AVL Trees with Relaxed Balance*

Kim S. Larsen†
University of Southern Denmark, Odense‡

Abstract

The idea of relaxed balance is to uncouple the rebalancing in search trees from the updating in order to speed up request processing in main-memory databases. In this paper, we describe a relaxed version of AVL trees. We prove that each update gives rise to at most a logarithmic number of rebalancing operations, and that the number of rebalancing operations in the semi-dynamic case is amortized constant.

1 Introduction

Relaxed balance is the term used for search trees where the rebalancing has been uncoupled from the updating. Normally, in balanced search trees, rebalancing is tightly coupled to the updating: as soon as an update has been performed, rebalancing operations are applied until the given balance constraints are again fulfilled. In search trees with relaxed balance, updating and rebalancing are processes which can be controlled separately. Typically, this means that balance constraints must be relaxed such that an update can leave the tree in a well-defined state. Thus, the assumptions under which rebalancing is carried out are changed. This poses the problem of still carrying out the rebalancing efficiently.

Relaxed balance deals with a fundamental question concerning one of the most important classes of data structures in computer science, the balanced

* To appear in Journal of Computer and System Sciences. A preliminary version of this paper appeared in the proceedings of the 8th International Parallel Processing Symposium (IPPS'94), pages 888-893, IEEE Computer Society Press, 1994.

† kslarsen@imada.sdu.dk, http://www.imada.sdu.dk/~kslarsen/

‡ Department of Mathematics and Computer Science, University of Southern Denmark, Main Campus: Odense University, Campusvej 55, DK-5230 Odense M, Denmark.

search trees. It is therefore important to obtain as full an understanding of the issue as possible. Additionally, there are practical applications for this line of work.

In standard search trees, the rebalancing part of an update is the more time-consuming part of the whole update. If search and update requests come in bursts (possibly from several external sources), the search tree may occasionally be unable to process the requests as fast as desirable. In search trees with relaxed balance, rebalancing can be "turned off" for a short period of time in order to speed up the request processing. When the burst is over, the tree can be rebalanced again, while searching and updating still continue at a slower pace. Of course, if rebalancing is postponed for too long, the tree can become completely unbalanced.

A different motivation comes from using search trees on shared-memory architectures. If rebalancing is carried out in connection with updates, either top-down or bottom-up, this creates a congestion problem at the root in particular, and all the locking involved seriously limits the amount of parallelism possible in the system. In search trees with relaxed balance, rebalancing operations can be carried out such that they, and their associated locks, are very localized in time as well as in space. In particular, exclusive locking of whole paths or step-wise exclusive locking down paths can be avoided. This means that the amount of parallelism possible is not limited by the height of the tree. The question as to what extent these properties can be exploited by present or future architectures is still only partly answered [9].

We give a brief account of previous work in relaxed balancing. The idea of uncoupling the rebalancing from the updating was first mentioned in [8], and the first partial result, dealing with insertions only, is from [12]. The first relaxed version of AVL trees [1] was defined in [25]. A different AVL based version was treated in [19, 28]. There, rebalancing is only performed at locations where all subtrees are known to be in balance. This is different from rebalancing just because some balance information exceeds some bounds. One update could have caused the balance information in some node to exceed the bound, reflecting some imbalance at that time, but another later update could have balanced the tree. However, the fact that the tree is balanced may not immediately be reflected in the balance information, because information from the last update may take some time to progress to the problem node, and this may cause unnecessary structural changes to the tree. In order to avoid this by only working on problem nodes with balanced subtrees, it must be registered which trees are in balance and which are not. In [19, 28], this was obtained by having updaters mark their search paths.

The first relaxed version of B-trees [3] is also from [25] with proofs of complexity matching the sequential ones in [16, 17]. Everything in [16, 17] is really done for the more general (a, b)-trees [11], so results are implied also for 2-3 trees [2], as well as for (a, b)-trees with other choices of a and b. A relaxed version of red-black trees [8] was presented in [23, 24] and complexities matching the sequential case were established in [6, 7] and [4, 5] for variants of the original proposal. In [15], all these results were matched while only using single and double rotations. Some of the earlier results are surveyed in [28]. In [18, 27], it is shown how a large class of standard search trees can automatically be equipped with relaxed balance, and a version of red-black trees based on these general ideas is described in [10]. Finally, a description and an analysis of group updates in relaxed trees can be found in [21], and experimental results which indicate that relaxed trees improve the performance in multi-processor environments are reported in [9].

As mentioned above, the first proposal for relaxed AVL trees is from [25]. A collection of rebalancing operations are defined, but there is no proof of complexity. In this paper, we introduce a modified collection of rebalancing operations which allows for a higher degree of parallelism in the rebalancing. For this set of operations, we can prove that each update gives rise to at most a constant number of rebalancing operations, and that rebalancing after insertions is amortized constant. These results match the complexities from the sequential case. A preliminary account of some of these results can be found in [14]. With these results we have shown that relaxed AVL trees can be defined such that they have all the properties one would hope for based on the knowledge from the sequential case.

