# Amortized Constant Relaxed Rebalancing using Standard Rotations

Kim S. Larsen[*]
Odense University

**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. This paper defines the first relaxed binary search tree with amortized constant rebalancing using only standard single or double rotations.

## 1 Introduction

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. The motivation is two-fold: If search and update requests for a search tree come in bursts (possibly from several external sources), the search tree may occasionally not be able to process the requests as fast as it might be desirable. For this reason, it would be convenient to be able to "turn off" rebalancing for a short period of time in order to speed up the request processing. However, when the burst is over, the tree should be rebalanced again, while searching and updating still continues at a slower pace, so preferably rebalancing should still be efficient.

The other 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

[*]This work was carried out while the author was visiting the Department of Computer Sciences, University of Wisconsin at Madison. 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).

particular, and all the locking involved would seriously limit the amount of parallelism possible in the system. See earlier papers on relaxed balance for more details.

Uncoupling the rebalancing from the updating can help in these situations, but of course, if rebalancing is postponed for too long, the tree can become completely unbalanced, which is also the reason why it is challenging to prove the complexity results for them.

We give a brief account of the work done 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 presented in [18] and proofs of complexity for the rebalancing, matching the complexity from the sequential case, was obtained in [13]. A different AVL-based version with some especially nice properties was treated in [22, 17]. The first relaxed version of B-trees [3] is also from [18] with proofs of complexity matching the sequential ones in [14, 15]. Since [14, 15] really treat $(a, b)$-trees [11], they also provide proofs for 2-3 trees [10, 2], as well as for $(a, b)$-trees with other choices of $a$ and $b$. A relaxed version of red-black trees [8] was introduced in [19] and complexities matching the sequential case were established in [6, 7] and [4, 5] for variants of the original proposal. In [22], some of these early results are surveyed. In [20, 16], it is shown how a large class of standard search trees can automatically be equipped with relaxed balance. In [9], a version of red-black trees based on these general ideas is presented.

Now we turn to the subject of this paper. In [5], in addition to a number of other topics, the authors prove that it is possible to implement relaxed balance in a binary search tree in such a way that rebalancing becomes amortized constant. This is, in our opinion, one of the most important results in the world of relaxed balance. For practical reasons because it shows that very few rebalancing operations are carried out in the worst case, and because in the parallel application, rebalancing will almost always take place very close to the leaves, and will therefore very rarely interfere with requests entering through the root. For theoretical reasons, it is important because the red-black tree is one of the search trees with most properties. It has worst-case logarithmic time rebalancing, amortized constant time rebalancing, and worst-case constant number of restructuring operations per update, so in that sense it is more advanced than AVL-trees, for instance. The result from [5] is the first example of a relaxed version having all of the same properties as one of the most advanced standard search trees. However, we see the following practical and theoretical problems with the result from [5]:

- Some rebalancing operations are not standard single or double rotations, but larger six-nodes transformations.
- There are many rebalancing operations.
- The proof that rebalancing is amortized constant is very long, and it is quite time consuming to check all the details.

From a practical point of view, the large set of operations as well as the large size of some of the operations makes it more time consuming to decide which operation to apply, and it takes longer to carry it out. The latter is particularly problematic in a parallel environment, where any unnecessary locking must be avoided, and where locks should be held for as short time periods as possible.

We address all the objections, presenting a smaller collection of operations, all of which are single or double rotations. This collection could be seen as a relaxed version of the operations from [21]. Note that in [5], there are even more operations than it appears, since root operations are not listed. Additionally, we give a very short proof of amortized constant time rebalancing, with details that are quite easy to check. We also prove that this new collection of operations has all the other properties from [7, 5], i.e., each update gives rise to at most a logarithmic number of rebalancing operations, of which at most a constant number are restructuring operations.

Note that in the general result from [16], standard rebalancing operations are used in a general rebalancing scheme. However, in order to avoid interference and deadlocks, these operations are embedded in some larger areas (referred to as operation and synchronization areas), so the resulting rebalancing operations are in fact gigantic.

