the relation between b and c. We ignore the cost for these rebuildings since it is included in the $o(\log n)$ term in the amortized update cost.

We associate the tree T with a potential function $\Phi(T)$. The function Φ is chosen such that each update (before any rebuilding) increases its value by an amount that is at most $((2^{1/c}-1)(2-2^{1/c}))c\log|T| + o(\log|T|)$. Each rebuilding decreases $\Phi(T)$ by an amount that covers the cost of the rebuilding. By showing that the potential is always positive or zero, we prove that the amortized cost of restructuring is at most $((2^{1/c}-1)/(2-2^{1/c}))c\log|T| + o(\log|T|)$.

Before we give the potential function, we would like to make an addition tot he definition of v_H and v_L from Section 2. Previously, we have only used v_H and v_L in association with nodes that are about to be involved in a partial rebuilding. In order to make our potential argument hold also for nodes that are not about to be involved in a rebuilding, we let the potential function look into the future. For each node v, we let v_H be that of v's subtrees which is going to be highest *the next time* v is involved in a rebuilding, regardless of which subtree is currently the highest. If v will never be involved in any future rebuilding, or if both subtrees will have the same height, we take v_H as the right subtree. This definition implies that we will not be able to compute the exact value of our potential function at a certain moment. However, at each update, we can still compute upper and lower bounds on the *change* in potential, which is enough for our purposes. In particular, this definition implies that a partial rebuilding at the node v will not affect the values of w_L and w_H for any node w outside the subtree v.

The potential $\Phi(T)$ is chosen as

$$\Phi(T) = \frac{2^{1/c}-1}{2-2^{1/c}}\sigma(T) + \frac{4\cdot(2^{2/c}-2^{1/c})}{(2-2^{1/c})^2}\tau(T), \tag{29}$$

where the function $\tau(v)$ is chosen as

$$\tau(v) = \begin{cases} 0, & \text{if } v \text{ is a leaf,} \\ \dfrac{\delta(v)\cdot\lceil c\log|v|\rceil}{|v|} + \tau(v_H) + \tau(v_L), & \text{otherwise.} \end{cases} \tag{30}$$

Note that, for a perfectly balanced tree, $\Phi(T) = 0$. A single update causes the value of $\tau(T)$ to be changed by at most $\lceil c\log|v|\rceil/|v|$ for each ancestor v of the inserted or deleted node. By definition of weight and height we have that $|v| \geq h(v) + 1$, which implies that

$$\frac{\lceil c \log|v| \rceil}{|v|} \leq \frac{\lceil c \log(h(v) + 1) \rceil}{h(v) + 1}. \tag{31}$$

Thus, the ancestor at height h causes an increase of $\tau(T)$ by at most $\lceil c \log(h + 1) \rceil/(h + 1)$. Therefore, an update causes the following change in $\tau(T)$:

$$\begin{aligned}\Delta\tau(v) &\leq \sum_{h=1}^{\lceil c \log|T| + b + 1 \rceil} \frac{\lceil c \log(h + 1) \rceil}{h} \\ &= O\left(\log^2 \lceil c \log|T| \rceil\right) \\ &= o(\log|T|). \end{aligned} \tag{32}$$

During an update the value of $\sigma(T)$ is changed by at most the number of ancestors of the inserted or deleted node. The number of ancestors is at most $\lceil c \log|T| + b \rceil$. Altogether we get an increase of Φ by

$$\begin{aligned}\Delta\Phi(T) &= \frac{2^{1/c} - 1}{2 - 2^{1/c}} \Delta\sigma(T) + \frac{4 \cdot (2^{2/c} - 2^{1/c})}{(2 - 2^{1/c})^2} \Delta\tau(T) \\ &\leq \frac{2^{1/c} - 1}{2 - 2^{1/c}} c \log|T| + o(\log|T|) \end{aligned} \tag{33}$$

per update.