In greater detail, suppose that the original tree T has $|T|$ nodes before k insertions and m deletions are performed. Then $N = |T| + 2k$ is the best bound one can give on the maximum number of nodes the tree ever has, since each insertion creates two new nodes (see below). We show that the tree will become rebalanced after at most $k\lfloor \log_\phi(N+2) + \log_\phi(\frac{\sqrt{5}}{2}) - 2 \rfloor + m(\lfloor \log_\phi(N+2) + \log_\phi(\frac{\sqrt{5}}{2}) - 3 \rfloor)$ rebalancing operations, where ϕ is the golden ratio. This is $O(\log(N))$ per update. Additionally, we show that starting with an empty tree, rebalancing after an insertion is amortized constant. Recall that even though the standard red-black trees and AVL trees are almost equally efficient in practice, the amortized constant rebalancing result for red-black trees [11, 20] does not hold in full generality for AVL trees [22], but only for the semi-dynamic case.

2 AVL Trees with Relaxed Balance

In this section, we define standard AVL trees as well as AVL trees with relaxed balance. The trees considered are leaf-oriented binary search trees, so the keys are stored in the leaves and the internal nodes only contain routers which guide the search through the tree. The router stored in a node is larger than or equal to any key in the left subtree and smaller than any key in the right subtree. The routers are not necessarily keys which are present in the tree, since we do not want to update routers when a deletion occurs. The tree is a full binary tree, so each node has either zero or two children.

If u is an internal node, then we let u_l and u_r denote the left and right child of u, respectively. The *height* of a node u is defined by

$$h(u) = \begin{cases} 0, & \text{if } u \text{ is a leaf} \\ \max(h(u_l), h(u_r)) + 1, & \text{otherwise} \end{cases}$$

The *balance factor* of an internal node u is defined by $b_u = h(u_l) - h(u_r)$. An AVL tree is a binary search tree, where the heights of the children of any internal node u differ with at most one, i.e., $b_u \in \{-1, 0, 1\}$.

We want to use AVL trees, but at the same time we want to uncouple updating from rebalancing. The question arises as to how we allow the tree to become somewhat unbalanced without loosing control completely. A technique introduced in [12] can be used in a modified form, as it is done in [25].

Every node u has an associated *tag value*, denoted t_u, which is an integer greater than or equal to -1, except that the tag value of a leaf must be greater than or equal to zero. We can now define the *relaxed height* of a node u as follows.

$$rh(u) = \begin{cases} t_u, & \text{if } u \text{ is a leaf} \\ \max(rh(u_l), rh(u_r)) + 1 + t_u, & \text{otherwise} \end{cases}$$

In analogy with the balance factor, the *relaxed balance factor* is defined as $b_u = rh(u_l) - rh(u_r)$. The non-relaxed balance factor will not be used again, so we reuse the symbol, and let b_u denote the relaxed balance factor from now on. A search tree is now an AVL tree with relaxed balance if for every internal node u, $b_u \in \{-1, 0, 1\}$. Clearly, if all tags have the value zero, then the tree is an AVL tree.

We now describe the operations which can be performed on AVL trees with relaxed balance. The operations are given in the appendix. By symmetry, it is sufficient in the proofs to show that we can handle problems in a left child. All the operations for this, are listed in Appendix A. In an actual implementation, there must be operations available for the symmetric cases as well. The operations which should be added in order to complete the set are listed in Appendix B. These operations will not be discussed in the proofs.

For notational convenience, we use $l(v)$ and $r(v)$, where v is an internal node, letting $l(v) = 1$ if the relaxed height of the left child of v is larger than the relaxed height of the right child, and $l(v) = 0$ otherwise. We use $r(v)$ similarly. In other words,

$$l(v) = \begin{cases} 1, & \text{if } b_v = 1 \\ 0, & \text{otherwise} \end{cases} \qquad r(v) = \begin{cases} 1, & \text{if } b_v = -1 \\ 0, & \text{otherwise} \end{cases}$$

Searching can be carried out exactly as for standard leaf-oriented AVL trees. The same is true for insertion and deletion (see the appendix, operations (i) and (d)), except that tag values are adjusted.

The purpose of the rebalancing operations is to modify an AVL tree with relaxed balance until it is a standard AVL tree and, thus, guaranteed to be balanced. Obviously, this is done by removing non-zero tag values. The difficulty is to do this without violating the relaxed balance constraint that $b_u \in \{-1, 0, 1\}$ for all internal nodes u.

Assume that we have a balance problem, i.e., a node with non-zero tag value. Now rebalancing can be performed provided that the tag value of the parent node is at least zero (see the appendix, operations (n) and (p)). Notice that in [25, 26], t_u must be zero, whereas we only require that t_u be different from -1, thus allowing for a higher degree of parallelism. If one of the two children of u has tag value -1 and the other has tag value greater than zero, then we require that the negative tag value be taken care of first.