This present paper contains the first relaxed binary search tree with amortized constant rebalancing using only standard single or double rotations.

## 2 A Relaxed Red-Black Tree

The search trees we are going to use will be *leaf-oriented.* This means that only the leaves contain keys. The internal nodes contain routers, which are of the same type as keys and which direct the searches to the right location as usual in a search tree. However, routers are values that may not be present as keys in our tree. This is because we do not want to have to update routers whenever a deletion takes place. For an internal node, the keys in its left subtree are smaller than or equal to its router, and the keys in its right subtree are larger.

We define a relaxed red-black tree as a relaxation of the balance constraints from the standard case. This is exactly as in [5]. Instead of just using the two colors, red and black, we use *weights*. So each node in the tree has a non-negative integer weight. We refer to nodes with weight zero as red and nodes with weight one as black. If a node has a larger weight, we call it *overweighted*, and its amount of *overweight* is the weight it has in excess of one. The *weight of a path* is the sum of the weights of the nodes on the path, and two *consecutive* nodes means consecutive on some path, i.e., one node is a child of the other. Now, the only requirements in this relaxed red-black tree are that *all paths from the root to a leaf have the same weight* and that *leaves have weight at least one*.

In such a relaxed red-black tree, two consecutive red nodes are referred to as a *red conflict*, and an overweighted node as a *weight conflict*. A standard red-black tree, or simply a red-black tree, can be defined as a relaxed red-black tree without any conflicts. The purpose of the rebalancing operations is to transform a relaxed red-black tree into a standard red-black tree by removing all conflicts. The difficulty in proving results for relaxed structures is that rebalancing operations and updates can be interleaved in any order, as opposed to the standard case where rebalancing is performed (and finished) immediately after an update.

Since the tree is leaf-oriented, it is always a *full* tree, i.e., every internal node has two children. Operations for the updates, *insertion* and *deletion*, as well as for rebalancing are shown in the appendix. Note that all operations preserve the tree as a relaxed red-black tree, i.e., leaves are non-red and all paths have the same weight. As usual in search tree papers, we do not show the subtrees, since the order in which subtrees from before a rebalancing operation is carried out should be attached again after the operation is uniquely determined: the only way to preserve the structure as a search tree is by removing the subtrees in-order before the operation is carried out and attaching them again in-order after the operation has been carried out. We do not list symmetric operations, nor discuss these in the proofs. Nodes that are definitely leaves are marked with a square. Internal nodes and nodes that may or may not be leaves are marked with a circle.

## 3 Complexity

In this section, we prove that as long as there are conflicts in a relaxed red-black tree, some rebalancing operation will be applicable. Then we prove that after $k$ updates at most $O(k)$ rebalancing operations can be carried out, i.e., a constant number of rebalancing operations per update, assuming that we start with an empty tree. Starting with a red-black tree, a bound of $O(k \log(n+i))$ is shown, where $n$ is the size of the tree and $i$ is the number of insertions (so $i \leq k$). Finally, we prove that at most $O(k)$ restructuring operations can be applied, even if we start with a red-black tree.

### The Collection of Operations is Complete

First we prove that rebalancing does not terminate before the tree is again in balance.

**Theorem 1** If a relaxed red-black tree is not red-black, then a rebalancing operation can be applied.

**Proof** Assume that there is a red conflict in the tree, and consider a top-most of these, i.e., a red conflict at a smallest distance from the root. Either its top node is the root, in which case *red-root* can be applied. Otherwise, the top node of the conflict has a parent. This parent is not red, because then the conflict under consideration would not be top-most. So, the parent has weight at least one. Now consider the sibling of the top node of the conflict. If it is red, then *red-push1* or *red-push2* can be applied, and otherwise, *red-dec1* or *red-dec2* can be applied.