It is left to show that the restructuring cost is covered by the potential function Φ. Since the cost of rebuilding trees of constant size can be included in the $o(\log n)$ term in the theorem, we may w.l.o.g. assume that

$$|v| > \frac{4 \cdot 2^{1/c}}{2 - 2^{1/c}}. \tag{34}$$

Using the value of $\delta(v)$ from Lemma 1 in the definition of $\tau(v)$, it follows that, when v is about to be rebuilt,

$$\begin{aligned}\tau(v) &\geq \frac{\delta(v) \cdot \lceil c \cdot \log|v| \rceil}{|v|} \\ &> \left(\frac{2 - 2^{1/c}}{2^{1/c}} |v| - 1 \right) \cdot \frac{\lceil c \cdot \log|v| \rceil}{|v|} \\ &= \frac{2 - 2^{1/c}}{2^{1/c}} \cdot \lceil c \cdot \log|v| \rceil - \frac{\lceil c \cdot \log|v| \rceil}{|v|}. \end{aligned} \tag{35}$$

After a rebuilding at the node v, $\Phi(v) = 0$. Furthermore, for each node w in T which is not part of the subtree v, the values of w_H and w_L remain unchanged after the rebuilding (due to the redefinition made above). Hence, Eq. (29) implies a decrease of T's potential such that

$$\Delta\Phi(T) = \Phi(v) = \frac{2^{1/c} - 1}{2 - 2^{1/c}}\sigma(v) + \frac{4 \cdot (2^{2/c} - 2^{1/c})}{(2 - 2^{1/c})^2}\tau(v). \quad (36)$$

Taking $\sigma(v)$ from Lemma 5 and $\tau(v)$ from Eq. (35) we get

$$\begin{aligned}
\Phi(v) > {} & \frac{2^{1/c} - 1}{2 - 2^{1/c}}\left(\frac{2 - 2^{1/c}}{2^{1/c} - 1} \cdot (|v| - 1) - 3\lceil c \log|v|\rceil\right) \\
& + \frac{4 \cdot (2^{2/c} - 2^{1/c})}{(2 - 2^{1/c})^2}\left(\frac{2 - 2^{1/c}}{2^{1/c}}\lceil c \log|v|\rceil - \frac{\lceil c \log|v|\rceil}{|v|}\right) \\
= {} & |v| - 1 - \frac{2^{1/c} - 1}{2 - 2^{1/c}} \cdot 3\lceil c \log|v|\rceil \\
& + \frac{2^{1/c} - 1}{2 - 2^{1/c}} \cdot 4\lceil c \log|v|\rceil - \frac{4 \cdot (2^{2/c} - 2^{1/c})}{(2 - 2^{1/c})^2} \cdot \frac{\lceil c \log|v|\rceil}{|v|} \\
= {} & |v| - 1 + \left(\frac{2^{1/c} - 1}{2 - 2^{1/c}} - \frac{4 \cdot (2^{2/c} - 2^{1/c})}{(2 - 2^{1/c})^2} \cdot \frac{1}{|v|}\right) \cdot \lceil c \log|v|\rceil. \quad (37)
\end{aligned}$$

Equation (34) implies that the last term is positive. Hence $\Delta\Phi(T) > |v| - 1$, which implies that the decrease in potential covers the cost of rebuilding at v. The proof follows from the fact that our potential function Φ is always nonnegative. ▮

5. A COMPARISON WITH WEIGHT-BALANCED TREES

As mentioned in the Introduction, the method of partial rebuilding is useful not only for ordinary binary search trees, but also for more complicated data structures, such as multidimensional search trees, where other balancing methods do not work. Previously, the only classes of balanced tree suitable for partial rebuilding have been the weight-balanced trees [19, 22, 23] and a modified version of α-balanced trees [3]. Since general balanced trees are also maintained by partial rebuilding, it is natural to compare the maintenance costs. Since α-balanced trees have the same maintenance cost as weight-balanced trees, we exclude them from our comparison.

In order to compare the structures we chose their balance criteria in such a way that the maximum heights of the trees are the same. Given a constant α, $0 < \alpha < 1/2$, each node v in a weight-balanced tree has to fulfill

$$\frac{|v\text{'s smallest subtree}|}{|v|} \geq \alpha. \tag{38}$$

The maximum height of the tree is

$$\left\lceil \frac{\log|T|}{\log(1/(1-\alpha))} \right\rceil. \tag{39}$$

A maximum height of $\lceil c \log|T| \rceil$ for weight-balanced trees is achieved by choosing $\alpha = 1 - 2^{-1/c}$. When a node v becomes out of balance, we have that [22]

