Relaxed Balance using Standard Rotations*

Kim S. Larsen† Eljas Soisalon-Soininen‡ Peter Widmayer§

Abstract

In search trees with relaxed balance, rebalancing transformations need not be connected with updates, but may be delayed. For standard AVL tree rebalancing, we prove that even though the rebalancing operations are uncoupled from updates, their total number is bounded by $O(M \log(M + N))$, where M is the number of updates to an AVL tree of initial size N. Hence, relaxed balancing of AVL trees comes at no extra cost asymptotically. Furthermore, our scheme differs from most other relaxed balancing schemes in an important aspect: No rebalancing transformation can be done in the wrong direction, i.e., no performed rotation can make the tree less balanced. Moreover, each performed rotation indeed corresponds to a real imbalance situation in the tree. Finally, and perhaps most importantly, our structure is capable of forgetting registered imbalance if later updates happen to improve the situation. Our results are of theoretical interest and have possible sequential and parallel applications.

*A preliminary version of this paper appeared in the proceedings of the Fifth International Workshop on Algorithms and Data Structures, 1997.

†Corresponding author. Department of Mathematics and Computer Science, University of Southern Denmark, Main campus: Odense University, Campusvej 55, DK–5230 Odense M, Denmark. Phone: +45 6550 2328. Fax: +45 6593 2691. Email: kslarsen@imada.sdu.dk. The work of this author was supported in part by SNF (Denmark), in part by NSF (U.S.) grant CCR-9510244, and in part by the ESPRIT Long Term Research Programme of the EU under project number 20244 (ALCOM-IT).

‡Department of Computer Science and Engineering, Helsinki University of Technology, P.O.Box 540, FIN-02015 HUT, Finland. E-mail: ess@cs.hut.fi.

§Institut für Theoretische Informatik, ETH Zentrum, CH-8092 Zürich, Switzerland. E-mail: widmayer@inf.ethz.ch.

1 Introduction

When using search trees, rebalancing is normally carried out in connection with and immediately following the updates. As early as in [5], it was suggested that the two processes could be separated such that no rebalancing is required immediately following the update. However, only in recent years has this possibility been studied extensively. Structures with these features are now generally referred to as search trees with relaxed balance. In these structures, rebalancing is taken care of at some later time in small steps which may be interleaved (or carried out concurrently) with new updates.

A main motivation for this work is theoretical insight regarding these important data structures, the search trees. Clearly, if rebalancing is not carried out completely immediately after an update, some control over the data structures is lost. In particular, the structure could become unbalanced. It is therefore very interesting to determine whether or not it is still possible to rebalance efficiently; preferably as efficiently as in the standard search trees.

Apart from the theoretical interest, there could be important applications. For instance, if updates come in bursts from some external source at a speed which temporarily exceeds the volume which can be handled by a standard search tree, then rebalancing could be "turned off" until the burst is over, relying on the updating being close enough to a uniform distribution during this small time period that search times do not increase significantly. After the burst, but while updating still continues at a slower pace, the delayed as well as the new rebalancing could be carried out.

In a parallel environment on a shared-memory multi-processor, relaxed structures offer a possible solution to some concurrency control problems, by allowing a high degree of concurrency. This is important in applications where dictionaries must be accessed fast and with no down periods (where the tree could be rebuilt). Examples of such applications are flight reservation systems and cellular phone switching systems. There are other options than using a search tree for these kind of problems. Various hashing schemes exist, some of which can be used in a concurrent environment. However, search trees can support more operations efficiently, such as nearest neighbor searches and batch updates.

A detailed account for why rebalancing immediately after insertions or dele-

tions counteracts attempts to increase parallelism can be found in many papers on data structures for shared-memory architectures. In [3, 10], in particular, this is discussed in the context of relaxed balancing. The problem is that the path from the highest "unsafe" node (which might be the root) to the update must be locked, since otherwise a process might loose the path during the rebalancing phase on its way back up the tree.

