Efficient Rebalancing of Chromatic Search Trees

Joan Boyar [*]

Dept. of Math. and Computer Science, Odense University, Denmark

Kim S. Larsen [†]

Computer Science Department, Aarhus University, Denmark

Abstract

Chromatic trees were defined in [8] by Nurmi and Soisalon-Soininen, as a new type of binary search tree for databases. The aim is to improve runtime performance by allowing a greater degree of concurrency, which, in turn, is obtained by uncoupling updating from rebalancing. This also allows rebalancing to be postponed completely or partially until after peak working hours.

The advantages of the proposal of Nurmi and Soisalon-Soininen are quite significant, but there are definite problems with it. First, they give no explicit upper bound on the complexity of their algorithm. Second, some of their rebalancing operations can be applied many more times than necessary. Third, some of their operations, when removing one problem, create another.

We define a new set of rebalancing operations which we prove give rise to at most $\lfloor \log_2(N+1) \rfloor - 1$ rebalancing operations per insertion and at most $\lfloor \log_2(N+1) \rfloor - 2$ rebalancing operations per deletion, where N is the maximum size the tree could ever have, given

[*] Most of this work was done while the first author was on leave from Loyola University Chicago, and some was done while she was visiting Aarhus University.

[†] Present address: Department of Mathematics and Computer Science, Odense University, Campusvej 55, 5230 Odense M, Denmark

its initial size and the number of insertions performed. Most of these rebalancing operations, in fact, do no restructuring; they simply move weights around. The number of operations which actually change the structure of the tree is at most one per update.

1 Introduction

In [8], Nurmi and Soisalon-Soininen considered the problem of fast execution of updates in relations which are laid out as dictionaries in a concurrent environment. A *dictionary* is a data structure which supports the operations *search*, *insert*, and *delete*. Since both insertion and deletion modify the data structure, they are called the *updating operations*. For an implementation of a dictionary, Nurmi and Soisalon-Soininen propose a new type of binary search tree, which they call a *chromatic* tree.

One standard implementation of a dictionary is as a *red-black* tree [4], which is a type of balanced binary search tree. However, often when a data structure is accessed and updated by different processes in a concurrent environment, parts of the structure have to be *locked* while data items are changed or deleted. In the case of red-black trees of size n, an update requires locking $O(\log_2(n))$ nodes, though not necessarily simultaneously [4], in order to rebalance the tree. No other users can access the subtree below a node which is locked. Since the root is often one of the nodes locked, this greatly limits the amount of concurrency possible.

This leads Nurmi and Soisalon-Soininen to consider a very interesting idea for making the concurrent use of binary search trees more efficient: uncouple the updating (insertion and deletion) from the rebalancing operations, so that updating becomes much faster. The rebalancing can then be done by a background process, or it can be delayed until after peak working hours. Nurmi and Soisalon-Soininen call this new data structure a *chromatic tree*. Another important property of this data structure is that each of the updating and rebalancing operations can be performed by locking only a small, constant number of nodes, so considerable parallelism is possible. In [8], one locking scheme is described in great detail. Other possibilities that help avoid some of the locking are described in [6].

The idea of uncoupling the updating from the rebalancing operations was first proposed in [4], and has been studied in connection with AVL trees [1] in [5, 9]. This idea has also been studied, to some extend, in connection with B-trees [2]. A summary of this, along with references, can be found in [6].

A completely different approach to the problem of implementing dictionaries in a concurrent environment is proposed by Pugh in [11]. The idea is to use skip lists [10], a probabilistic data structure, which is fundamentally different from the standard balanced binary search tree approaches. There is no guarantee that a skip list is balanced or even that it will become balanced at some point after updating has stopped. However, this data structure exhibits a very fine average performance. In addition, when using skip lists, the risk of a very bad performance for any particular search is really quite insignificant. The main disadvantage in using skip lists is that each element uses a variable amount of space. This is a problem in main memory, and even more serious in designing an efficient storage plan for secondary memory. Though the data structure is definitely intended for main memory use, reasonable performance is required if the data structure, or parts of it, must be placed (temporarily) on secondary storage. One can, of course, allocate maximum space for all elements, but then thirty-two extra words have to be allocated per element (using the constants suggested by Pugh). In [11], Pugh conjectured

> It might be possible to design concurrent balanced tree algorithms that allowed $O(n)$ busy writers with high efficiency, but the complexity of such algorithms probably would make their implementation prohibitive.