We have proven that if there is a red conflict, then a rebalancing operation can be applied. Now assume that there are no red conflicts, but there is a weight conflict. Consider a weight conflict at a largest distance from the root. If the overweight is located at the root, then *weight-root* can be applied. Otherwise, the overweighted node has a parent. Let $u$ denote the sibling of the overweighted node. We divide into three cases depending on whether $u$ is red, overweighted, or black.

If $u$ is red, then it has children, since leaves are not red. Neither the children of $u$ nor its parent is red, since, by assumption, there are no red conflicts. Since we are considering a conflict at a largest distance from the root, the children of $u$ are not overweighted, thus they are black, and *weight-temp* can be applied. If $u$ is overweighted, then *weight-dec3* can be applied. If $u$ is black, we consider $u$'s children, which it has, since otherwise we would not have the same weight on all paths going from the root to a leaf. If $u$'s right child is red, then *weight-dec1* can be applied. Otherwise, if $u$'s left child is red, then *weight-dec2* can be applied. Finally, if none of them are red, then *weight-push* can be applied. □

## Amortized Constant Rebalancing

We prove that starting with an empty tree, rebalancing is amortized constant. The following observations can easily be, and have been, verified by inspection of the operations in the appendix.

**Observation 1**

- *red-root*, *red-dec1*, and *red-dec2* decrease the total number of red conflicts in the tree.
- *weight-root*, *weight-dec1*, *weight-dec2*, and *weight-dec3* decrease the total amount of overweight in the tree.
- An *insertion* increases the number of red conflicts by at most one, and a *deletion* increases the amount of overweight by at most one.
- No rebalancing operation increases the number of red conflicts nor the number of weight conflicts in the tree.

□

We use a potential function to compute the time complexity [23]. For the purpose of defining this potential function, we have to keep an eye on certain patterns in the tree.

**Definition 1** We define three patterns that we call *potential types.* See figure 1. □

The crucial part of the proof is finding the correct potential types and analyzing how the number of these different types in the tree changes as rebalancing operations are carried out.

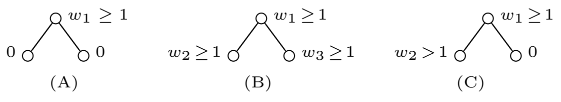


Figure 1: Potential types A, B, and C.

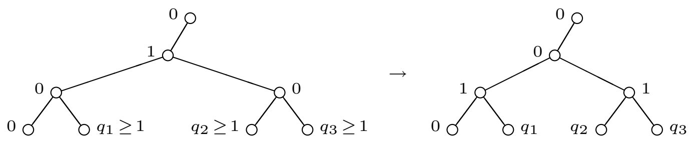


Figure 2: Red-push1 when the number of red conflicts is not reduced.

**Lemma 1**

1. If *red-push1* or *red-push2* do not reduce the total number of red conflicts in the tree, then they reduce the number of potential types (A), and increase the number of potential types (B) and (C) by at most one each.

2. The operation *weight-temp* reduces the number of potential types (C) by one, and does not increase the number of the other potential types.

3. If *weight-push* does not decrease the total amount of overweight in the tree, then it decreases the number of potential types (B) by two, and only increases the number of potential types (C); and by at most two.

**Proof** We prove the three parts separately.

1. We only prove the result for *red-push1*. The proof for *red-push2* is similar. Since the number of red conflicts is not reduced by the operation, we can deduce a fair amount about the context in which it must be applied; see figure 2. The red conflict that disappears must be replaced by another one at the top of the operation. Additionally, none of the other three nodes at the same level as the bottom-most node of the red conflict can be red.

   Now it is easy to see that the number of potential types (A) is reduced by one, and the number of potential types (B) is increased by one. Additionally, the number of potential types (C) may increase by one.

2. Easy observation.

3. Clearly, no configurations of potential type (A) can be created. Additionally, if the total amount of overweight is not decreased, then $w_1 \geq 1$. This means that no new configuration of potential type (B) can be created right above the root of the operation; if there is one there after the operation, then it was already present before the operation. The same is true for the position at the left-most leaf of the operation. Thus, no new configurations of potential type (B) can be created.