The alternative option of using top-down rebalancing is not attractive either because of all the unnecessary structural changes that top-down methods incur. With bottom-up rebalancing, only the changes necessitated by the update are carried out. With top-down rebalancing on the other hand, one must, on the way down the search path, prepare for the worst-case scenario further down in the tree. In order to prevent problems from propagating back up the search path, the top-down process must ensure that configurations on the search path are not left in the extremes. For example, when making an insertion into an AVL tree, whenever the top-down process advances to the left (right) from a node with balance factor $+1$ (-1), a rotation must be performed. This implies that several costly rotations that block all concurrent operations must be performed, in contrast to the bottom-up strategy that requires at most one rotation.

Several proposals for solving the problem of relaxed balancing in search tree structures have been presented [3, 4, 6, 7, 8, 9, 10, 11]. However, a drawback in the previous solutions for the problem of relaxed balancing is that a single balancing transformation need not make the tree more balanced. It is natural to require that each structure changing operation is locally beneficial because in a concurrent environment urgent searches are present all the time. Moreover, it can be expected that the total number of necessary balancing transformations will be decreased, at least on the average, when none of them makes the tree less balanced.

Our solution has the distinctive property that it maintains the actual heights of the nodes and will thus base all decisions of whether or not to perform a rotation on the heights of the nodes in the tree. This property is important because then delayed rebalancing tasks will be forgotten when the ongoing insert and delete operations themselves make the tree balanced. Observe that all previous solutions to relaxed balancing work in such a way that balance conflicts from the history are remembered and gradually resolved. For example, a chromatic tree [4, 11] or a stratified tree [12] may be full of

conflicts that must be resolved even though the tree is perfectly balanced. This ability to forget problems of imbalance when they are taken care of automatically by later updates could be very important; in particular, if updates come from a uniform distribution. However, this property cannot be captured in a proof of worst-case complexity.

In the present paper, we propose a solution in which the balancing transformations are the standard ones used in a sequential solution. This means in particular that they are small compared to rebalancing operations in some of the previous proposals. Also this is important in a parallel environment, because fewer locks are necessary at any given time, and they would also be held less long, since less work has to be done in order to carry out the operation. The results of the present paper may also be considered to be of general interest as regards the theory of search tree structures. The question we address is the following: Is it possible to balance a binary search tree globally in such a way that only those portions of the tree which are indeed out of balance are modified? Moreover, changes in the tree should be allowed in the meantime, and the efficiency of balancing from the standard structures should be retained. We have obtained positive answers to these questions in the case of AVL trees.

In [13], a preliminary sketch of our relaxed balancing scheme was presented. In this present paper, that scheme is presented in full. Additionally, this paper contains a proof that each update gives rise to at most a logarithmic number of rebalancing operations.

2 Height-Valued Binary Search Trees

We consider binary search trees as implementing a totally ordered finite set S of *keys* chosen from a given domain. We allow the standard operations *search(k)*, *insert(k)*, and *delete(k)*, that is, search for key k, insert key k into set S, and delete key k from set S.

We assume that the trees are *leaf-oriented* binary search trees, which are full binary trees (each node has either two or no children) with the keys stored in the leaves. The internal nodes contain *routers*, which guide the search from the root to a leaf. The router stored in a node v must be greater than or

equal to any key stored in the leaves of v's left subtree and smaller than any key in the leaves of v's right subtree. The routers need not be keys stored in the leaves of the tree.

The height of a node u in a tree, denoted as $height(u)$, is defined as the length of a longest path from u to a leaf in the subtree rooted at u. The *height of a tree* is the height of its root. We will consider search trees where the balance condition is the AVL balance condition, that is, for each internal node the difference of the heights of its two subtrees is at most one.

With each node u of a binary search tree we associate an integer, called a *height value*, denoted $hv(u)$, which is either -1 or $height(u)$. If $hv(u) \neq -1$, that is, $hv(u) = height(u)$, then we require that for the child nodes v_1 and v_2 of u both (i) and (ii) hold:

(i) $hv(v_1) = height(v_1)$, $hv(v_2) = height(v_2)$, and

(ii) the difference of the heights of v_1 and v_2, is at most one, i.e., $|height(v_1) - height(v_2)| \leq 1$.