We believe that in this paper, a preliminary version of which appeared in [3], we prove that conjecture false. Our operations have been implemented by Malmi [7], though only for use in a sequential algorithm.

In this paper, we consider chromatic trees and propose a new set of rebalancing operations which leads to significantly more efficient rebalancing. Suppose the original search tree, T, has $|T|$ nodes before k insertions and s 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). In [8], it is shown that if their rebalancing operations are applied, then the tree will eventually become balanced. In contrast, with these new operations, the tree will become rebalanced after at most $k(\lfloor \log_2(N+1) \rfloor - 1) + s(\lfloor \log_2(N+1) \rfloor - 2) = (k+s)(\lfloor \log_2(N+1) \rfloor - 1) - s$, rebalancing operations which is $O(\log_2(N))$ for each update. In any balanced binary search tree with N nodes, a single insertion or deletion would require $\Theta(\log_2(N))$ steps in the worst case simply to access the item, so the rebalancing is also efficient. Most of these rebalancing operations, however, do no restructuring; they simply move weights around. The total number of operations which actually change the structure of the tree is at most equal to the number of updates. Since it is only when the actual structure of the tree is being changed that a user who is searching in the tree should be prevented from accessing certain nodes, this should allow a considerable degree of concurrency.

2 Chromatic Trees

The definition used in [8] for a chromatic tree is a modification of the definition in [4] for red-black trees. In this section, we give both of those definitions. The binary search 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 v is greater than or equal to any key in the left subtree and less 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.

Each edge e in the tree has an associated nonnegative integer weight $w(e)$. If $w(e) = 0$, we call the edge *red*; if $w(e) = 1$, we say the edge is *black*; and if $w(e) > 1$, we say the edge is *overweighted.* The *weight* of a path is the sum of the weights on its edges, and the *weighted level* of a node is the weight of the path from the root to that node. The weighted level of the root is zero.

Definition 2.1 A full binary search tree T with the following balance conditions is a *red-black* tree:

B1: The parent edges of T's leaves are black.

B2: All leaves of T have the same weighted level.

B3: No path from T's root to a leaf contains two consecutive red edges.

B4: T has only red and black edges. □

The definition of a chromatic tree is merely a relaxation of the balance conditions.

Definition 2.2 A full binary search tree T with the following conditions is a *chromatic tree*.

C1: The parent edges of T's leaves are not red.

C2: All leaves of T have the same weighted level. □

Insertion and deletion are the updates allowed in chromatic trees (and dictionaries in general). As for search trees in general, the operations are carried out by first searching for the element to be deleted or for the right place to insert a new element, after which the actual operation is performed. How this is done can be seen in the appendix. The lower case letters are names for the edges, and the upper case letters are names for the nodes. The other labels are weights. We do not list symmetric cases.

In ordinary balanced search trees, rebalancing is performed at the time the update occurs, moving from the leaf in question towards the root or in the opposite direction. In a chromatic tree, the data structure is left as it is after an update and rebalancing is taken care of later by other processes. The advantages of this are faster updates and more parallelism in the rebalancing process.

The maximum depth of any node in a red-black tree is $O(\log_2(n))$, but a chromatic tree could be very unbalanced. We follow [8] in assuming that initially the search tree is a red-black 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 a chromatic tree, and after enough rebalancing operations, it should again be a red-black tree.

A chromatic tree can have two types of problems which prevent it from being a red-black tree. First, it could have two consecutive red edges on some root-to-leaf path; we call this a *red-red conflict.* Second, there could be some overweighted edges; we call the sum $\sum_e \max(0, w(e) - 1)$ the amount of overweight in the tree. These two problems are easily identified when they are created (for this purpose, it is necessary for each node to have a parent pointer, in addition to the left and right child pointers). When a problem is identified, a pointer to the top-most node involved is placed in a queue for the rebalancing processes. We will ignore the problem of maintaining this queue in a concurrent environment and ask the question: "How many rebalancing operations are necessary?"

The proposal in [8] is quite successful in uncoupling the updating from the rebalancing operations and in making the updates themselves fast. The problem is that if the tree is large and is updated extensively, the number of rebalancing operations that might be applied before the tree is red-black again could be very large. The only bound they give on this number of operations is that it will be finite, but using their proof of termination, one can prove a specific finite bound of $O(sN^2)$, where s is the number of deletions and N is the maximum size the tree could ever have. It seems hard to obtain a better bound because their operations, when removing overweight, can create new red-red conflicts.