Applying operation (n) or (p) has the effect of either removing a problem (changing a -1 to 0, or decreasing a positive tag value), or moving it closer to the root. At the root, problems disappear as the tag value of the root can always be set to zero without violating the relaxed balance constraints. However, operations (n) and (p) might violate the constraints since the relaxed balance factor of the top node can change to 2 or -2. So, as an indivisible part of the rebalancing step, a check for this problem is made, and the appropriate action is taken. In case of a violation after having applied

operation (n), one of the operations (n1) through (n4) are performed. If operation (p) is the cause of a violation, then we consider the node w, the sibling of v. Because of the requirement that negative values be taken care of first, we can assume that $t_w \geq 0$. If $t_w > 0$, then operation (p0) is applied. If $t_w = 0$, then one of the operations (p1) through (p4) are applied. Only after this has been taken care of, and the tree is indeed an AVL tree with relaxed balance, has this rebalancing step been completed.

It is an easy exercise to check that the rebalancing operations will not violate the relaxed balance constraints. We merely state the result here. It can be verified by simple case analysis proofs for one operation at a time.

Lemma 1 Assume that one of the operations (n) or (p) are performed on an AVL tree with relaxed balance. In case the relaxed balance factor of u (see the appendix) changes to 2 or -2, if the relevant one of the operations (n1) through (n4) and (p0) through (p4) is applied, then the tree is still an AVL tree with relaxed balance.

Additionally, insertion and deletion (operations (i) and (d)) cannot be the cause of a violation of the relaxed balance constraints. □

One particular implication of this lemma which will be used later is the following.

Corollary 1 Every operation leaves the relaxed height of the top node involved in the operation unchanged. □

Finally, notice that if the tree has tag values which are non-zero, then one of the operations (n) and (p) can be applied. To see this, consider the path from the root down to a node with non-zero tag value. On this path, let v be the first node with non-zero tag value. The tag value of the root is always zero, so v is not the root. Thus, one of the operations (n) and (p) can be applied to v and its parent.

In the next section, we bound how many times operations (n) and (p) can be applied. So, if updating stops at some point, then the tree will eventually become a standard AVL tree. As the next section will show, this happens fast.

3 Logarithmic Rebalancing

For the complexity analysis, we assume that initially the search tree is an AVL tree, and then a series of search, insert, and delete operations occur. These operations may be interspersed with rebalancing operations. The rebalancing operations may also occur after all of the search and update operations have been completed; our results are independent of the order in which the operations occur. In any case, the search tree is always an AVL tree with relaxed balance, and after enough rebalancing operations, it will again be an AVL tree. We only need to bound the number of times operations (n) and (p) are applied, since the operations (n1) through (n4) and (p0) through (p4) are only used immediately after operations (n) or (p), if they give rise to a violation of the relaxed balance constraints. So, the operations (n1) through (n4) and (p0) through (p4) are considered to be a part of the operation which gave rise to their application.

If some of the operations are done in parallel, they must involve edges and nodes which are completely disjoint from each other. The effect will be exactly the same as if they were done sequentially, in any order. Thus throughout the proofs, we assume that the operations are done sequentially. At time 0, there is an AVL tree, at time 1 the first operation has just occurred, at time 2 the second operation has just occurred, etc.

It is well known that the minimum size of AVL trees is closely related to the Fibonacci sequence. In the following, we use the definition of the Fibonacci sequence from [13]. Let $F_0 = 0$, $F_1 = 1$, and for $n \geq 0$, let $F_{n+2} = F_{n+1} + F_n$. The Fibonacci sequence is closely related to the golden ratio.

Proposition 1 Let $\phi = \frac{1+\sqrt{5}}{2}$ and $\hat{\phi} = 1 - \phi$. Then $F_n = \frac{1}{\sqrt{5}}(\phi^n - (\hat{\phi})^n)$. Furthermore, $\frac{\phi^n}{\sqrt{5}}$ rounded to the nearest integer equals F_n.

Proof See [13]. □

Clearly, if a node u in an AVL tree has a large height, then the subtree in which u is the root will also be large. In an AVL tree with relaxed balance, however, a node could have a large relaxed height because many nodes below it have been deleted and have caused tag values to increase. In this case, u may not have a large subtree remaining. In order to establish that large tag values can only be caused by large subtrees, we have to somehow count those nodes below u which have been deleted. Nodes are inserted and deleted and

we want to associate every node which has ever existed with nodes currently in the tree.

Definition 1 Any node in the tree at time t is associated with itself. When a node is deleted, the two nodes which disappear, and all of their associated nodes, will be associated with the node which remains, i.e., the sibling of the node to be deleted (this is node v in the appendix, operation (d)). □

Thus, every node that was ever in the tree is associated with exactly one node which is currently in the tree.

Definition 2 Define an A-subtree (associated subtree) of a node u at a particular time t to be the set of all the nodes associated with any of the nodes in the subtree with root u at time t. □

We can now prove a strong connection between relaxed heights and A-subtrees.