A binary search tree with the associated height values stored in the nodes is called a *height-valued tree*.

We say that node u exhibits a *balance conflict* if $hv(u) = -1$; otherwise, u is said to be *in balance*. A height-valued tree T is said to be *completely in balance*, if all its nodes are in balance. In this case, T is clearly an AVL tree.

The idea in the use of height-valued trees is that insertions and deletions cause non-negative height values in the search paths to be set to -1. These values of -1 are gradually changed to real heights by balancing transformations.

The insert and delete operations are defined below.

Insert(k): The tree is searched with key k. Whenever along this search path there is an internal node u with $hv(u) \neq -1$, $hv(u)$ is set to -1. If the key is found, the process terminates.

An unsuccessful search ends up in a leaf, say l. A new internal node u is created in place of l, and l and a new leaf l' containing the key k are made child nodes of u. The children are ordered such that the one containing the smaller key will be the left child of u. The router of u is a copy of the key contained in its left child.

The height value of l' is set to 0, and the height value of u is set to 1.

Delete(k): The tree is searched with key k. Whenever along this search path there is an internal node u such that $hv(u) \neq -1$, $hv(u)$ is set to -1. If the key is not found, the process terminates. Otherwise the leaf, denoted l, containing the key k is removed. Its parent is replaced by the sibling node of l.

The following observation is immediate.

Observation 1 When applied to a height-valued tree, the insert and delete operations preserve the height-valued property of the tree. □

Some of the definitions given above are motivated by the choice to prepare the structure for possible concurrent use. For instance, height value fields are set to -1 on the way to an update before it is known what the outcome of the update is. In that way there is no need to return later, which is difficult in a concurrent setting. Likewise, a leaf-oriented version is used to ensure that deletions can be carried out locally. In the standard setting, deleting a key in a binary node involves another node (containing the key's successor or predecessor) which may be more than a constant distance away.

3 Rebalancing a Height-Valued Tree

The task of rebalancing a height-valued tree is to remove all balance conflicts from the tree. Moreover, this should be possible by using small local transformations that allow—besides the concurrent searches—new insertions or deletions to occur.

Our strategy in resolving a balance conflict in a height-valued tree is to advance bottom-up so that a conflict in a node u will be resolved only if both subtrees of u contain no conflicts. This can be checked by looking at the child nodes of u only, because the subtrees of u are free from conflicts exactly when the height values of the child nodes of u are their true heights.

Let u be a node that exhibits a conflict, and let v_1 and v_2 be the children of u, such that both v_1 and v_2 have height values different from -1. Call u the *root of the operation before rebalancing.* We have two cases to consider.

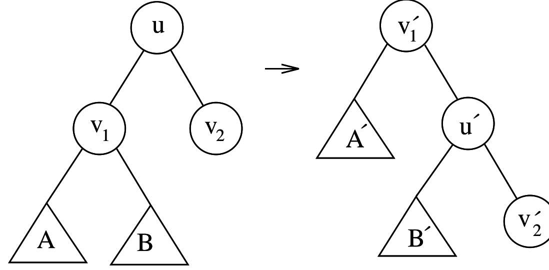

Figure 1: Single rotation.

Case 1: Finishing rebalancing; $|height(v_1) - height(v_2)| \leq 1$. In this case, the AVL balance condition is retained at u, and we simply set

$$hv(u) = \max\{hv(v_1), hv(v_2)\} + 1.$$

Case 2: Rotations; $|height(v_1) - height(v_2)| > 1$. In this case, a single or a double rotation is performed exactly in the same way as is done in the standard AVL tree balancing algorithm [1]. We assume here that v_1 is the left child and that $height(v_1) > height(v_2)$. The left subtree of node v_1 is denoted by A and the right subtree by B. There are two sub cases depending on the heights of A and B.

Case 2a: Single rotation; $height(A) \geq height(B)$. In this case, a single rotation to the right at u will be performed, see Fig. 1. After the rotation, the nodes u, v_1, v_2, and the subtrees A and B are denoted by u', v_1', v_2', A', and B', respectively. The height values of v_1' and u' are set to -1. Node v_1' is called the *root of the operation after rebalancing*.