3 New Rebalancing Operations

The new rebalancing operations are shown in the appendix. The seven weight decreasing operations are referred to as (w1) through (w7). The order in which operations are applied is unrestricted, except that an operation cannot be applied if the conditions shown are not met. When an operation which alters the actual structure of the tree occurs, all of the nodes being changed must be locked. With the other operations, the weights being changed must be locked, but users *searching* in the tree could still access (pass through) those nodes.

The rebalancing operations alter chromatic search trees in a well-defined way. It is clear how the subtrees not shown should be attached, since there are the same number of places for subtrees before and after any given operation, and the order of the subtrees must be preserved in a search tree. For example, consider the sixth weight decreasing operation. The subtree which was below edge b should remain below edge b, and the subtree below edge e should remain below edge e. In addition, the left subtree below edge d should become the right subtree of the new edge c, and the right subtree below edge d should become the left subtree below the new edge d.

It is also easy to update the routers in the nodes involved in an operation. When an insertion occurs, the router in the new internal node should be given the value of the key in its left child. (Insertions will only be made to the right of an existing leaf, when the new key is the largest in the tree, or when the new key is less than or equal to a key which has been deleted, even though routers to it have not. Thus there will be no problem with other routers.) When a deletion occurs, no routers need to be updated. Consider the nodes involved in any of the rebalancing operations and the routers in these nodes. We can use the same routers after the operation, since there are the same number of nodes before and after the operation, and since the same subtrees are present below where the operation occurs. Before the operation occurs, an in-order traversal of the nodes in the section of the tree to be modified gives an ordered list of these routers. This ordered list of routers can simply be written onto the nodes of the modified section using an in-order traversal.

Note that, even though this is not shown in the appendix, all operations, except the first four weight decreasing operations, are applicable when the edge a is not present because the operation is occurring at the root. In this case, there is obviously no need to adjust weight w_1. In practice, operations (w1) and (w7) would obviously be altered to allow the shifting of more than one unit of overweight at a time. However, this would not improve the worst case analysis.

In order to discuss these operations, we need the following definitions.

Suppose that e is an edge from a parent u to a child v and that e' is an edge from the same parent u to another child v'. Then we call e' the *sibling edge* of e. We use the terms *parent edge* and *parent node* to refer to the edge or node immediately above another edge or node.

We will now briefly describe a situation in which the operations from [8] can be applied many more times than is necessary if the order chosen turns out to be unlucky. Consider the red-balancing operations. Nurmi and Soisalon-

Soininen have similar operations, but they do not require that w_1 be at least one. One can show that, with their original operations, $\Omega(k^2)$ red-balancing operations can occur, regardless of the original size of the tree. To see this, consider k insertions, each one inserting a new smallest element into the search tree. This will create a sequence of k red edges and $k-1$ red-red conflicts. Now start applying the first red-balancing operation to the left-most red-red conflict. The same bottom edge will take part in $k-1$ operations. Then, below the final sibling to that edge, there will be a sequence of $k-2$ red edges in a string going to the left. The bottom red edge in the left-most red-red conflict will now take part in $k-3$ red-balancing operations. Continuing like this, a total of $\Omega(k^2)$ operations will occur. This is fairly serious since the red-balancing operations are among those which change the actual structure of the tree and thus necessitate locks which prevent users who are simply searching in the tree from accessing certain nodes. In contrast, with our new operations, the number of these restructuring operations is never more than the number of updates.

With the modifications we have made, applying one of the red-balancing operations decreases the number of red-red conflicts in the tree. This greatly limits the number of times they can be applied. Furthermore, as opposed to the operations proposed in [8], none of our overweight handling operations can increase the number of red-red conflicts. We avoid this by increasing the number of distinct rebalancing operations allowed. In some cases, this implies that we lock as many as four more nodes than they would have, though often it would be the same number.

However, these modified operations significantly improve the worst-case number of rebalancing operations.

The following lemma shows that these operations are sufficient for rebalancing any chromatic tree, given that the process eventually terminates.

Lemma 3.1 Suppose the tree T is a chromatic tree, but is not a red-black tree. Then at least one of the operations listed in the appendix can be applied.

Proof Suppose T contains an overweight edge e. If e's sibling edge, f, is overweighted, then (w7) can be applied. If f has weight one, then (w5), (w6), or the push operation can be applied. So, assume that f has weight zero.

