Alcom-FT Technical Report Series
ALCOMFT-TR-01-186

Relaxed Red-Black Trees with Group Updates*

Kim S. Larsen
University of Southern Denmark, Odense†

Abstract

In search trees with relaxed balance, updating and rebalancing have been uncoupled such that rebalancing can be controlled separately. Recently, it has been shown how an advanced update such as an insertion of an entire tree into a relaxed multi-way structure can be implemented efficiently. This indicates a similar result for binary trees by a naive interpretation of small multi-way tree nodes as binary configurations. However, this would imply that nodes must be connected by level links, which significantly deviates from the usual structural implementations of binary trees. In this paper, we show that it is possible to define binary schemes which are both natural and efficient.

Introduction

Red-black trees with relaxed balance is the name of a structure which can be viewed as a generalization of a red-black tree [5]. The term *relaxed balance* was introduced in [13] to mean a search tree where the traditional tight coupling between updates and rebalancing is removed.

---

* Supported in part by the Danish Natural Sciences Research Council (SNF) and in part by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

† Department of Mathematics and Computer Science, University of Southern Denmark, Main campus: Odense University, Campusvej 55, DK-5230 Odense M, Denmark. E-mail: kslarsen@imada.sdu.dk.

Not having the traditional restriction that rebalancing must be carried out immediately following an update gives significant extra control, which could for instance be used to delay rebalancing during load peeks. Relaxed balance also offers a possible solution to a standard concurrency control problem in search trees. If it must be possible to rebalance immediately following an update, the updating process must be allowed to rebalance on the entire search path from the location of the update back up to the root, and this significantly limits the amount of concurrency which can be allowed.

Until recently, updates were restricted to insertions and deletions, but in [11], group updates were introduced for the first time in a relaxed setting. The idea is that it might be more efficient to bring in more updates at the same time, but also that some applications may require that a whole collection of keys be made available simultaneously.

Group updates can be group deletions or group insertions. If a large number of keys are either to be deleted or inserted, some of these may be neighboring nodes, and this might make it possible to perform the entire operation more efficiently. All existing relaxed proposals can already exploit the extra possibilities which are available when carrying out neighboring deletions to do so more efficiently, so our focus is group insertion.

We study the core problem of moving m keys in between two neighbor keys in the search tree. Thus, if one considers the problem of moving m arbitrary keys in, they first have to be divided up into groups (via some search procedure). This can be done for our structure exactly as it has been done in [6, 11, 16]. Of newer possible applications for such group insertions are main-memory databases for search engines, document databases using inverted index techniques [3, 4], or other applications where a large collection of keys must or may be brought into the main index at the same time [8].

The informal model of relaxed balance which has been used in earlier papers and which will also be used here is the following. First, it must be possible to perform an update and leave the tree in a well-defined state without performing any rebalancing. Second, it must be possible to perform rebalancing in small (typically constant-sized) independent steps. Third, it must be possible to interleave update and rebalancing operations freely. This is a satisfactory model from a theoretical point of view and it has the advantage of not assuming too much such that some of the practical applications would be ruled out.

Note that there is nothing in the model which forces any rebalancing at any point. Thus, all relaxed search trees, including the one we present here, can contain paths of super-logarithmic length. Thus, the extra freedom to carry out rebalancing whenever this is convenient should of course be exercised with some care.

For the same reason, it is a challenge to prove good complexity results for these data structures. When there is no good bound on the length of the paths, and when rebalancing operations can be applied in any order, care must be taken in order not to introduce super-logarithmic rebalancing or even loops or deadlocks which could follow from negative interference between rebalancing operations.

There has been a significant amount work on relaxed balance, references to most of which can be found via [9]. Work on red-black trees with relaxed balance was initiated in [14, 15] and continued in [2, 1, 9]. Of particular relevance to the present paper is the first study of group insertion in a relaxed setting from [11] and the results obtained in [10], which show that for (a, b)-trees [12], a relaxed definition can be given which allows for amortized constant insertions and deletions and amortized logarithmic group insertions.

In [6], a variant of red-black trees is considered. Their variant is based on [1], and therefore has the slight disadvantage, at least from a practical perspective, that some of the rebalancing operations are quite large (triple rotations). They show that the group insertion of a tree of size m can be performed in time $O(\log^2 m)$, provided that the tree is red-black when the operation is performed. If used relaxed, the removal of negative weight can create a super-linear amount of overweight. Large or small weights indicate balance problems and it is hard to see how a good complexity bound could be established for the relaxed case based on the operations in that paper.

In this paper, we develop another variant based on the collection of operations from [9]. This means that the sizes of the rebalancing operations are smaller. Furthermore, the operations which do any restructuring (changes pointers) are single rotations, with the exception of one, which is a double rotation.