Case 2b: Double rotation; $height(A) < height(B)$. The root of B is denoted by w and its subtrees by B_1 and B_2. In this case, a double rotation will be performed, see Fig. 2. The height values of w' and u' are set to -1. Node w' is called the *root of the operation after rebalancing*.

We prove the following invariant for height-valued trees.

Proposition 2 If a node has height value different from -1, then no conflicts can exist below this node. Additionally, there is always a conflicting node with both child nodes without conflicts, if the tree contains conflicts at all.

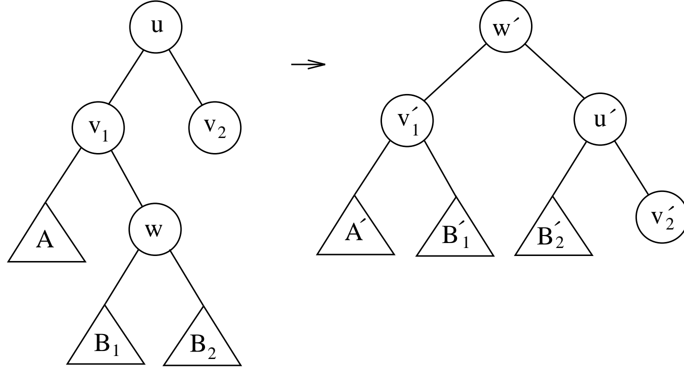

Figure 2: Double rotation.

Proof We prove the first claim first. The proof is by induction on the number of operations which are carried out.

For the base case, when no operations have been carried out on a given height-balanced tree, all height values are different from -1, so the claim follows trivially.

For the induction step, we assume that the invariant has held up until a given point and consider all possible next operations.

If the next operation is an update, any node u with $hv(u) \neq -1$ after the update must lie off the search path. Therefore, no nodes in its subtree are altered. So by the induction hypothesis, the property still holds.

Assume that the next operation is a rebalancing operation. Since $hv(v_1) \neq -1$ and $hv(v_2) \neq -1$, it follows from the induction hypothesis that every node w in the subtrees of v_1 and v_2 has $hv(w) \neq -1$. Thus, v_2' in Case 2a and v_1' and v_2' in Case 2b can safely be made different from -1. Furthermore, since $hv(u) = -1$ before the operation, we can set the height value of all nodes in any connected component which includes the top node after the operation to -1. This cannot violate the invariant.

The second claim of the proposition follows from the fact that height values of leaves are different from -1. □

Note that due to Proposition 2, a tree that is not completely in balance

must contain a node to which one of the two cases from the definition of the rebalancing applies. In the next section, we will analyze the progress towards a balanced tree that such a step is guaranteed to make.

4 The Efficiency of Relaxed Balancing

We assume that at some point we have a height-valued tree which is completely in balance. In this section, we show that rebalancing is logarithmic. More precisely, if M updates are carried out on the tree, then the number of rebalancing operations which can be applied before the tree is again in balance is $O(M \log(N + M))$, where N is the size of the tree when it was last in balance. This result is proven to hold no matter how updates and rebalancing operations are interleaved. The proof technique is amortized analysis [14]. The case where all updates are carried out before rebalancing is initiated was treated in [13].

For the purpose of the analysis, we divide the nodes up into three categories: *passive*, *active*, and *hyperactive*. Initially, all nodes are passive. After that, nodes that are traversed during an update, or that are involved in a rebalancing operation change status as described below.

Leaves are always categorized as passive. For an update, all nodes on the update path become active no matter what their status was before the update. A finishing rebalancing operation changes the status of the root of the operation to passive. Any other rebalancing operation leaves the root of the operation (after it is carried out) as active, and the node u' as hyperactive. Because rebalancing advances from bottom to top, we get the following:

Observation 3

- A passive node can only have passive children.
- The children of a hyperactive node are passive.