If none of the operations (w1), (w2), (w3), or (w4) can be applied, then at least one of f's children must also have weight zero. Hence, if neither a push nor a weight decreasing operation can be applied, there must be a red-red conflict.

Suppose T contains a red-red conflict. Consider a red-red conflict which is closest to the root. Let e_1 be the bottom edge and e_2 the top edge. The parent of e_2 cannot be red since otherwise there would be another red-red conflict closer to the root. If neither of the red-balancing operations can be applied, then the sibling of e_2 must be red. Hence the blacking operation can be applied.

Therefore, if a chromatic tree is not a red-black tree, there is always some operation which can be applied. □

In the remainder of this paper, we prove bounds on the number of times these operations can be applied. Thus, after a finite number of operations applied in any order, a chromatic tree T will become a red-black tree.

4 Complexity

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 will assume that the operations are done sequentially. At time 0, there is a red-black tree, at time 1 the first operation has just occurred, at time 2 the second operation has just occurred, etc.

In order to bound the number of operations which can occur, it is useful to follow red-red conflicts and units of overweight as they move around in the tree due to various operations. In order to do so, we notice that each of the rebalancing operations preserves the number of edges, and one can give a one-to-one mapping from the edges before an operation to those after. Thus one can talk about an edge e over time, even though its end points may change during that time. In the appendix, the one-to-one correspondence has been illustrated by the naming of the edges.

We define a fall from an edge e to be a path from e down to a leaf.

Definition 4.1 A *fall* from an edge e at time t is a sequence of edges $e_1, \ldots, e_k$, $k \geq 1$, in the tree at time t, such that $e_1 = e$, e_k is a leaf edge, and for each $2 \leq i \leq k$, e_i is a child of e_{i-1}. The *weight* of this fall is the sum $\Sigma_{1 \leq i \leq k} w(e_i)$. $\square$

Because of balance condition C2 of definition 2.2, all of the falls, from any given edge e in a chromatic search tree, have the same weight. Clearly, if the tree is red-black and an edge e has heavy falls, then there is a large subtree below e. In a chromatic tree, however, an edge could have heavy falls because many edges below it have been deleted and have caused edges to become overweighted. In this case, e may not have a large subtree remaining. It will be useful, though, to somehow count those edges below e which have been deleted. Edges are inserted and deleted and we want to associate every edge which has ever existed with edges currently in the tree.

Definition 4.2 Any edge in the tree at time t is *associated* with itself. When a node is deleted, the two edges which disappear, and all of their associated edges, will be associated with the edge which was the parent edge immediately before the deletion. $\square$

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

Definition 4.3 Define an *A-subtree* (associated subtree) of an edge e at a particular time t to be the set of all the edges associated with any of the edges currently in the subtree below e together with the edges currently associated with e. $\square$

Lemma 4.4 If an edge e has falls of weight W at time t, then there are at least $2^W - 1$ edges in the A-subtree of e at time t.

Proof The proof will be by induction on the time.

Base case:

We look at the situation at time 0. The A-subtree of e is simply e together with its subtree.

We will show that the subtree is large enough using induction on W. If $W = 1$, everything is fine because there is at least one edge and $2^W - 1 = 1$.

Suppose W is greater than one. Consider any fall from e. As the tree is red-black at time 0, all edges have weight zero or one. Among the edges on this fall which have weight one, let g be the one which is closest to the root. The edge g must have two children, f_1 and f_2. Then f_1 and f_2 both have falls of weight $W - 1$. Let S_1 be the subtree of f_1 and S_2 be the subtree of f_2. By the induction hypothesis, S_1 and S_2 each have at least $2^{W-1} - 1$ edges. The subtree of e contains the disjoint subtrees S_1 and S_2, along with the edge e, so it contains at least $(2^{W-1} - 1) + (2^{W-1} - 1) + 1 = 2^W - 1$ edges.

Induction step:

Assume that $t \geq 1$ and consider the possible operations at time t individually.

Insertion: If e is one of the edges just added, then $W = 1$, and $2^W - 1 = 1$. Since the A-subtree of e contains the edge e, it has enough edges. No other falls change weight, and no A-subtrees decrease in size.

Deletion: Suppose that e is the parent edge from the deletion. At time $t - 1$, a fall of weight W also existed, so a sufficiently large A-subtree existed then. Everything that was in the A-subtree of e at time $t - 1$ is still there, so the A-subtree is large enough. The argument is similar for edges above e. For the remaining edges, there is nothing to show.