$$\delta(v) \geq (1 - 2\alpha)|v| - 1 = (2^{1-1/c} - 1)|v| - 1. \tag{40}$$

Thus, if the cost of rebuilding a subtree v is $|v| - 1$, the amortized cost of an update is at most $1/(2^{1-1/c} - 1)$ for each ancestor of the inserted or deleted node. Since an update is made below at most $\lceil c \log|T| \rceil$ nodes, the amortized cost per update in the entire tree is at most $\lceil c \log|T| \rceil/(2^{1-1/c} - 1)$. (A more detailed study of weight-balanced trees maintained by partial rebuilding can be found in the literature [22]). It should be noted that this bound on maintaining weight-balanced trees is the same as the first bound given for GB-trees at the beginning of Section 4.

If we compare the cost of GB(c)-trees (Theorem 3) with the cost of weight-balanced trees we get

$$\begin{aligned}
&\frac{\text{cost of general balanced trees}}{\text{cost of weight-balanced trees}} \\
&\quad = \frac{(2^{1/c} - 1)/(2 - 2^{1/c})\lceil c \log|T| \rceil + o(\log|T|)}{\lceil c \log|T| \rceil/(2^{1-1/c} - 1)} \\
&\quad \approx \frac{2^{1/c} - 1}{2 - 2^{1/c}} \cdot (2^{1-1/c} - 1) \\
&\quad = 1 - 2^{-1/c} \\
&\quad < \frac{1}{2}.
\end{aligned} \tag{41}$$

Thus, the upper bound on the restructuring cost of maintaining balanced trees by partial rebuilding has been reduced by a factor at least 2.

6. APPLICATION TO MULTIDIMENSIONAL SEARCH TREES

In cases when rotations are costly, such as when maintaining k-d-trees [7], the method of partial rebuilding offers a powerful alternative. This has been shown for weight-balanced trees [19, 22, 23] and the same asymptotic bounds can be derived for general balanced trees. Also here we achieve the advantage of less stored information and a lower constant factor. A detailed study of these matters has been made by Mark Overmars [22]. For the sake of completeness, we just mention that if the cost of rebalancing a subtree v is $O(P(|v|))$, the amortized cost of an update will be $O((P(n)/n)\log n)$. For example, applied on k-d-trees, we get an amortized update cost of $O(\log^2 n)$.

7. CONCLUSIONS

Introducing general balanced trees, we have shown that there is no need for sophisticated balance criteria in order to maintain balance efficiently. The presented trees use a natural and attractive maintenance strategy; rebalancing is not made until it is really needed. Baer and Schwab [5] made an attempt to use a global balance criterion and make restructurings only when this criterion is violated. The amortized cost for their method is $\Theta(n)$ per operation. Therefore, they concluded that the best balancing methods are the ones that involve the strictest balance criteria. Here we have shown that, choosing carefully where to make rebuilding, a tree with a global balance criterion can be efficiently maintained. By comparing general balanced trees with weight-balanced trees, we have also shown that a restricted balance criterion is not necessarily the best.

Another advantage of the general balanced trees is that they can be easily maintained without any extra information stored in the nodes. As shown by Brown [8, 9], the explicitly stored balance information may in some classes of balanced trees be eliminated by coding the information through the location of empty pointers. However, the information is still stored, although implicitly. The *splay tree* presented by Sleator and Tarjan [24] does not require any balance information stored in the nodes. However, the height of a splay tree is not guaranteed to be $O(\log n)$. The logarithmic cost for searching in a splay tree is amortized, while we obtain a logarithmic worst-case cost.

In summary, the discovery of general balanced trees fills a gap in the well-studied area of binary search trees, showing that what may be the simplest balance criterion works surprisingly well. We believe that these trees offer a competitive alternative to other balanced tree structures, both from a theoretical and practical point of view.

Finally, we note an open problem: In this article we have given a better upper bound on the constant factor for GB-trees than what has been shown for weight-balanced trees. We conjecture that the constant factor is indeed better for GB-trees and we leave it as an open problem to prove this.

ACKNOWLEDGMENTS

Many thanks to Amir Ben-Amram for spotting two flaws in the proof of Theorem 3 and for a large number of improving comments during the revising process.

The comments by Kerstin Andersson, Svante Carlsson, Johan Håstad, Thomas Ottmann, and Ola Petersson have also improved the presentation of this work.