□

Proposition 4 On a path from the root to a leaf, there is at most one hyperactive node.

Proof Only rotations introduce hyperactive nodes. Consider a path from the root through a hyperactive node, newly created by a rotation. The only node on that path which could be hyperactive before the rotation was carried out is the top node of the operation before the rotation is carried out, since, by Observation 3, nodes higher up in the tree do not have passive children. □

The maximal height difference between two subtrees of a node u, both of which are balanced AVL trees, is denoted $mhd(n)$, where n is the number of nodes in the subtree rooted at u. We will abuse notation and frequently write $mhd(u)$ instead of $mhd(n)$. Since the smallest possible subtree is a leaf, $mhd(n)$ must equal the height of the highest AVL tree with $n-2$ nodes. It is well known that the minimum number of nodes in an AVL tree of height h is $F_{h+3}-1$ [2], where the Fibonacci numbers F_i are defined recursively by $F_1 = F_2 = 1$ and $F_i = F_{i-1}+F_{i-2}$, $i \geq 3$. From the inequality $F_{h+3}-1 \leq n-2$, it is easy to show that $h \leq \lfloor \log_\phi(\sqrt{5}n) \rfloor - 3 = mhd(n)$, where ϕ is the golden ratio $\frac{1+\sqrt{5}}{2}$, approximately equal to 1.618.

Proposition 5 A rebalancing operation *op* can increase the height difference for at most one node not involved in the operation. If this happens, the increase is one.

Proof Clearly, only nodes between the location for the rebalancing operation and the root of the tree can be affected. Let $u_1, u_2, \ldots, u_k$ be these nodes, where u_k is the root of the tree, and u_1 is the root of the operation before rebalancing. Note that the rebalancing operation cannot increase the height of the subtree with root u_1. This height can remain unchanged or decrease by one. This is true for both single and double rotations. Since height is defined recursively, whenever the height of some node u_i remains unchanged, no nodes above u_i will change height. So, assume that the height of u_1 decreases. Let t be the smallest index of a node on the path, the height of which remains unchanged. For any index $j < t$, since the height of u_j decreases, its subtree containing u_1 was the highest before the operation. Thus, the height difference at u_j decreases. This means that only at u_t can the height difference possibly increase; and at most by one. □

Definition 6 We refer to the rebalancing operation *op* and the node u_t defined in Proposition 5 and the proof of that proposition. We call *op* a height difference increasing operation targeted at u_t. □

Let us now define a potential function that measures the degree of imbalance in a tree, relative to the AVL tree requirements. Intuitively, in the course of events, a node u will become active at some point in time, and then stay active and stay the root of its subtree for a while. During that time, updates may pass through u down into u's subtree, and rotations targeted at u may be carried out in u's subtree. Both intuitively contribute to the imbalance at u. In addition, u starts with an initial imbalance that may be high if its state went from hyperactive to active, and that is zero or one if u was passive before becoming hyperactive. On the other hand, the imbalance at u intuitively cannot exceed the worst imbalance in an AVL tree, $mhd(n)$. The potential function takes these three effects into account, in the following way. For an active node u, let $upd(u)$ be the number of updates passing through u since u was last non-active (hyperactive or passive). Let $hio(u)$ be the number of height difference increasing operations targeted at u since u was last non-active. Let $lna(u)$ be the height difference between the two subtrees of u when u was last non-active.

Definition 7 The potential of a height-valued tree T is defined as the sum of the potentials of its nodes. The potential of a node u with children v and w is defined as follows, depending on the state of the node:

State	Potential
passive	0
active	$2[\min(upd(u) + hio(u) + lna(u), mhd(n)) - 1] + 1$
hyperactive	$2[\lvert height(v) - height(w)\rvert - 1] + 1$

where

$$[x] = \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

□

In the following, for a node u, we refer to $2[mhd(u) - 1] + 1$ as u's *maximal potential.*

More intuition concerning the potential function: The -1 reflects that AVL trees allow heights to differ with one. The $+1$ reflects that even if height difference is zero, if the node is still active (because rebalancing is currently taking place beneath it), we must be able to provide a decrease in the potential function when we return to perform a finishing rebalancing operation. The multiplication with 2 ensures that when a rebalancing operation decreases a height difference with one, two units are "released". One of these will give us the potential decrease, whereas the other will become the $+1$ on a node which becomes active due to the rebalancing operation.