Our main focus is on amortized complexity, which we believe is more interesting in practice than the usual worst-case complexity. Often, for large systems, we are really interested in average complexities for some, possibly varying, distributions. However, those results are usually very difficult to obtain. Fortunately, amortized results give upper bounds on the average complexities under any distribution, and often very good ones.

To be precise, we show that insertion and deletion are amortized constant and worst-case logarithmic, and that group insertion of a tree of size m is amortized $O(\log m)$ and worst-case $O(\log^2 m)$. We also show that restructuring after insertions and deletions is worst-case constant and worst-case logarithmic after a group insertion.

A Red-Black Tree with Relaxed Balance

As always in the world of relaxed balance, we consider *leaf-oriented* trees. This means that all keys are stored in the leaves, and the internal nodes only contain so-called routers which direct the search to the correct subtree. The reason for choosing leaf-oriented trees is that otherwise a deletion cannot be performed completely locally. In general, to delete an internal node with two children, the predecessor or successor node must be found [5], and this node may be more than a constant distance away.

A red-black tree with relaxed balance is a search tree, so the usual search tree ordering invariant must be maintained. Additionally, each node is equipped with an integer weight used for rebalancing purposes. This weight can be viewed as a generalization of the red/black colors used in red-black trees. Red-black tree with relaxed balance must maintain the invariant that the sum of all weights of nodes on any path is the same.

The goal of the rebalancing process is to transform the tree towards a red-black tree. Thus, we interpret the weight zero as a red node and the weight one as a black node. Configurations in the tree which prevent it from being red-black are called *conflicts*. In particular, a node with negative weight is referred to as a *negative* conflict, a node with weight at least two is referred to as an *overweight* conflict, and two consecutive red nodes on a path are referred to as a *red* conflict. In fact, we think of a negative conflict on a node with weight smaller than -1 to consist of a number of conflicts corresponding to the numerical value of the weight, e.g., the weight -3 means that there are three negative conflicts on that node. Similarly, a node with weight three has two overweight conflicts. A red conflict involves two nodes and we decide to consider the top-most of these the location for the conflict.

Clearly, a red-black tree with relaxed balance with no conflicts fulfills the conditions of red-black trees from [5], and as a consequence, they are balanced.

A red-black tree with relaxed balance is equipped with a collection of update operations and rebalancing operations, depicted in the appendix. The operations are divided in two. The first collection is the operations from [9] which deal with insertion and deletion. The second gives the extra operations for handling group insertions.

The operations must of course preserve the tree as a search tree. However, provided that updates are performed correctly, this follows immediately since the operations either do not perform any restructuring or perform a single or double rotation which are known from any textbook on the subject to preserve the ordering invariant. The standard rotations also define where subtrees from before an operation is carried out should be attached afterwards. However, this can also be said more generally: For any transformation on a binary search tree which preserves the number of nodes, if the subtrees and routers from before the transformation is carried out are removed in-order and again attached in-order after the transformation, then the tree is still a search tree.

Furthermore, the operations should preserve the tree as a red-black tree with relaxed balance, i.e., they should maintain the invariant that the sum of weights of nodes on any path is the same. This can be, and has been, verified by checking all possible different paths down through the transformations.

The conditions for when the different operations can be applied are written next to the nodes. For group insertion, the value h, is the weight of a path from the root to a leaf in the tree which is inserted, excluding the weight of the root. We will refer to this as the black height of the tree, even though in the standard definition [5], the weight of the root is included.

For the insertion operation in the appendix, it is a requirement that the weight of the leaf is at least one before the operation. It is easy to verify by inspection of the operations that no operation can decrease the weight of a leaf below one, so an insertion is always possible.

The collection of operations in the appendix show one of two symmetric variants, i.e., to make the set complete, we should for each operation include the symmetric variant which can be created by reflecting about a line down through the root of the operation. All necessary deeper symmetries, such as *red-push1* and *red-push2*, are included directly.

Complexity Analysis

We are now ready to prove all complexity results described in the introduction. Our data structure is a pure generalization of the one from [9], i.e., the data structure and all the complexity results in [9] form a special case of what we show here, and we can use a similar organization of the proofs. However, the harder proofs from [9] become significantly more difficult when negative weight can be introduced, and even the smaller lemmas need new proofs.

Complete Collection of Rebalancing Operations

First, we prove that the collection of rebalancing operations is complete, i.e., if there is conflict somewhere in the tree, then it is possible to apply some rebalancing operation.

Compared with [9], the proof becomes more involved. The technique in [9] was to consider top-most red conflicts and, if none of those were present, bottom-most weight conflicts. Because of interference from the negative weights, this approach is no longer possible. Since negative weights cannot be removed unconditionally without considering conflicts in its surroundings, we have to assume the presence of all conflicts simultaneously, and overweight conflicts also have to be considered top-down.