REFERENCES

1. G. M. Adelson-Velskii and E. M. Landis, An algorithm for the organization of information, *Dokl. Akad. Nauk SSSR* **146**(2) (1962), 1259–1262.
2. A. Andersson, Improving partial rebuilding by using simple balance criteria, *in* "Proc. Workshop on Algorithms and Data Structures," pp. 393–402, Springer-Verlag, Berlin/New York, 1989.
3. A. Andersson, Maintaining α-balanced trees by partial rebuilding, *Internat. J. Comput. Math.* **38** (1990), 37–48.
4. A. Andersson, Ch. Icking, R. Klein, and Th. Ottmann, Binary search trees of almost optimal height, *Acta Inform.* **28** (1990), 165–178.
5. J-L. Baer and B. Schwab, A comparison of tree-balancing algorithms, *Comm. ACM* **20**(5) (1977), 322–330.
6. R. Bayer, Symmetric binary B-trees: Data structure and maintenance algorithms, *Acta Inform.* **1**(4) (1972), 290–306.
7. J. L. Bentley, Multidimensional binary search trees used for associative searching, *Comm. ACM* **18**(9) (1975), 509–517.
8. M. R. Brown, A storage scheme for height-balanced trees, *Inform. Process. Lett.* **7**(5), (1978), 231–232.
9. M. R. Brown, Addendum to "A storage scheme for height-balanced trees," *Inform. Process. Lett.* **8**(3) (1979), 154–156.
10. H. Chang and S. S. Iynegar, Efficient algorithms to globally balance a binary search tree, *Comm. ACM* **27**(7) (1984), 695–702.
11. A. C. Day, Balancing a binary tree, *Comput. J.* **19**(4) (1976), 360–361.
12. C. C. Foster, A generalization of AVL-trees, *Comm. ACM*, **16**(8) (1973), 513–517.
13. I. Galperin and R. L. Rivest, Scapegoat trees, *in* "Proceedings of The Fourth Annual ACM–SIAM Symposium on Discrete Algorithms, 1993," p. 165–174.
14. L. J. Guibas and R. Sedgewick, A dichromatic framework for balanced trees, *in* "Proc. 19th Ann. IEEE Symp. on Foundations of Computer Science, 1978," p. 8–21.
15. D. E. Knuth, "The Art of Computer Programming," Vol. 3, "Sorting and Searching," Addison-Wesley, Reading, MA, 1973. (ISBN 0-201-03803-X.)

16. S. R. Kosaraju, Insertion and deletion in one-sided height-balanced trees, *Comm. ACM*, **21** (1978), 226–227.
17. W. A. Martin and D. N. Ness, Optimizing binary trees grown with a sorting algorithm, *Comm. ACM* **15**(2) (1972), 88–93.
18. H. A. Maurer, Th. Ottmann, and H. W. Six, Implementing dictionaries using binary trees of very small height, *Inform. Process. Lett.* **5**(1) (1976), 11–14.
19. K. Mehlhorn, ‘‘Data Structures and Algorithms 1: Sorting and Searching,’’ Springer-Verlag, Berlin/New York, 1984. (ISBN 3-540-13302-X.)
20. J. Nievergelt and E. M. Reingold, Binary trees of bounded balance, *SIAM J. Comput.* **2**(1) (1973), 33–43.
21. H. J. Olivie, ‘‘A Study of Balanced Binary Trees and Balanced One-Two-Trees,’’ Ph.D. thesis, Department of Mathematics, University of Antwerp, 1980.
22. M. H. Overmars, ‘‘The Design of Dynamic Data Structures,’’ Lecture Notes in Computer Science, Vol. 156, Springer-Verlag, Berlin/New York, 1983. (ISBN 3-540-12330-X.)
23. M. H. Overmars and J. van Leeuwen, Dynamic multi-dimensional data structures based on quad- and *k-d* trees, *Acta Inform.* **17** (1982), 267–285.
24. D. D. Sleator and R. E. Tarjan, Self-adjusting binary search trees, *J. Assoc. Comput. Mach.* **32**(3) (1985), 652–686.
25. Q. F. Stout and B. L. Warren, Tree rebalancing in optimal time and space, *Comm. ACM* **29**(9) (1986), 902–908.