   Two configurations of potential type (B) disappear. That is the one located at the node which changes weight from one to zero, as well as the one located at the root of the operation (recall that $w_1 \geq 1$).

   The number of configurations of potential type (C) increases by at most two. One can appear at the root of the operation (if $w_2 \geq 3$) and one can appear right above the root of the operation.

□

The main theorem can now be proven.

**Theorem 2** Rebalancing is amortized constant.

**Proof** We use the standard potential function technique [23]. Let the potential types (A), (B), and (C) have weights 4, 2, and 1, respectively, and let $a(T)$, $b(T)$, and $c(T)$ denote the number of configurations in the tree $T$ of types (A), (B), and (C), respectively. We refer to $4a(T) + 2b(T) + c(T)$ as the *weighted sum*.

When a rebalancing operation reduces the total number of red conflicts or weight conflicts, it may create some configurations of potential types (A), (B), and (C), and it may remove some. Let $m$ be the maximum increase of the weighted sum that may accompany any rebalancing operation.

Let $r(T)$ and $w(T)$ denote the number of red conflicts and the total amount of overweight, respectively, in a tree $T$, The potential of a tree $T$ is denoted $\Phi(T)$, and is defined to be $\Phi(T) = (m+1)(r(T) + w(T)) + 4a(T) + 2b(T) + c(T)$. Clearly, $\Phi(T)$ is always non-negative.

Since *insertion* and *deletion* only change a constant number of nodes, the potential increase due to these operations is bounded by a constant.

In order to prove that each update gives rise to at most an amortized constant number of rebalancing operations, we prove that whenever a rebalancing operation is carried out, there will be a potential decrease of at least one. Since we start with an empty tree with potential zero, the result will follow.

By observation 1 and the choice of $m$, this holds for the operations *red-root*, *red-dec1*, *red-dec2*, *weight-root*, *weight-dec1*, *weight-dec2*, and *weight-dec3*. The remaining operations are the ones we considered in lemma 1. If these operations reduce the number of conflicts in the tree, again by the choice of $m$, we are done. The remaining cases are exactly the ones addressed in lemma 1.

For *red-push1* and *red-push2*, the potential change is at most $-4+2+1=-1$. For *weight-temp*, it is $-1$, and for *weight-push*, it is at most $-4+2=-2$. □

## Worst-Case Logarithmic Rebalancing

The operations from [5] are basically the same as the ones from [7], and in the latter it was shown that for any $k$, starting with a red-black tree (not necessarily empty), $k$ updates give rise to at most $O(k \log(n+i))$ rebalancing operations, where $n$ is the number of leaves and $i$ is the number of insertions since the tree was last red-black. If this result held only when updates are made into an initially empty tree, the amortized result from the present paper would always be better. However, the result from [7] holds even when updates are made into an initially non-empty tree, so in that case, the results are incomparable. For instance, at most a logarithmic number of rebalancing operations are necessary in order to re-establish the red-black invariant if one update is made into a tree which is red-black. This does not follow from the amortized constant result. However, the operations in our paper also have this property, and the proof from [7] carries over with minor modifications, except that an entirely new lemma is required in order to deal with the operation *weight-temp*. We give the proof below.

When weights become large, many rebalancing operations can be necessary. In order to obtain a good result, it is necessary to be able to argue that many deletions have taken place to create the large weights. Therefore, we must keep track of the number and the location of deletions. To this end, we maintain a count function, $c$, from the set of nodes in the tree to the natural numbers. Its purpose is to remember how many nodes there have been in a given subtree, including the current nodes. The function is defined as follows.