Other Operations: These operations preserve the properties of chromatic trees, so any two falls from a particular edge have the same weight. They can change the A-subtrees of some edges, but no A-subtrees of edges with weight greater than one are altered. In addition, no edge with weight greater than one has heavier falls after one of these operations than it did before. Operations having these properties cannot make the lemma fail. To see this, assume to the contrary that it fails at time t. Among those edges which no longer have large enough A-subtrees, choose an edge e which is furthest from the root of the tree. As argued above, e can only have weight zero or one. Consider one of the falls from e. This fall must have weight greater than one, or there is no problem, so the edge must have two children, say f_1 and f_2. Both of these children have falls of weight $W - 1$ or W, so, as they are further from the root than e, they have A-subtrees of size at least $2^{W-1} - 1$. Since the A-subtree of e contains both of these A-subtrees and the edge e, it contains at least $2^W - 1$ edges, contradicting the assumption. Therefore, all the A-subtrees are still large enough. $\square$

Recall that $N = |T| + 2k$ is the bound on the maximum number of nodes the tree ever has. Let $M = \lfloor \log_2(N + 1) \rfloor - 1$. In the following theorem, we prove that no edge can have falls of weight greater than M. In proofs to come, this number will turn up frequently.

Theorem 4.5 *If the falls from edge e have weight W at any time $t \geq 0$, then $W \leq M$.*

Proof Lemma 4.4 says that if e has falls of weight W, then the A-subtree of e contains at least $2^W - 1$ edges. As T is chromatic, e's sibling edge also has falls of weight W, implying that the A-subtree of e's sibling also contains at least $2^W - 1$ edges. Thus, there must have been at least $2^{W+1} - 2$ distinct edges in the tree, though not necessarily all at the same time. The total number of edges in T through time t is bounded by $N - 1$, so $2^{W+1} \leq N + 1$, from which the theorem follows. $\square$

In the following sections, we look at the types of operations individually and bound the number of times they can be applied. Theorem 4.5 is used to bound the number of times the blacking operation and the push operation can be applied. The other operations are much easier to handle; the theorem is unnecessary for bounding the number of times they are applied.

5 Red-Red Conflicts

The only operation which increases the number of red-red conflicts is the insertion; each insertion increases this total by at most one. The edge above the top edge in the red-red conflict will be called the parent edge.

The blacking operation is only applied when at least one of the child edges is involved in a red-red conflict. That red-red conflict is eliminated, but the operation could create another red-red conflict involving the parent edge. If only one of the child edges was involved in a red-red conflict before the operation, but a new red-red conflict was created, one can identify the new red-red conflict with the old one. If both child edges were involved in red-red conflicts, if a new red-red conflict was created, one can identify the new red-red conflict with the old one the left child edge was involved in. With each of the other operations which move the location of a red-red conflict in the process of rebalancing, one can always identify the new red-red conflict with a previous one; the lower edge in each new red-red conflict was also involved in a previous red-red conflict. Thus, one can follow the progress of a red-red conflict.

Lemma 5.1 No red-red conflict can be involved in more than $M-1$ blacking operations.

Proof Suppose a particular red-red conflict is involved in blacking operations at times $t_1, t_2, \ldots, t_r$, where $t_i < t_{i+1}$ for $1 \leq i \leq r-1$. Let e_i be the lower edge of the red-red conflict at time $t_i - 1$, let f_i be the higher edge, and let g_i be the parent edge for the blacking operation. At time t_i, the edge f_i has just been made black and the edge g_i has just been made red (though, if $i = r$, then g_i may not have been made red). If $i \neq r$, then g_i becomes the lower edge of the red-red conflict. Clearly, falls from g_1 will have weight W for some $W \geq 2$, as the weight of g_1 is at least one at time $t_1 - 1$ and as there will always be a leaf edge of weight at least one somewhere under e_1.

It follows from the above that g_i is e_{i+1}, since it becomes the lower edge of the red-red conflict. We show that the weights of falls from e_{i+1} do not change from time t_i through time $t_{i+1} - 1$. Notice that all of the operations preserve the weights of falls beginning above or below the location where the operation takes place; this is necessary, of course, in order to maintain condition C2 of definition 2.2. This means that for the weight of the falls from e_{i+1} to change, e_{i+1} has to be involved in the operation which causes the weight change. We now discuss the different operations.