Lemma 2 At time t, if u is a node in the tree, then there are at least $2F_{rh(u)+2} - 1$ nodes in the A-subtree of u.

Proof By induction in t.

The base case is when $t = 0$. At that point, the tree is a standard AVL tree and $rh(u) = h(u)$. A simple induction argument shows that the result holds here. The smallest leaf-oriented trees of heights 0 and 1 have 1 and 3 nodes, respectively. The minimum number of nodes in a leaf-oriented AVL tree of height $h \geq 2$ is 1 for the root plus the minimum number of nodes in a leaf-oriented AVL tree of height $h - 1$ and one of height $h - 2$. By induction, this value is $1 + (2F_{(h-1)+2} - 1) + (2F_{(h-2)+2} - 1) = 2F_{h+2} - 1$. This concludes the base case of the outermost induction argument.

For the induction part, assume that the lemma holds up until time t, where $t \geq 0$. By checking all the operations, we prove that the lemma also holds at time $t + 1$.

When an insertion takes place, two new nodes are added. These are given tag values zero, which results in a relaxed height which is also zero. Since $2F_2 - 1 = 1$, the result holds here. The relaxed height of the top node is maintained while the size of its A-subtree is increased, so the result holds for that node too.

When a deletion occurs, two nodes are deleted. However, these two nodes, and all their associated nodes, are now associated with the parent node at

the time of the deletion. So, the size of the A-subtree of u is unchanged as, by Corollary 1, is $rh(u)$.

For the remaining operations, we first make some general observations. Notice first that the number of nodes is always preserved. So, by Corollary 1, the result always holds for the top node. Additionally, if the tag value of a node is decreased while its subtree remains unchanged (or has its size increased), then the lemma cannot fail for that node either.

We have dealt with the top nodes and the following argument will take care of the major part of the remaining nodes. It turns out that a node which is given the tag value zero and which gets two subtrees that were not changed in the operation cannot make the lemma fail. To see this, assume that v is the node in question and that it has children v_l and v_r. Assume without loss of generality that $rh(v_l) \geq rh(v_r)$. Then $rh(v_l) = rh(v) - 1$. Since we have already dealt with the top nodes, we can assume that v is not such a node. Thus, we know that the relaxed balance factor of v belongs to $\{-1, 0, 1\}$, and so we can assume that $rh(v_r) \geq rh(v) - 2$. The number of nodes in the A-subtree of v is now at least $(2F_{rh(v_l)+2} - 1) + (2F_{rh(v_r)+2} - 1) + 1 \geq 2F_{rh(v)+1} + 2F_{rh(v)} - 1 = 2F_{rh(v)+2} - 1$.

The v node in operation (n2) and the w node in operation (p2) are the only nodes which have not been covered by one of the cases in the above. However, their relaxed heights are decreased while their A-subtrees remain unchanged. □

Corollary 2 The relaxed height of any node at any time is at most

$$\lfloor \log_\phi(N + 2) + \log_\phi(\tfrac{\sqrt{5}}{2}) - 2 \rfloor.$$

Proof Notice first that since tag values cannot be smaller than -1, relaxed heights cannot decrease when moving towards the root. Thus, no relaxed height can be larger than the relaxed height of the root.

At any time, the number of nodes in the A-subtree of the root r is bounded by N. Since, by Lemma 2, there are at least $2F_{rh(r)+2} - 1$ nodes in the A-subtree of r, the inequality $2F_{rh(r)+2} - 1 \leq N$ must hold.

By Proposition 1, $F_n \geq \frac{\phi^n}{\sqrt{5}} - \frac{1}{2}$, so $\phi^{rh(r)+2} \leq \frac{\sqrt{5}}{2}(N+2)$, which implies that $rh(r) \leq \log_\phi(\frac{\sqrt{5}}{2}(N + 2)) - 2$. The result follows from the fact that $rh(r)$ must be an integer. □

The values of ϕ and $\log_\phi(\frac{\sqrt{5}}{2})$ are approximately 1.618 and 0.232, respectively.

In order to bound the number of operations which can occur, it is useful to consider a positive tag value t_u as consisting of t_u positive units. Likewise, a negative tag value will be referred to as a negative unit. We can now, for the purpose of the complexity proof, assume that units have identities such that they can be followed around in tree from the moment they are created until they disappear. If, as the effect of an operation, a negative or positive unit is deleted from some node only to be introduced at another, we say that the unit has been moved.

Proposition 2 An insertion can create at most one extra negative unit and a deletion can create at most one extra positive unit. No other operation creates positive or negative units.