To be precise, a top-most conflict means that there is no other conflict at a distance closer to the root.

Theorem 1 If a red-black tree with relaxed balance is not red-black, then one of the rebalancing operations can be applied.

Proof Assume that a red-black tree with relaxed balance is not red-black. Thus, there is a conflict in the tree. Consider a top-most of these. If there is a choice between the three types of conflicts, we choose a negative weight conflict if possible. Otherwise, we choose a red conflict over an overweight conflict,

Assume first that this top-most conflict is a negative weight conflict. If it is located at the root or immediately below, then *neg-root1* or *neg-root2* can be applied. We may now assume that the node u with negative weight has at least two ancestors, both of which are non-negative.

Consider the parent p of u. If it is positive, then *neg-push1* can be applied. If it is zero, we consider p's parent g. If g's weight is positive, then *neg-push2* or *neg-push3* can be applied Otherwise, its weight must be zero. However, this is not possible, because then g and p form a red conflict, which means that the negative weight conflict under consideration would not be top-most.

Now we assume that the top-most conflict is a red conflict. If it is located at the root, then *red-root* can be applied. Otherwise, the node u at which the conflict is located must have a parent p, and because we are considering a top-most conflict, the weight of p must be one. Furthermore, because we decided to choose a negative weight conflict as the top-most conflict to consider, if at all possible, we may assume that if there are any negative weight conflicts in the tree, then they are located at a level strictly below u. Thus, the weight of the sibling of u must be non-negative. If this sibling has weight zero, either *red-push1* or *red-push2* can be applied. If its weight is positive, then *red-dec1* or *red-dec2* can be applied.

Finally, we assume that the top-most conflict is an overweight. If it is located at the root, then *weight-root* can be applied. Otherwise, the overweighted node u has a sibling v. By the priority of the conflicts in the choice of a top-most conflict, v cannot have negative weight.

Assume first that the weight of v is positive. If the weight of v is at least two, then *weight-dec3* can be applied, so assume that the weight of v is one. If any of the children of v have negative weight, then *neg-push1* can be applied. Thus, we can assume that both children of v have non-negative weights. If the outer-most child of v has weight zero, then *weight-dec1* can be applied. If the weight of that child is positive, then, depending on whether the other child of v has weight zero or positive weight, either *weight-dec2* or *weight-push* can be applied.

As the last case, assume that the weight of v is zero. By the priorities in the choice of a top-most conflict, the children of v cannot have weight zero and the weight of the parent of u and v must be one. Now, if any of the children of v have negative weights, then *neg-push2* or *neg-push3* can be applied. So, we can assume that they have positive weights.

If the outer-most child of v has weight one, then *weight-temp* can be applied, so assume that it has weight at least two. To show that some operation can be carried out also in this case, we now focus on this child x of v. Let us recall that x is overweighted, and it has a sibling y with positive weight. If y is overweighted, then *weight-dec3* can be applied, so assume that the weight if y is one. If either of y's children have negative weight, then *neg-push1* can be applied. Thus, we can assume that both of y's children have non-negative weights. As above, if the outer-most child of y has weight zero, then *weight-dec1* can be applied. If the weight of that child is positive, then, depending on whether the other child of y has weight zero or positive weight, either *weight-dec2* or *weight-push* can be applied. □

We have proven that as long as the tree is not red-black, some rebalancing operation can be applied. In that analysis, we have focused on a top-most conflict, but this does of course not imply that only the top-most conflict can be addressed. In fact, any conflict, the surroundings of which are red-black can be dealt with, and, as can be seen from the operations, quite often a conflict can be addressed in the presence of others.

Amortized Complexity of Update Operations

In this section, we use the potential function technique for proving the amortized results [17]. We start by making some observations regarding the behavior of the different operations in relation to conflicts in the tree.

The following observations are easily verified by inspection of the operations in the appendix, and we have of course carried out this procedure.

Observation 1 The following statements hold:

- *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.
- *neg-root1*, *neg-root2*, *neg-push1*, *neg-push2*, and *neg-push3* decrease the total amount of negative weight in the tree.
- An *insert* increases the number of red conflicts by at most one, a *delete* increases the amount of overweight by at most one, and a *group-insert*

increases the number of red conflicts by at most one and creates at most $h-1$ units of negative weight.

- No rebalancing operation increases the number of negative weight units in the tree.
- No rebalancing operation, except *neg-root2*, *neg-push1*, *neg-push2*, and *neg-push3*, increases the number of red conflicts nor the number of weight conflicts in the tree. The operations *neg-root2*, *neg-push1*, *neg-push2*, and *neg-push3* increase the number of red conflicts and overweight conflicts by at most a constant.