Our strategy for obtaining the complexity result is to prove that an update increases the potential with at most $O(\log n)$, where n is the current size of the tree, and that a rebalancing operation decreases the potential with at least one.

Lemma 8 Fix any path from a leaf to the root, let $u_1, u_2, \ldots, u_k$ be those nodes on this path for which the potential of u_i, $i = 1, \ldots, k$, is non-zero and less than maximal. Let n_i be the size of the subtree rooted at u_i, and assume that the nodes are listed in the order they are encountered. Then $k \leq \lfloor \log_\phi(\sqrt{5} n_k) \rfloor - 3 + \log_\phi(2(n_k + 1))$.

Proof Consider the node u_k. Since its potential is not maximal, fewer than $mhd(n_k) - 1$ updates have gone through u_k since it was last passive or hyperactive. Since, by Observation 3, a node which is passive or hyperactive has balanced subtrees, u_k had a balanced subtree at the time it was last non-active, and the height of that subtree was at least $h = k - 1 - upd(u_k) \geq k - mhd(n_k)$. Such a tree contains at least $F_{h+3} - 1$ nodes. So, $n_k \geq F_{h+3} - 1 - upd(u_k) \geq F_{h+3} - 1 - mhd(n_k) - 1$. This gives an upper bound on k: Since $F_i \geq \phi^{i-2}$,

$$\phi^{k - mhd(n_k)} - 2 \leq F_{k-1-mhd(n_k)+3} - 2 \leq n_k + mhd(n_k).$$

So,

$$\phi^k \leq \phi^{mhd(n_k)}(n_k + mhd(n_k) + 2).$$

From the value of $mhd(n_k)$ and the (trivial) fact that $mhd(n_k) \leq n_k$,

$$k \leq \lfloor \log_\phi(\sqrt{5}n_k) \rfloor - 3 + \log_\phi(2(n_k+1)).$$

□

Lemma 9 For any path from a leaf to the root, let u_1, u_2, ..., u_k be those nodes on the path with maximum potential, listed in the order they are encountered. Let n_i denote the size of the subtree rooted at u_i. The potential increase for these k nodes due to an insertion below u_1 is bounded by $2(mhd(n_k+1) - mhd(n_1))$.

Proof First note that since u_i is in the subtree of u_{i+1}, $i \in \{1,\ldots,k-1\}$, we have that $n_i < n_{i+1}$, so $n_i + 1 \leq n_{i+1}$ and $mhd(n_i+1) \leq mhd(n_{i+1})$, since this function is nondecreasing.

Now, the potential increase, I, is

$$\begin{aligned} I &= \Sigma_{i=1}^k(2(mhd(n_i+1)-1)+1) - \Sigma_{i=1}^k(2(mhd(n_i)-1)+1) \\ &= 2\Sigma_{i=1}^k(mhd(n_i+1) - mhd(n_i)) \\ &\leq 2(mhd(n_k+1) - mhd(n_1)), \text{ by the observation above} \end{aligned}$$

□

Lemma 10 For any path from a leaf to the root, let u_1, u_2, ..., u_k be all those nodes on the path with zero potential, listed in the order they are encountered. Then $k \leq \lfloor \log_\phi(\sqrt{5}(n_k+1)+1) \rfloor - 2$.

Proof Since the nodes have zero potential, they are all passive. By Observation 3, u_k is the root of a subtree where all nodes are passive, i.e., the subtree is a standard AVL tree. Thus, $F_{k-1+3} - 1 \leq n_k$ from which the result follows. □

Lemma 11 An update increases the potential by at most $O(\log(N+M))$.