Proof The claim for insertion is trivially fulfilled. For deletion, observe that in order to make the tag value negative, t_u and t_v must both be -1 and $r(u)$ must be 0, in which case, we actually get rid of at least one negative unit. The question is whether a deletion can create more than one positive unit. Clearly, this will happen if and only if $t_u \geq 0$, $t_v \geq 0$, $r(u) = 1$, and $t_w = 0$ (if $t_w > 0$, then we would loose at least one positive unit when deleting w). Now, $t_w = 0$ implies that $rh(w) = 0$ (since w is a leaf). So, since $r(u) = 1$ means that $b_u = -1$, we get that $rh(v) = b_u + rh(w) = -1$. However, this is impossible since the relaxed height of a leaf is at least zero and the relaxed height of an internal node is at least the relaxed height of its children.

In the operations (n), (p), (n2), (n4), (p0), (p2), and (p4), negative or positive units either disappear or are moved to other nodes.

It could appear that operations (n1) and (n3) might introduce a new positive unit at the top node. However, these operations are only applied immediately after operation (n) and only in the case where b_u becomes 2. In that case, $r(u) = 0$, so a negative unit is moved to the top node u. Adding one to the tag value of u in an immediately succeeding operation will simply make that negative unit disappear.

Likewise, operations (p1) and (p3) might increase the tag value of the top node. However, these operations are only applied immediately after operation (p) in case b_u becomes -2. In that case, $l(u) = 0$, so a positive unit disappeared at first, but then if one of the operations (p1) or (p3) are applied, it may be added again, and we simply consider it to have moved. □

We now prove that whenever a positive or negative unit is moved to another node, then the relaxed height of the new node will be larger than the relaxed height of the old node. Since there is a limit to how large relaxed heights can become, this is a crucial step towards limiting the number of times operations can be applied.

Lemma 3 When a negative unit is involved in operation (n), it either disappears or it is moved to a node with a larger relaxed height. Additionally, no operation will decrease the relaxed height of a node with tag value -1.

Proof Consider operation (n) and assume that the negative unit does not disappear. Since $t_u \geq 0$, clearly $rh(u)$ before the operation is larger than $rh(v)$. By Corollary 1, the negative unit is moved to a node with larger relaxed height. It might be necessary to apply one of the operations (n1) through (n4) afterwards, but again by Corollary 1, they preserve the relaxed height of the top node.

For the remaining operations, an insertion does not involve an existing negative unit, and a deletion removes negative units unless $t_u = t_v = -1$ and $r(u) = 0$, in which case one disappears and the other is associated with the node v with relaxed height $rh(u)$.

The remaining operations may involve a negative unit associated with the top node, but this unit will also be associated with the top node after the operation, which, by Corollary 1, is safe. Finally, operations (n4) and (p4) involve a negative unit t_w and t_x, respectively, but this unit disappears. □

Lemma 4 When a positive unit is involved in operation (p), it either disappears or it is moved to a node with larger relaxed height. Additionally, if some other operation decreases the relaxed height of a node by decreasing the tag value, then it decreases the tag value by at least the same amount.

Proof Consider operation (p) and assume that the positive unit does not disappear. As $t_u \geq 0$, clearly $rh(u)$ before the operation is larger than $rh(v)$. By Corollary 1, the positive unit is moved to a node with larger relaxed height. It might be necessary to apply one of the operations (p0) through (p4) afterwards, but again by Corollary 1, they preserve the relaxed height of interest here.

For the remaining operations, an insertion may remove a positive unit, while a deletion might move the positive units t_v to a node with larger relaxed height.

The remaining operations may involve a positive unit associated with the top node, but this unit will also be associated with the top node after the operation, which, by Corollary 1, is safe. Finally, the operations (n2), (p0), and (p2) move a positive unit to a node with larger relaxed height.

Operation (p0) is the only operation which decreases the relaxed height of a node with positive tag value. However, the tag value of that node is decreased by the same amount. □

At this point, we have all the partial results that will enable us to prove the first main theorem.

Theorem 1 Assume that an AVL tree T initially has $|T|$ nodes. Furthermore, assume that k insertions, m deletions, and a number of rebalancing operations are performed. If $N = |T| + 2k$, then the number of rebalancing operations is at most

$$(k+m)\lfloor \log_\phi(N+2) + \log_\phi(\tfrac{\sqrt{5}}{2}) - 2 \rfloor - m.$$

Proof If an insertion creates a negative unit, then this unit is associated with a node which has relaxed height zero. From Lemma 3, we know that unless the negative unit disappears, it is associated with a node of larger relaxed height every time operation (n) has been applied involving this unit. From Corollary 2, it therefore follows that a particular negative unit can be moved by operation (n) at most $\lfloor \log_\phi(N+2) + \log_\phi(\frac{\sqrt{5}}{2}) - 2 \rfloor$ times. As operation (n) can only be applied if a negative unit is involved, it follows from Proposition 2 that operation (n) can be applied at most $k\lfloor \log_\phi(N+2) + \log_\phi(\frac{\sqrt{5}}{2}) - 2 \rfloor$ times.

The proof for deletion is similar, except that Lemma 4 is used. However, when a positive unit is created, it is associated with a node of relaxed height at least one, so the bound becomes $m(\lfloor \log_\phi(N+2) + \log_\phi(\frac{\sqrt{5}}{2}) - 3 \rfloor)$.