$\square$

We want to establish a proof of the fact that insertions and deletions are amortized constant and that group insertion is amortized logarithmic in the height of the inserted tree. Since group insertion introduces negative weight proportional to the height of the inserted tree, this can be reformulated. By recalling that each unit of negative weight and each unit of overweight is considered to be a conflict, we want to show that each conflict which is introduced is removed in amortized constant time.

One problem in establishing the proof is the new operations *neg-root2*, *neg-push1*, *neg-push2*, and *neg-push3*, which may create new conflicts. It turns out that defining an ordering of conflicts, such that it is considered better to have an overweight conflict than a negative weight conflict, can help here.

Another problem in establishing the proof is to show that progress is made also when operations which do not actually remove a problem are carried out. The operation *weight-temp* is one such operation, but *red-push1*, *red-push2*, and *weight-push* are similar, in that they do remove a problem, but the may introduce a problem of the same type further up in the tree. To obtain the proof, it is necessary to show that carrying out these operations also represent progress.

One way to approach this is by identifying patterns which enable the application of these operations. If such a pattern is removed, it represents progress, since an operation which requires this pattern cannot be applied there again until such a pattern has been created again.

The following patterns were also identified in [9]:

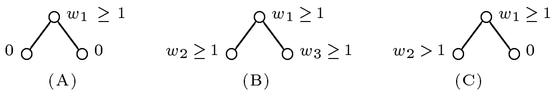


Figure 1: Potential types A, B, and C.

Definition 1 The three patterns displayed in Fig. 1 are referred to as *potential types*. As for all other operations in this paper, we only display one of each symmetric configuration, so the symmetric variant of (C) is also a configuration of potential type (C). □

A relation between actions of the operations which do not remove conflicts and the total collection of potential types in the tree is established in the following lemma, which is a slight modification of a similar lemma in [9].

Lemma 1 The following statements hold:

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).
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), and does not increase the number of potential types (A).

Proof We prove the three parts separately.

1. The proofs for *red-push1* and *red-push2* are very similar, and we just give the proof for the former. If the number of red conflicts is not reduced by the operation, then a red conflict must be created right above the current. Thus, $w_1 = 1$; see Fig. 2. Similarly, the number of red conflicts would be reduced if any of the q_i's were zero. Clearly, the number of potential types (A) is reduced by one.
2. Easy inspection.

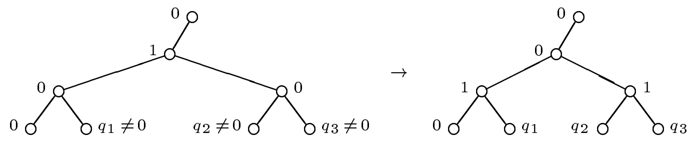


Figure 2: Red-push1 when the number of red conflicts is not reduced.

3. If *weight-push* does not decrease the total amount of overweight, then we must have that $w_1 \geq 1$. Thus, the operation cannot create siblings with weight zero, so no configuration of potential type (A) is created. Two configurations of potential type (B) are removed: one rooted at the node labelled w_1 and one rooted at the node labelled 1. No new configurations of potential type (B) can be created.

□

As a final lemma before the main result, we show that we can build red-black trees which do not contain many potential types. This is important since they will be used to define potential, and we do not want a *group-insert* to increase the potential too much. The lemma can be derived from a corollary in [7] followed by a transformation from $(2,4)$-trees [12] to red-black trees, but for completeness, we give a direct proof.

Lemma 2 Given m keys, a red-black tree can be build such that the number of potential types (A), (B), and (C) in the tree is $O(\log m)$.

Proof Start by arranging the m keys in a list of m leaves in sorted order. Recursively build layers bottom-up. A layer is build as follows. From left to right, while there is at least five nodes left, consider three nodes at a time, and join these using one red node to join two neighboring nodes and one black node to join in the remaining node. The last three, four, or five nodes are treated as follows. If the remaining number is three or five, we continue as before one more time. Depending on which case we were in, there are now either zero, two, or four nodes left. These are connected using either zero, one, or two configurations of potential type (A). In this way, at most two potential types are used per layer. □

Theorem 2 When starting from an empty tree, the number of rebalancing operations is amortized constant in response to an insertion or deletion and amortized $O(\log m)$ in response to a group insertion of a red-black tree of size m.

Proof We now define an ordering on the various types of conflicts and potential types in the tree. The ordering, which we refer to as our *abstract problem list*, is

negative weight, overweight, red conflicts, (A), (B), and (C).