Proof By Proposition 4, there is at most one hyperactive node on an update path from the root to a leaf. This node becomes active instead. The potential

increase for this node is certainly bounded by $O(mhd(N))$. The rest of the nodes are either active or passive. Active nodes remain active, but since the number of nodes in their subtrees may change due to the update, the potential may increase. The active nodes may either have reached their maximal potential or not. By Lemma 8, the total potential increase for the active nodes with less than maximal potential is logarithmically bounded, and by Lemma 9, the total potential increase for the nodes of maximal potential is logarithmically bounded. Finally, passive nodes become active and their potential increases from 0 to a constant (at most 3). Due to Lemma 10, the total increase for these nodes is also logarithmically bounded. We conclude that an update gives rise to a potential increase of at most $O(\log(N + M))$. □

Lemma 12 A rebalancing operation decreases the potential by at least 1.

Proof There are three types of rebalancing operations: finishing, single rotation, and double rotation. We treat them separately.

Finishing: Only the top node of the operation changes status from active or hyperactive to passive. By definition, active and hyperactive nodes have potential at least 1 and passive nodes have potential 0.

Rotations: For both types of rotations, the root u of the operation *before* the operation is carried out has its status changed from active to hyperactive (if it is not already hyperactive). We analyze this first.

When u was last hyperactive or passive, it had a potential of $2[x - 1] + 1$, where x was the height difference of its subtrees. Since then, the height difference of u's subtrees can only increase if there is an update, and in that case clearly by at most 1, or if a height difference increasing operation targeted at u is carried out, and in that case, by Proposition 5, by at most one. So, $upd(u) + hio(u) + lna(u)$ is at least the current height difference of its subtrees. On the other hand, u's children are roots of balanced subtrees, so the height difference cannot exceed $mhd(n)$, where n is the size of the subtree rooted at u. In summary, changing the status of u from active to hyperactive cannot increase the potential. In the rest of this proof, we can therefore safely assume that u is hyperactive.

Single rotation: Referring to Fig. 1, the assumption is that $height(A) \geq$

$height(B)$. Since the subtree rooted at v_1 is balanced, $height(A) = height(B)$ or $height(A) = height(B) + 1$. We treat these two cases separately.

If $height(A) = height(B)+1$, then the height difference between the subtrees of u' is two less than the height difference between the subtrees of u. This gives a potential decrease of 4. The node v_1' becomes active, but the height difference between its subtrees is zero, so it only needs a potential of 1. Even if there is a potential increase of 2 due to this operation increasing the height difference between two subtrees elsewhere in the tree (by Proposition 5, this can happen to at most one node), the total potential change is still negative.

If $height(A) = height(B)$, then the height of the root of the operation before it is carried out is equal to the height of the root after the operation is carried out. Thus, the operation will not increase the height difference between the subtrees of any other nodes in the tree. The height difference at u' is one less than it was at u, decreasing the potential by 2, allowing the potential at v_1' to be set to 1, while still getting a total potential decrease.

Double rotation: Referring to Fig. 2, note that v_1 remains passive. The height difference at u' is two less than it was at u, and the height of the subtree decreases by one, so the argument is the same as for the "$height(A) = height(B) + 1$" case above. □

Theorem 13 Rebalancing is amortized logarithmic.

Proof We assume that we have an AVL tree T with N nodes. The potential of such a tree is zero. In Lemma 11 it is shown that an update increases the potential by at most $O(\log(N + M))$, and in Lemma 12 it is shown that a rebalancing operation decreases the potential by at least one. Since the potential cannot be negative, the result follows from these two parts. □

5 Concluding Remarks

We summarize the important aspects of the results in this paper.

The fact that a relaxed scheme with a complexity comparable to the standard case can be designed was already known. From a theoretical point of view, the news is that it can be done using only standard single and double rotations,

and it can be done in such a way that every rebalancing operation which is carried out is applied to a genuine problem of imbalance, and thus, makes the tree more balanced. In the other proposals for relaxed balance, trees can be full of conflicts which rebalancing operations must work on, despite of actually being in balance. We explain this in greater detail for the proposals for relaxed red-black trees (chromatic trees) [3, 4, 10]. In those proposals, conflicts are sequences of red nodes in a row, or overweighted nodes, i.e., nodes that are not red or black, but double black, triple black, etc. It is possible to have conflicts in a tree despite of the fact that the nodes of the tree could be colored red and black in such a way that there would be no conflicts at all. Thus, in such a proposal, conflicts do not necessarily correspond to a real problem of imbalance, and conflicts are not forgotten when later updates automatically balances the tree.