Initially, we have a red-black tree, and we define all nodes $u$ to have $c(u) =$

1. We now describe how each operation changes the count function. For an insertion, the leaf before the operation is the internal node after the operation, and it keeps its count value. Thus, the two leaves after the operation are new, and when such a new node, $u$, is created, we define $c(u) = 1$. A deletion involves three nodes: the leaf $u$ to be deleted, its sibling $v$, and their parent $x$. Two nodes disappear. We consider $x$ to be the node which remains. The function value $c(x)$ for the parent $x$ of the deleted leaf is changed to $c(u) + c(v) + c(x)$, referring to the values from before the operation is carried out. This ensures that the sum of all the count values equals the number of nodes which are or have been in the tree since it was red-black.

When a rebalancing operation changes the structure of the tree, nodes are moved around. To preserve the function $c$, we have to define where the nodes move to, i.e., given a fixed node $u$ immediately prior to a rebalancing operation, which node is $u$ after the restructuring.

For the operations which only change weights, we use the obvious identification by location. For the restructuring operations, the node which is the root before the operation is also the root after the operation. The identity of the remaining nodes is determined by the ordering (their keys): ignoring the root, the $i$th node encountered in an in-order traversal of the nodes involved in the operation before the operation is carried out is again the $i$th node in an in-order traversal after the operation is carried out.

We can now establish an exponential connection between the path weights and the number of nodes which are or have been in a subtree. We let $T_u$ denote the *subtree rooted by* $u$, refer to $c(u)$ as the *count of* $u$, refer to $\sum_{v \in T_u} c(v)$ as the *count sum of* $u$, and define the *weighted height* of a node to be the sum of all the weights from that node down to a leaf (recall that no matter which leaf is chosen, this weight is the same, so it is well-defined).

**Lemma 2** If the weighted height of $u$ is $w$, then $\sum_{v \in T_u} c(v) \geq 2^w - 1$.

**Proof** By induction on the number of operations performed on the tree. Notice first that for any node $u$, $c(u)$ is initialized to 1, and from then on, $c(u)$ can only increase.

The base case is when no operations have been performed, so the tree is red-black. We establish the result for this case by a proof by induction on height. If a node $u$ is a leaf, then $w = 1$, so $2^w - 1 = 1 = c(u)$. If $u$ is not a leaf, it has two children $v_1$ and $v_2$. Since the tree is red-black, $u$ has weight at most one, so if $u$ has weighted height $w$, its children have weighted height at least $w-1$. Thus, by induction, $\sum_{v \in T_{v_1}} c(v) \geq 2^{w-1} - 1$ and $\sum_{v \in T_{v_2}} c(v) \geq 2^{w-1} - 1$. Since $c(u) = 1$, $\sum_{v \in T_u} c(v) \geq (2^{w-1} - 1) + (2^{w-1} - 1) + 1 = 2^w - 1$.

For the induction step, we consider each operation in turn.

For *insertion*, the new nodes are given weight 1 and the count function is set to 1. No other nodes have their weighted heights changed or their count sum decreased.

For *deletion*, count sums and weighted heights remain unchanged for all nodes in the tree which are remaining after the operation.

For the rebalancing operations, general arguments can handle all cases. With the exception of *red-root* and *weight-root*, the weighted heights of the root of the operations as well as their count sums remain unchanged. Furthermore, the result holds for leaves of an operation which keep the same subtrees they had before the operation was carried out, provided that their own weight is not increased (this includes *weight-root*). Finally, the result holds for a node of weight 0 or 1, provided that it is a leaf or that the result holds for both of its subtrees (the same proof as was used in the base case). By these arguments, the result holds for all nodes in the tree, no matter which operation was applied. $\square$

Let $n$ be the number of nodes in the structure at a given time when the tree is red-black. An upper bound on the number of nodes in the structure at a later point, after $i$ insertions have taken place, is $n + 2i$. The following corollary of the lemma above bounds the maximum weighted heights in the tree.