Observation 1 and Lemma 1 together establish that for every rebalancing operation, there is a problem in the abstract problem list, the total number of which is reduced when the operation is carried out. Additionally, if the rebalancing operation increases the total number of some problem, then that problem is further down the abstract problem list compared to the problem type which is decreased.

For all rebalancing operations, weights are only increased or decreased by one, and no operation involves more than a constant number of nodes. This means that any operation which creates problems from our problem list can create at most a constant number of these problems.

As a result, we can define a potential function as a weighted (the meaning of the word *weight* here should not be confused with the weights appearing in the tree) sum of the number of problems of each type in the abstract problems list, giving higher weight to problems early in the list. Since each rebalancing operation can create only a constant number of other problems, the weights of the weighted sum can also be constants, and they can be chosen such that every rebalancing operation decreases the potential by at least one.

Since *insert* and *delete* involve only a constant number of nodes and only increase or decrease weights in the tree by at most one, they can only increase the potential by a constant.

A *group-insert* of a tree of size m can be performed such that the increase in potential is at most $O(\log m)$. This follows from Lemma 2 and from the fact that the black height of a red-black tree of size m is $\Theta(\log m)$ [5], which implies that at most $O(\log m)$ negative weight is introduced. $\square$

Worst-Case Complexity of Update Operations

If we consider starting with an initially non-empty red-black tree, what are the complexities of the operations? Amortized results generally assume an initially empty structure in which potential can be build up gradually. Since a red-black tree can only contain a linear amount of potential, by distributing the cost out over a sequence of operations, it is clear that when $\Omega(n)$ operations have been performed on a red-black tree of initial size n, all operations have the amortized complexities shown in Theorem 2. Until that happens, the results proven in this section applies.

We introduce an accounting scheme inspired by [9], but more complicated. The overall idea is to establish a connection between the number of nodes in a tree and its weighted height, where the weighted height of a node is defined to be the sum of all weights from that node down to a leaf.

We define a "counting" function c from the set of nodes in the tree at a given point in time to the real numbers. The sum of all the function values will have the property that it equals the total number of nodes which have been in the tree since it was last red-black.

We now give the rules for updating c when an operation is applied. When an *insert* is made, the two new nodes are given function value one. Similarly, when a *group-insert* is made, all new nodes get function value one. When a *delete* is made, the total sum of function values for the three nodes from before the operation is carried out is the new value of the remaining node.

Other operations only move values around. Since values follow the nodes, we have to define which nodes after an operation is carried out correspond to which nodes before. For operations which make no structural changes, the identification is given by location. For the remaining operations, the root of a configuration before the operation is carried out is identified with the root after the configuration. For the remaining nodes, the identification is made by the order of the nodes in an in-order traversal of the configurations, i.e., completely skipping the root of the configuration, the ith node encountered in an in-order traversal of the configuration before the operation is carried out is identified with the ith node encountered in an in-order traversal of the configuration after the operation is carried out.

Finally, when operations for negative weight are applied, function value is taken from the subtree of the negatively weighted node and given to the other at most two nodes which have their weights increased. Note that if there are two such nodes, they have the same weighted height w after the operation. Each of these at most two nodes receive the value $\frac{1}{9}(2^w - 1)$. The amount which is taken away from the subtree with negative root is collected in such a manner that all original nodes in the subtree after the operation have the same function value. By original nodes, we mean the nodes which were inserted at the time that the negative weight was created. It will be an invariant that these nodes always have the same function value, so this requirement can always be fulfilled again.

We let T_u denote the set of nodes in the subtree rooted by u.

Lemma 3 If u is a node with non-negative weight and weighted height w, then $\sum_{v \in T_u} c(v) \geq \frac{1}{9}(2^w - 1)$.

Proof The proof is by induction in the number of operations performed on the tree. The base case is when no operations have been performed. Thus, we have a red-black tree. The base case is proven by a simple structural induction. Clearly it holds for the leaves, which initially have weight one, and if it holds for two subtrees of a node u, then u's weighted height w is at most one more than its childrens', and, by induction, its function value is at least $\frac{1}{9}(2^{w-1} - 1) + \frac{1}{9}(2^{w-1} - 1) + 1 \geq \frac{1}{9}(2^w - 1)$.

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

For *insert*, the new node has weighted height one and has function value one. No other nodes have their weighted heights changed or function values decreased. For *delete*, the weighted height as well as the subtree sum are unchanged for the nodes that remain in the tree after the operation. For *group-insert*, the inserted tree is red-black, so the results hold internally in that tree, just as in the base case. We do not have to show anything for the negatively weighted node, and no other nodes have their weighted heights changed or function values decreased.

Now we consider the operations for negative weight. Assume that u is the node with negative weight. Clearly, by the scheme we have outlined, the other nodes, which have one added to their weight, will have function values large enough. However, at some point, the weight of u is going to be changed from -1 to zero, and at that point, it must have a sufficiently large subtree sum.