The theorem follows from adding up the bounds for the operations (n) and (p). □

4 Amortized Constant Insertion Rebalancing

We prove that in the semi-dynamic case, where only searches and insertions are allowed, rebalancing after the insertion of p keys into a relaxed AVL tree can be performed in linear time, $O(p)$. In other words, rebalancing after an insertion is amortized constant.

Theorem 2 In the semi-dynamic case, starting with an empty tree, rebalancing after an insertion is amortized constant time.

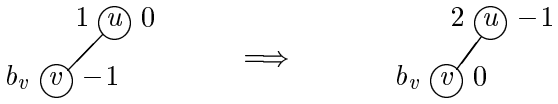
Proof We use a potential function to compute the time complexity [29]. If T is a relaxed AVL tree, we define its potential to be the number of nodes in T with relaxed balance factor zero.

Tag values of -1 are created by insertions only, as shown in Proposition 2. When a tag value of -1 is created, the potential can increase by at most a constant, since only a constant number of nodes are involved. Similarly, when a tag value of -1 is removed, the potential can only increase by at most a constant. Thus, due to these operations, p insertions will give rise to an increase in the potential by at most $O(p)$.

The remaining operations are the ones which just move a tag value of -1. Below we show that any such operation will decrease the potential by at least one. Since the potential is always non-negative, only $O(p)$ of these operations can be carried out, and the result will follow.

Considering operation (n), the tag value of -1 disappears unless $t_u = 0$ and $r(u) = 0$. If $r(u) = 0$, then $b_u \in \{0, 1\}$. If $b_u = 0$, then the top node after the operation has relaxed balance factor 1. Thus, none of the operations (n1) through (n4) will be applied, and we get the desired drop in potential.

We can now assume that $t_u = 0$, $r(u) = 0$, and $b_u = 1$. So we examine a special case of operation (n), immediately followed by one of the operations (n1) through (n4). This special case of operation (n) is the following.



If operation (n1) is applied immediately after, then the -1 disappears if $l(v) = 1$, so assume that $l(v) = 0$. Since operation (n1) requires that $b_v \geq 0$, this implies that $b_v = 0$. Thus, $b_v - 1$ and $1 - l(v)$ are different from zero, and we obtain a decrease in potential.

In the operations (n2) and (n3) the tag value of -1 disappears, and if operation (n4) is applied, the tag value of -1 on node w disappears. □

5 Conclusion

The contribution of this paper is to provide a proof of complexity for the relaxed AVL tree which was introduced in [25, 26]. A proof of complexity is important for several reasons. First of all, before one implements a complicated data structure, it is desirable to know in advance that it is efficient, and there are many examples of data structures with a reasonable design which turn out to be inefficient, i.e., to have super logarithmic rebalancing.

Another reason for wanting a proof of complexity comes from the desire to design rebalancing operations with the aim of obtaining a high degree of parallelism in a distributed implementation. With many restrictions on when operations can be applied, it is easier to ensure a good complexity. However, in order to obtain a high degree of parallelism, restrictions should be as few as possible. Without a proof of complexity, whenever there is a choice in the design, one simply has to guess, running the risk of either ruining the complexity or limiting the degree of parallelism unnecessarily.

Finally, in an actual implementation, one might want to make minor changes throughout the operations. Once a proof has been set up, it is fairly easy to go through the details of the proofs to check that such changes do not effect the overall complexity.

With the proofs in this paper, these concerns have been addressed for AVL trees with relaxed balance.

Acknowledgment

The author would like to thank the anonymous referees for valuable comments which have improved the presentation of the result.

References

[1] G. M. Adel'son-Vel'skiĭ and E. M. Landis. An Algorithm for the Organisation of Information. *Doklady Akadamii Nauk SSSR*, 146:263–266, 1962. In Russian. English translation in *Soviet Math. Doklady*, 3:1259–1263, 1962.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[3] R. Bayer and E. McCreight. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica*, 1:173–189, 1972.

[4] J. Boyar, R. Fagerberg, and K. S. Larsen. Amortization Results for Chromatic Search Trees, with an Application to Priority Queues. In *Fourth International Workshop on Algorithms and Data Structures*, volume 955 of *Lecture Notes in Computer Science*, pages 270–281. Springer-Verlag, 1995.

[5] 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(3):504–521, 1997.

[6] J. F. Boyar and K. S. Larsen. Efficient Rebalancing of Chromatic Search Trees. In *Proceedings of the Third Scandinavian Workshop on Algorithm Theory*, volume 621 of *Lecture Notes in Computer Science*, pages 151–164. Springer-Verlag, 1992.

[7] J. F. Boyar and K. S. Larsen. Efficient Rebalancing of Chromatic Search Trees. *Journal of Computer and System Sciences*, 49(3):667–682, 1994.

[8] L. J. Guibas and R. Sedgewick. A Dichromatic Framework for Balanced Trees. In *Proceedings of the 19th Annual IEEE Symposium on the Foundations of Computer Science*, pages 8–21, 1978.