**Corollary 1** The largest weighted height any node can have is bounded by $\lfloor \log_2(n + 2i + 1) \rfloor$.

**Proof** Clearly, by definition of $c$, the count sum of the root equals $n + 2i$. Thus, if $w$ is the weighted height of the root, $n+2i \geq 2^w - 1$. So, the largest weighted height any node can have is bounded by $\lfloor \log_2(n + 2i + 1) \rfloor$, since the root has the largest weighted height in the tree, and since this weighted height must be an integer. $\square$

This can be used to bound the number of operations which can be carried out, since the operations, with the exception of *weight-temp*, have been designed in such a way that if they do not remove a problem of imbalance, they move it to a node of a larger weighted height. To be precise, the location of a red conflict is the bottom-most node of the two red nodes. The location of a weight conflict is the node which is overweighted. Now, whenever a conflict is removed by a rebalancing operation just to be introduced again higher up in the tree (the operations *red-push1*, *red-push2*, and *weight-push* do this), we say that the conflict has *moved*.

Finally, we define the *weighted height of a conflict*: the weighted height of a red conflict is the weighted height of the node where it is located. If a node has weight $w_1 > 1$ and it has weighted height $w$, we consider this to be $w_1 - 1$ weight conflicts of weighted height $w - w_1 + 2, w - w_1 + 3, \ldots, w$, respectively. We also refer to the node as having $w_1 - 1$ *units of overweight*. When a unit of overweight is moved or removed, we always assume that it is the one with the largest weighted height.

**Proposition 1** The operations *red-root*, *red-dec1*, and *red-dec2* remove at least one red conflict, and the operations *weight-root*, *weight-dec1*, *weight-dec2*, and *weight-dec3* remove at least one unit of overweight. All these operations leave the remaining conflicts at the same weighted height as before the operation was carried out, and no new conflicts are created.

**Proof** Easy inspection of the operations in the appendix. Note that using the weighted height of a conflict instead of simply the weighted height of the location of the conflict was necessary to make the result hold for the nodes which have overweight $w_2 - 1$ after an operation (see the appendix). □

**Proposition 2** The operations *red-push1*, *red-push2*, and *weight-push* either remove a conflict or move a conflict such that it is located at a larger weighted height after the operation. Other conflicts remain at the same weighted height as before the operation was carried out, and no new conflicts are created.

**Proof** Easy inspection of the operations in the appendix. For *red-push1* and *red-push2*, the new location for the conflict, if it does not disappear, is the root of the operation after it has been carried out. Clearly, the weighted height of the conflict has increased with one. For *weight-push*, the increase in weighted height, if the conflict is moved, is $w_1 + 1$, which is at least one. □

**Proposition 3** The operation *weight-temp* does not change the weighted height of any conflicts, and no new conflicts are created.

**Proof** Easy inspection of operation *weight-temp* in the appendix. □

As already mentioned, *weight-temp* does not have the desirable property that it moves conflicts to a node with a larger weighted height. Thus, a bound for the number of *weight-temp* operations which are carried out must be obtained in a different way.

If an overweighted node has a red parent, which in turn has a non-red sibling and a non-red parent, we refer to this as a *weight-temp configuration*, since the operation *weight-temp* creates such configurations. If $u$ is the overweighted node in such a configuration, we refer to the other nodes as the parent, the uncle, and the grandparent of the configuration (or simply of $u$).

**Lemma 3** Weight-temp configurations can only disappear through the application of an operation which decreases the total number of conflicts in the tree.

**Proof** By inspection of the operations in the appendix. Clearly, in order to change the configuration, an operation must overlap nodes in the configuration. Let $u$ be the overweighted node in the configuration (the one with a red parent).

The operation *insertion* cannot make the configuration disappear (note that if the uncle of the configuration has weight one, then it cannot be a leaf), and if a *deletion* changes the situation, it is because $w_1 = 0$ and $w_3 > 1$ (within the *deletion* operation). However, we must have that $w_2 \geq w_3$, so the total amount of overweight decreases by $w_2 - 1$.

It is only necessary to discuss the rebalancing operations which do not necessarily decrease the total number of conflicts in the tree, i.e., *red-push1*, *red-push2*, *weight-temp*, and *weight-push*.