At the time of the insertion, u has weighted height one, since the black height of the inserted tree is h and u has weight $-h+1$. Since it is a red-black tree of black height h, its subtree sum is at least $2^h - 1$. Whenever some function value is taken from the subtree of u and given to other nodes, at most two nodes receive. The recipient nodes have weighted height identical to u's at the time. During the operation, the weight of u increases, and no operation ever decreases the weight of a negatively weighted node. First time, the weighted height of the involved nodes is two, and last time, it is $h+1$, referring to the weighted heights after each operation has been carried out. Thus, in total, u can give away at most:

$$\sum_{j=2}^{h+1} 2\tfrac{1}{9}(2^j - 1) \leq \tfrac{8}{9}2^h - 1$$

So, there is at least

$$2^h - 1 - (\tfrac{8}{9}2^h - 1) \geq \tfrac{1}{9}(2^h - 1)$$

left for u at the time it is needed.

Since the nodes of this subtree cannot again have a negatively weighted ancestor, no more function value will be taken away from these nodes.

General arguments cover all remaining cases:

First, the root of an operation will always have the same weighted height before and after the operation, unless it is the root of the entire tree, since otherwise it would violate the weighted height constraint. Since the root of an operation never has its function value altered, the result holds for such nodes.

Second, nodes which after the operation have weight zero or one and which have subtrees that are unchanged by the operation or have already been established to have large enough function values will themselves have large enough function values. This follows since the weighted height w of such nodes is at most one more than their childrens' and since their own function value of at least $\frac{1}{9}$ results in the sum $\frac{1}{9}(2^{w-1}-1)+\frac{1}{9}(2^{w-1}-1)+\frac{1}{9} = \frac{1}{9}(2^w-1)$.

Third, for nodes which are leaves of an operation and keep their subtree from before the operation is carried out, the result holds provided that their weight is not increased.

By these arguments, all nodes in the operations have been covered. □

Corollary 1 Let n' denote the total number of nodes inserted into the tree since it was a red-black tree of size n. Then the largest weighted height of a node which is not in the subtree of a negatively weighted node is bounded by $\lfloor \log(9(n + n') + 1) \rfloor$.

Proof By definition of c, the sum of all function value in the tree is $n + n'$. Assume that the weight of the root is non-negative. Then no other node in the tree can have a larger weighted height than the root, unless it is in the subtree of a negatively weighted node.

By Lemma 3, if the weighted height of the root is w, then $n+n' \geq \frac{1}{9}(2^w - 1)$, so $w \leq \lfloor \log(9(n + n') + 1) \rfloor$. □

The upper bound on the largest possible weighted height can be used to bound the number of times most of the operations can be applied. This is because the operations have been designed with the aim of moving conflicts to larger weighted heights, if they cannot be removed completely in the given configuration.

By *moving a conflict*, we refer to the scenario where an operation removes a certain conflict, only to introduce a new one of the same type further up in the tree. The operation *red-push1*, for instance, may do that if $w_1 = 1$ and the parent of that node is red.

As already described, overweighted nodes with weights larger than two as well as negatively weighted nodes with weights smaller than -1 are considered as a number of unit conflicts. So, if a node has weight $w_1 > 1$ and weighted height w, this is interpreted as $w_1 - 1$ weight conflicts of weighted height $w - w_1 + 2, w - w_1 + 3, \ldots, w$, respectively. 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 following statements hold:

- *red-root*, *red-dec1*, and *red-dec2* remove at least one red conflict.
- *weight-root*, *weight-dec1*, *weight-dec2*, and *weight-dec3* remove at least one unit of overweight.
- *red-push1*, *red-push2*, and *weight-push* either remove a conflict or move a conflict to a larger weighted height.

- All the operations leave all other conflicts at the same weighted height as before the operation was carried out.
- Only *neg-root2*, *neg-push1*, *neg-push2*, and *neg-push3* create new conflicts, which is at most two red conflicts and at most two units of overweight.

Proof Most follow by easy inspection of the operations in the appendix.

For the scenario where *red-push1* (similarly for *red-push2*) moves a conflict, refer to Fig. 2.

Note that if *weight-push* moves a conflict, then $w_1 \geq 1$. □

The proof is build up around the movement of conflicts to larger weighted heights. Unfortunately, since *weight-temp* does not accomplish this, an alternative method of bounding the application of this operation must be found.

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 u.

Lemma 4 Weight-temp configurations can only disappear through the application of an operation which decreases the total number of conflicts in the tree or which decreases the number of negative weight conflicts.