[9] S. Hanke. The Performance of Concurrent Red-Black Tree Algorithms. In *Third International Workshop on Algorithm Engineering, WAE'99*, volume 1668 of *Lecture Notes in Computer Science*, pages 286–300. Springer-Verlag, 1999.

[10] S. Hanke, T. Ottmann, and E. Soisalon-Soininen. Relaxed Balanced Red-Black Trees. In *Proc. 3rd Italian Conference on Algorithms and Complexity*, volume 1203 of *Lecture Notes in Computer Science*, pages 193–204. Springer-Verlag, 1997.

[11] S. Huddleston and K. Mehlhorn. A New Data Structure for Representing Sorted Lists. *Acta Informatica*, 17:157–184, 1982.

[12] J. L. W. Kessels. On-the-Fly Optimization of Data Structures. *Communications of the ACM*, 26:895–901, 1983.

[13] D. E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, 1968.

[14] K. S. Larsen. AVL Trees with Relaxed Balance. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 888–893. IEEE Computer Society Press, 1994.

[15] K. S. Larsen. Amortized Constant Relaxed Rebalancing using Standard Rotations. *Acta Informatica*, 35(10):859–874, 1998.

[16] K. S. Larsen and R. Fagerberg. B-Trees with Relaxed Balance. In *Proceedings of the 9th International Parallel Processing Symposium*, pages 196–202. IEEE Computer Society Press, 1995.

[17] K. S. Larsen and R. Fagerberg. Efficient Rebalancing of B-Trees with Relaxed Balance. *International Journal of Foundations of Computer Science*, 7(2):169–186, 1996.

[18] K. S. Larsen, T. Ottmann, and E. Soisalon-Soininen. Relaxed Balance for Search Trees with Local Rebalancing. In *Fifth Annual European Symposium on Algorithms*, volume 1284 of *Lecture Notes in Computer Science*, pages 350–363. Springer-Verlag, 1997.

[19] K. S. Larsen, E. Soisalon-Soininen, and P. Widmayer. Relaxed Balance through Standard Rotations. In *Fifth International Workshop on Algorithms and Data Structures*, volume 1272 of *Lecture Notes in Computer Science*, pages 450–461. Springer-Verlag, 1997.

[20] D. Maier and S. C. Salveter. Hysterical B-Trees. *Information Processing Letters*, 12(4):199–202, 1981.

[21] L. Malmi and E. Soisalon-Soininen. Group Updates for Relaxed Height-Balanced Trees. In *Proceedings of the Eighteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 358–367, 1999.

[22] K. Mehlhorn and A. Tsakalidis. An Amortized Analysis of Insertions into AVL-Trees. *SIAM Journal on Computing*, 15(1):22–33, 1986.

[23] O. Nurmi and E. Soisalon-Soininen. Uncoupling Updating and Rebalancing in Chromatic Binary Search Trees. In *Proceedings of the Tenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 192–198, 1991.

[24] O. Nurmi and E. Soisalon-Soininen. Chromatic Binary Search Trees—A Structure for Concurrent Rebalancing. *Acta Informatica*, 33(6):547–557, 1996.

[25] O. Nurmi, E. Soisalon-Soininen, and D. Wood. Concurrency Control in Database Structures with Relaxed Balance. In *Proceedings of the 6th ACM Symposium on Principles of Database Systems*, pages 170–176, 1987.

[26] O. Nurmi, E. Soisalon-Soininen, and D. Wood. Relaxed AVL Trees, Main-Memory Databases and Concurrency. *International Journal of Computer Mathematics*, 62:23–44, 1996.

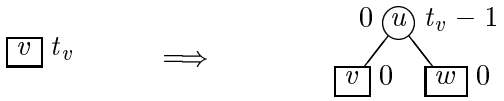
[27] T. Ottmann and E. Soisalon-Soininen. Relaxed Balancing Made Simple. Technical Report 71, Institut für Informatik, Universität Freiburg, 1995.

[28] E. Soisalon-Soininen and P. Widmayer. Relaxed Balancing in Search Trees. In *Advances in Algorithms, Languages, and Complexity*, pages 267–283. Kluwer Academic Publishers, 1997.

[29] R. E Tarjan. Amortized Computational Complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985.

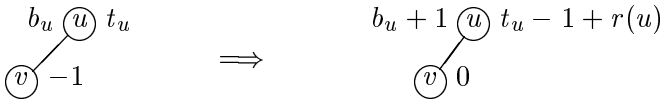
A Operations Used in the Proofs

Relaxed balance factors are shown to the left of nodes and tag values to the right. We use "rbf" as short for "relaxed balance factor".

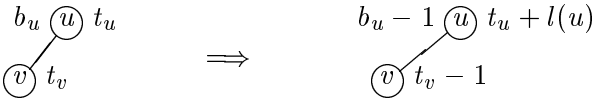


(i) Insertion of w to the right of v.