The operation *red-push1* can be applied in this situation if $u$ is the top-node of the *red-push1* operation. In that case, the amount of overweight as well as the number of red conflicts decrease. It can also be applied at a position where $u$'s uncle is the top node of the *red-push1* operation. In that case, a red conflict disappears. The operation *red-push2* is similar.

The operation *weight-temp* can only be applied to nodes in this configuration if either $u$ or its uncle is the root of the *weight-temp* operation. Since the weight of the root of a *weight-temp* operation is not changed when the operation is carried out, neither is the configuration.

Finally, *weight-push* can overlap the configuration if the parent of $u$ is the root of the *weight-push* operation. However, in that case, the amount of overweight decreases. It can also overlap if the uncle of $u$ is the root of *weight-push*. In that case, the uncle of $u$ has its weight increased, so it will still be non-red. □

**Corollary 2** If $i$ *insertions* and $d$ *deletions* are made into a red-black tree, at most $i + d$ *weight-temp* operations can be applied.

**Proof** By observation 1, no rebalancing operation increases the number of conflicts. Since *insertion* and *deletion* create at most one conflict each time they are applied, the total number of conflicts ever introduced in the tree is bounded by $i + d$.

By lemma 3, a weight-temp configuration can only be removed through the application of an operation which decreases the total number of conflicts in the tree. Thus, weight-temp configurations can be removed at most $i + d$ times.

After $i$ insertions and $d$ deletions, by theorem 1 and 2, the tree will eventually become red-black. Since a weight-temp configuration contains overweight, this will therefore eventually be removed. Thus, at most $i + d$ weight-temp configurations can ever be created. □

We can now prove that starting with a red-black tree, each update gives rise to at most a logarithmic number of rebalancing operations.

**Theorem 3** Rebalancing is worst-case logarithmic.

**Proof** We show that if $k$ updates are made to a red-black tree with $n$ keys, $k(\lfloor \log_2(n + 2i + 1) \rfloor + 1)$ rebalancing operations are sufficient to balance the tree, i.e., to make it red-black again. Here $k = i + d$, where $i$ is the number of insertions and $d$ is the number of deletions.

Observe by inspection of the appendix that the operation *insertion* creates at most one red conflict, and creates it with a weighted height of one. The operation *deletion* creates at most one additional unit of overweight, and it gets weighted height at least two (we only create *new* overweight if $w_1 \geq 1$ and $w_3 \geq 1$).

By propositions 1, 2, and 3, no rebalancing operations decrease the weighted height of a conflict. By proposition 1, the operations *red-root*, *red-dec1*, and *red-dec2* can be applied at most $i$ times, and the operations *weight-root*, *weight-dec1*, *weight-dec2*, and *weight-dec3* can be applied at most $d$ times.

By corollary 1, the largest weighted height any node can have is bounded by $M = \lfloor \log_2(n + 2i + 1) \rfloor$. Since, by proposition 2, every time the operations *red-push1* and *red-push2* are applied, they move a conflict to a node with a larger weighted height, these operations can be applied at most $i(M - 1)$ times. Similarly, also by proposition 2, the operation *weight-push* can be applied at most $d(M - 2)$ times.

By corollary 2, at most $i + d$ *weight-temp* operations can be applied.

Summing up, at most $i(M + 1) + dM \leq k(\lfloor \log_2(n + 2i + 1) \rfloor + 1)$ operations can be applied. □

**Worst-Case Constant Restructuring**

Even starting with a red-black tree, each update gives rise to at most a constant number of restructuring operations.

**Theorem 4** Restructuring is worst-case constant.

**Proof** The rebalancing operations which make structural changes are *red-dec1*, *red-dec2*, *weight-temp*, *weight-dec1*, *weight-dec2*, and *weight-dec3*. By proposition 1 and corollary 2, these operations can be applied at most $2(i+d)$ times when starting with a red-black tree. □

## 4 Concluding Remarks

We have defined a relaxed red-black search tree with rebalancing operations which are single or double rotations. Starting with a red-black tree (which could be empty), rebalancing has been shown to be worst-case logarithmic and the number of restructuring operations worst-case constant per update. Starting with an empty tree, rebalancing has been shown to be amortized constant.