Proof Clearly, in order to change the configuration, an operation must overlap with nodes in the configuration. Let u be the overweighted node in the configuration (the one with a red parent).

The operation *insert* cannot make the configuration disappear (note that if the uncle of u has weight one, then it cannot be a leaf). Similarly for *group-insert*. If a *delete* changes the situation, it is because $w_1 = 0$ and $w_3 > 1$ (within the *delete* 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 or the number of negatively weighted nodes, 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 the *weight-push* operation. In that case, the uncle of u has its weight increased, so it will still be non-red. □

Corollary 2 If i *insert* operations, d *delete* operations, and g *group-insert* operations of black heights $h_1, \ldots, h_g$ are made into a red-black tree, at most $i + d + 5\sum_{j=1}^{g} h_j$ *weight-temp* operations can be applied.

Proof By Observation 1, only the rebalancing operations for negative weight increase the number of conflicts. The operations *insert* and *delete* create at most one conflict each time they are applied. As it appears from the proof of Lemma 3, the negative operations can introduce at most $4\sum_{j=1}^{g} h_j$ red conflicts or units of overweight. Every time a conflict is created this way, a unit of negative weight disappears. Thus, the total number of non-negative conflicts ever introduced in the tree is bounded by $i + d + 4\sum_{j=1}^{g} h_j$.

By Lemma 4, a weight-temp configuration can only be removed through the application of an operation which either decreases the total number of conflicts in the tree or removes negative weight. Thus, weight-temp configurations can be removed at most $i+d+4\sum_{j=1}^{g} h_j + \sum_{j=1}^{g} h_j = i+d+5\sum_{j=1}^{g} h_j$ times.

By Theorems 1 and 2, the tree will eventually become red-black. Since a weight-temp configuration contains overweight, every weight-temp configuration will eventually be removed. Thus, at most $i+d+5\sum_{j=1}^{g} h_j$ 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 Assume that i *insert* operations, d *delete* operations, and g *group-insert* operations of trees of sizes $n_1, \ldots, n_g$ are made into a red-black tree of size n. Let $N = n+i+\sum_{j=1}^{g} n_j$. Then at most $O(N \log N)$ rebalancing operations are carried out.

Proof Assume that the inserted trees have black heights $h_1, \ldots, h_g$. By Proposition 1, at most $\sum_{j=1}^{g} h_j$ operations for negative weight can be applied, and these can give rise to at most $4\sum_{j=1}^{g} h_j$ other conflicts.

As it appears from Corollary 2, $M = i+d+5\sum_{j=1}^{g} h_j$ is an upper bound on the number of conflicts ever introduced. This immediately bounds the number of operations which remove conflicts by M. Corollary 2 gives a bound on the number of *weight-temp* operations which can be applied, which is at most M.

By Proposition 1, all other operations move conflicts to a larger weighted height, so by Corollary 1, at most $M\lfloor \log(9(n + 2(i + \sum_{j=1}^{g} n_j)) + 1)\rfloor$ such operations can be carried out.

The result follows since $h_j \in O(\log n_j)$ and $\lfloor \log(9(n+2(i+\sum_{j=1}^{g} n_j))+1)\rfloor \in O(\log N)$. $\square$

Worst-Case Restructuring Complexity

Finally, we prove a bound on the number of operations which actually change the structure of the tree. The reason for singling these out is that they are generally more expensive, and in parallel applications they require exclusive locking [1, 14].

Theorem 4 Assume that i *insert* operations, d *delete* operations, and g *group-insert* operations of trees of sizes $n_1, \ldots, n_g$ are made into a red-black tree of size n. Then at most $O(i + d + \sum_{j=1}^{g} \log n_j)$ restructuring rebalancing operations are carried out.

Proof Only *weight-temp* operations and operations which decrease the number of conflicts do any restructuring. By the proof of Theorem 3, this amounts to $2M$ operations from which the result follows. $\square$

Concluding Remarks

We have defined a collection of operations for which it is possible to prove all the good complexity bounds one could hope for. However, adjustments are still possible. Sometimes it is possible to push more than one unit of weight at a time, and this could be allowed by the operations. One could also consider creating positive interference between different conflicts such that more frequently more than one conflict can be handled at a time. It is also possible to merge operations; the *weight-temp* operation, for instance, can be merged with the weight decreasing operations to get a collection of fewer, but larger, rebalancing operations.

With the proofs in this paper, it is often quite easy to verify, by checking the proofs, whether or not a desired change in the collection of rebalancing operations will give a new collection which also has good complexity bounds.

References

[1] 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.

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

[3] Alfonso F. Cardenas. Analysis and Performance of Inverted Data Base Structures. *Communications of the ACM*, 18(5):253–263, 1975.