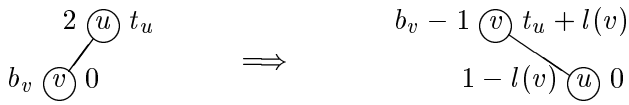
(d) Deletion of w.



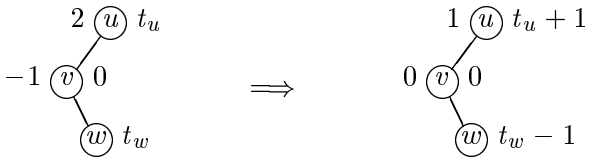
(n) Moving up negative tags. Requirement: $t_u \geq 0$.



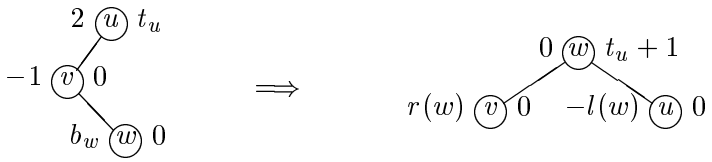
(p) Moving up positive tags. Requirement: $t_u \geq 0$, $t_v > 0$.



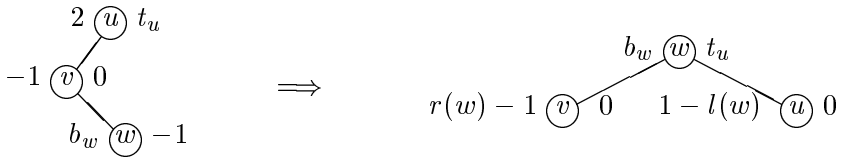
(n1) Too large rbf: single rotation. Requirement: $b_v \geq 0$.



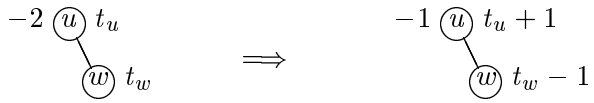
(n2) Too large rbf: moving tags. Requirement: $t_w > 0$.



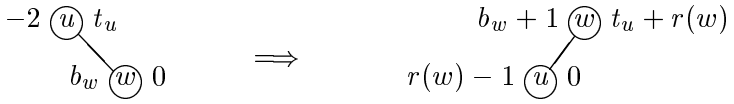
(n3) Too large rbf: double rotation ($t_w = 0$).



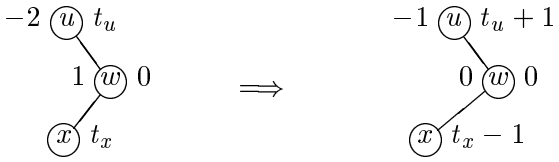
(n4) Too large rbf: double rotation ($t_w = -1$).



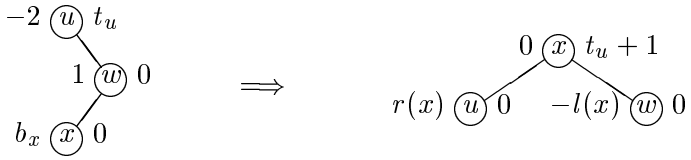
(p0) Too small rbf: moving tags. Requirement: $t_w > 0$.



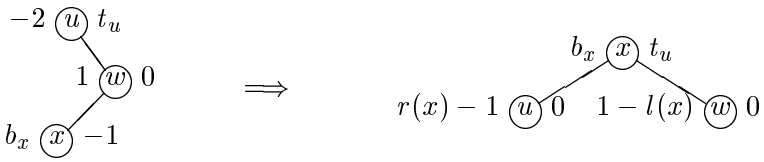
(p1) Too small rbf: single rotation. Requirement: $b_w \leq 0$.



(p2) Too small rbf: moving tags. Requirement: $t_x > 0$.

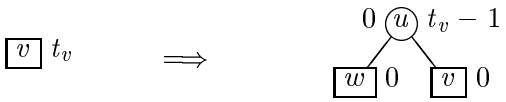


(p3) Too small rbf: double rotation ($t_x = 0$).



(p4) Too small rbf: double rotation ($t_x = -1$).

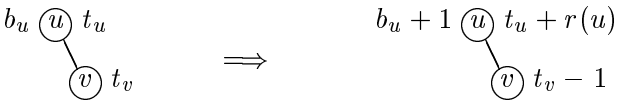
B Additional Operations to Complete the Set



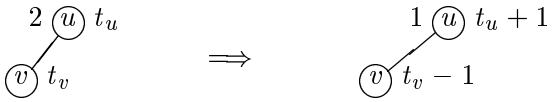
(i') Insertion of w to the left of v.

(d') Deletion of w.

(n') Moving up negative tags. Requirement: $t_u \geq 0$.



(p') Moving up positive tags. Requirement: $t_u \geq 0$, $t_v > 0$.



(n0) Too large rbf: moving tags. Requirement: $t_v > 0$.