Notice that since each potential type contributes only a constant amount to the potential of a whole relaxed red-black tree, the potential of a tree is at most linear in the size of the tree. This implies that if we start with a red-black tree of size $n$ instead of an empty tree, after $\Omega(n)$ updates, rebalancing will again be amortized constant.

**Acknowledgments**

The author would like to thank the three very competent anonymous referees for many excellent suggestions, which have definitely improved the content of the paper as well as the presentation of the results.

## 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] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1983.

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

[4] Joan Boyar, Rolf Fagerberg, and Kim 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] Joan Boyar, Rolf Fagerberg, and Kim 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] Joan F. Boyar and Kim 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] Joan F. Boyar and Kim S. Larsen. Efficient Rebalancing of Chromatic Search Trees. *Journal of Computer and System Sciences*, 49(3):667–682, 1994.

[8] Leo J. Guibas and Robert 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, Th. 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.

[10] J. E. Hopcroft. Title unknown. Unpublished work on 2-3 trees, 1970.

[11] Scott Huddleston and Kurt 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] Kim S. Larsen. AVL Trees with Relaxed Balance. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 888–893. IEEE Computer Society Press, 1994.

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

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

[16] Kim S. Larsen, Thomas Ottmann, and Eljas 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.

[17] Kim S. Larsen, Eljas Soisalon-Soininen, and Peter 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.

[18] 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.

[19] Otto Nurmi and Eljas 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.

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

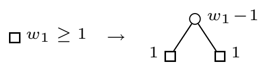
[21] Neil Sarnak and Robert E. Tarjan. Planar Point Location Using Persistent Search Trees. *Communications of the ACM*, 29:669–679, 1986.

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

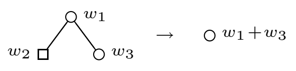
[23] Robert Endre Tarjan. Amortized Computational Complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985.

# Appendix: The Operations

## Update Operations

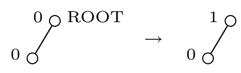


(insert)

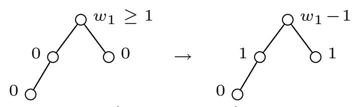


(delete)

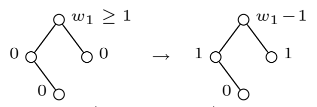
## Rebalancing Operations



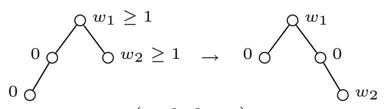
(red-root)



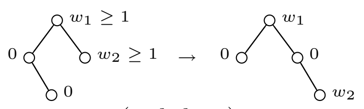
(red-push1)



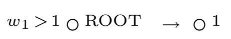
(red-push2)



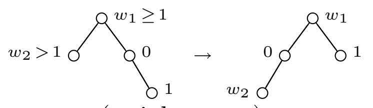
(red-dec1)



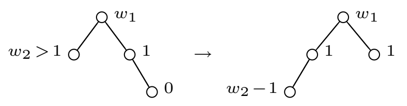
(red-dec2)



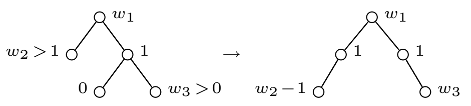
(weight-root)



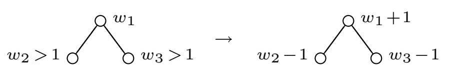
(weight-temp)



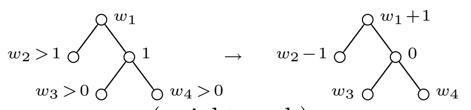
(weight-dec1)



(weight-dec2)



(weight-dec3)



(weight-push)