[4] Christos Faloutsos and H. V. Jagadish. Hybrid Index Organizations for Text Databases. In *Third International Conference on Extending Database Technology*, volume 580 of *Lecture Notes in Computer Science*, pages 310–327, 1992.

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

[6] Sabina Hanke and Eljas Soisalon-Soininen. Group Updates for Red-Black Trees. In *4th Italian Conference on Algorithms and Complexity*, volume 1767 of *Lecture Notes in Computer Science*, pages 253–262. Springer-Verlag, 2000.

[7] Lars Jacobsen, Kim S. Larsen, and Morten N. Nielsen. On the Existence and Construction of Non-Extreme (a,b)-Trees. Tech. report 11, Department of Mathematics and Computer Science, University of Southern Denmark, Odense, 2001.

[8] Sheau-Dong Lang, James R. Driscoll, and Jiann H. Jou. Batch Insertion for Tree Structured File Organizations—Improving Differential Database Representation. *Information Systems*, 11(2):167–175, 1986.

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

[10] Kim S. Larsen. Relaxed Multi-Way Trees with Group Updates. In *Twentieth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 93–101. ACM Press, 2001.

[11] Lauri Malmi and Eljas Soisalon-Soininen. Group Updates for Relaxed Height-Balanced Trees. In *Eighteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 358–367. ACM Press, 1999.

[12] Kurt Mehlhorn. *Sorting and Searching*, volume 1 of *Data Structures and Algorithms*. Springer-Verlag, 1984.

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

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

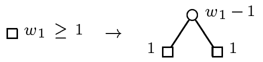
[15] Otto Nurmi and Eljas Soisalon-Soininen. Chromatic Binary Search Trees—A Structure for Concurrent Rebalancing. *Acta Informatica*, 33(6):547–557, 1996.

[16] Kerttu Pollari-Malmi, Eljas Soisalon-Soininen, and Tatu Ylönen. Concurrency Control in B-Trees with Batch Updates. *IEEE Transactions on Knowledge and Data Engineering*, 8(6):975–984, 1996.

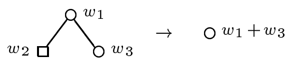
[17] Robert Endre Tarjan. Amortized Computational Complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985.

Appendix: The Operations

Update Operations

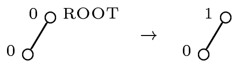


(insert)

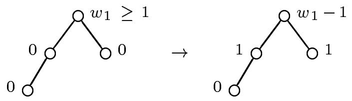


(delete)

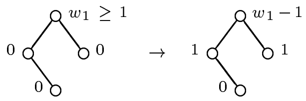
Rebalancing Operations



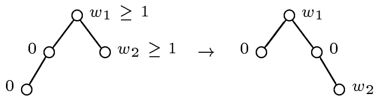
(red-root)



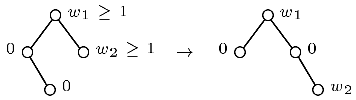
(red-push1)



(red-push2)

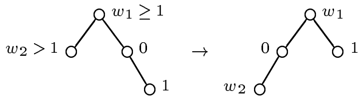


(red-dec1)

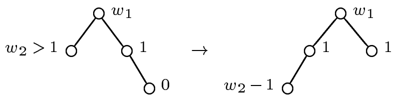


(red-dec2)

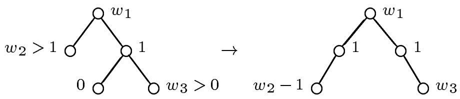
(weight-root)



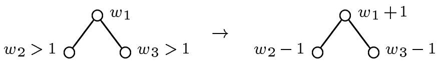
(weight-temp)



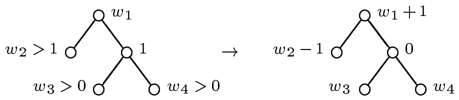
(weight-dec1)



(weight-dec2)

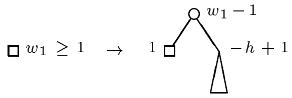


(weight-dec3)



(weight-push)

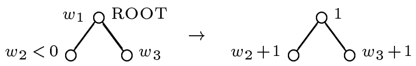
Group Insertion



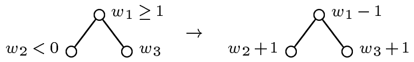
(group-insert)

Rebalancing after Group Insertion

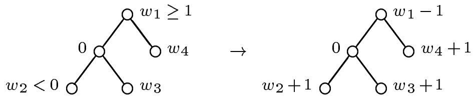
(neg-root1)



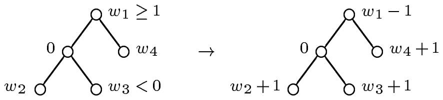
(neg-root2)



(neg-push1)



(neg-push2)



(neg-push3)