With regards to an implementation on a shared-memory system, we have the following additional comments.

The rebalancing operations are small. Our scheme uses standard rotations as known from the sequential case instead of the larger operations proposed in some papers. This means that the number of exclusive locks that has to be held at the same time is smaller. Since fewer locks must be obtained and fewer pointers moved, the locks that we have to hold are also released earlier than in other proposals. This is important because other processors are prevented from entering the whole subtree of a node which is exclusively locked.

In comparison with several other schemes for relaxed balancing, ours is more restricted since operations can only be applied to bottom-most problems. However, the issues of registering problems of imbalance and deciding when and where to rebalance must still be considered in a concrete application. Solutions will vary depending on the concrete balance schemes and on whether sequential or concurrent implementations are considered. We leave it as an open problem to find the optimal algorithms for the different scenarios.

Finally, in a concrete application, it is important that the number of rebalancing operations which has to be carried out in response to an update is fairly small. We have proven that it is logarithmic, but the constant in front of the logarithmic term is also quite small. From the potential function, it is clear that the constant is at most two, possibly smaller.

References

[1] G. M. Adel'son-Vels'kiĭ and E. M. Landis, An algorithm for the organisation of information, *Soviet Math. Dokl.* **3** (1962) 1259–1262.

[2] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *Data structures and algorithms*, Addison-Wesley, 1983.

[3] J. Boyar, R. Fagerberg, and K. S. Larsen, Amortization results for chromatic search trees, with an application to priority queues, *Journal of Computer and System Sciences* **55** (1997) 504–521.

[4] J. Boyar and K. S. Larsen, Efficient rebalancing of chromatic search trees. *Journal of Computer and System Sciences* **49** (1994) 667–682.

[5] L. J. Guibas and R. Sedgewick, A dichromatic framework for balanced trees. Proceedings of the 19th Annual IEEE Symposium on the Foundations of Computer Science (1978) 8–21.

[6] S. Hanke, T. Ottmann, and E. Soisalon-Soininen, Relaxed balanced red-black trees, Third Italian Conference on Algorithms and Complexity, *Lecture Notes in Computer Science* **1203** (1997) 193–204.

[7] J. L. W. Kessels, On-the-fly optimization of data structures, *Comm. ACM* **26** (1983) 895–901.

[8] K. S. Larsen, AVL trees with relaxed balance, *Proc. 8th International Parallel Processing Symposium*, IEEE Computer Society Press, 1994, pp. 888–893.

[9] K. S. Larsen and R. Fagerberg, Efficient rebalancing of B-trees with relaxed balance, *International Journal of Foundations of Computer Science* **7** (1996) 196–202.

[10] O. Nurmi and E. Soisalon-Soininen, Uncoupling updating and rebalancing in chromatic binary search trees, *Proc. 10th ACM Symposium on Principles of Database Systems*, 1991, pp. 192–198.

[11] O. Nurmi, E. Soisalon-Soininen and D. Wood, Concurrency control in database structures with relaxed balance, *Proc. 6th ACM Symposium on Principles of Database Systems*, 1987, pp. 170–176.

[12] T. Ottmann and E. Soisalon-Soininen, Relaxed balancing made simple, Technical Report 71, Institut für Informatik, Universität Freiburg, Germany, 1995.

[13] E. Soisalon-Soininen and P. Widmayer, Relaxed balancing in search trees, *Advances in Algorithms, Languages, and Complexity* (D.-Z. Du and K.-I. Ko, eds.), Kluwer Academic Publishers, 1997, pp. 267–283.

[14] R. E. Tarjan, Amortized computational complexity. *SIAM J. Alg. Disc. Meth.* **6** (1985) 306–318.