The operation in question cannot be the blacking operation as, by assumption, this red-red conflict is not involved in a blacking operation (again) until time t_{i+1}. An insertion cannot involve a red-red conflict. Any red-red conflict involved in a deletion operation, (w1), (w2), (w7), or a push would disappear, which, by assumption, it does not. For the operations (w5) and (w6), e_{i+1} could only be the top edge, a, which clearly does not have the weight of its falls changed. For the operations (w3) and (w4), e_{i+1} could only be the edge b, but then the red-red conflict would disappear. Finally, for the red-balancing operations, e_{i+1} can only be the edge b, but this is impossible, since the red-red conflict would disappear.

We have proved that the weight of falls from e_{i+1} remains the same from time t_i through time $t_{i+1} - 1$. At time t_{i+1}, the blacking operation has occurred, so the weight of f_{i+1} has changed from zero to one. Thus, the weight of falls from g_{i+1}, which is also e_{i+2}, is exactly one more than the weight of falls from e_{i+1}.

By induction, it follows that at time t_r, the weight of falls from f_r is at least $W + r - 1$ which is at least $r + 1$, as $W \geq 2$. By theorem 4.5, we find that $r \leq M - 1$. □

Because the blacking operation can only be used when there is a red-red conflict, we obtain the following:

Corollary 5.2 At most $k(M - 1)$ blacking operations can occur. □

It is easy to bound the number of times the red-handling operations can be applied:

Lemma 5.3 At most k red-balancing operations can occur.

Proof Each red-balancing operation removes a red-red conflict. As only the insertion operation increases the number of red-red conflicts, and it can increase this number by at most one, it follows that the number of red-balancing operations is bounded by the number of insertions. □

6 Overweight

The only operation which can increase the total amount of overweight in the tree is the deletion, and each deletion increases this overweight by at most one. Each of the weight decreasing operations decreases the total amount of overweight in the tree. The push operation decreases the overweight of some edge, but not necessarily the total amount of overweight (when $w_1 = 0$, the total amount of overweight in the tree is decreased). In the following, we only refer to the push operation as a push if it fails to decrease the total amount of overweight in the tree. Otherwise, we simply consider it to be one of the overweight decreasing operations.

Whenever new overweight is created by a deletion, we say that a new *unit* of overweight has been created. For the sake of the following proof, we assume that units of overweight are *marked* such that they can be distinguished and we can follow them as they move around in the tree.

Lemma 6.1 At most $s(M-2)$ push operations can occur.

Proof Suppose a particular unit of overweight u is moved up by a push operation at times $t_1, t_2, \ldots, t_r$, where $t_i < t_{i+1}$ for $1 \leq i \leq r-1$. Just before the first push operation at time $t_1 - 1$, u sits on an overweighted edge (the edge b), so this edge has falls of weight at least two.

Assume that at time $t_i - 1$, u sits on an edge with falls of weight W. At time t_i, it has been pushed up (onto the edge a). We will argue that at any time t between time t_i and time $t_{i+1} - 1$, this unit will sit on an edge with falls of weight at least $W + w - 1$, where w is the weight of the edge at time t. If this holds, then at time $t_{i+1} - 1$ this unit must sit on an edge of weight at least two, and the falls must then have weight at least $W + 2 - 1 = W + 1$.

Notice first that all the operations preserve the weight of falls from the topmost edges involved in the operation. Actually, this is an absolutely necessary requirement to ensure that condition C2 of definition 2.2 is fulfilled after the operation. This means that the weight of falls from an overweighted edge cannot change due to operations taking place in its subtree. Thus, an inspection of the operations shows that the weight of falls from an overweighted edge can only change by decreasing the weight on the overweighted edge itself. This means that while a unit of overweight remains on an edge, the claim is certainly true.

In between the push operations, units of overweight can also be moved onto another edge when a deletion occurs (see the deletion operation in the appendix). However, if the claim holds immediately before the deletion, then these units of overweight sit on edge b, which must have falls of weight at least $W + w_2 - 1$. When they are moved to edge a, the claim still holds, since this edge has falls of weight $W + (w_1 + w_2) - 1$.

We have now proved that at time $t_{i+1} - 1$ this unit of overweight will sit on an edge with falls of weight at least one greater than at time $t_i - 1$. By induction, we have obtained that at time t_r, the weight of falls from edge a in the last push operation is at least $r + 2$. By theorem 4.5, we find that $r \leq M - 2$, from which the result follows. $\square$

It is easy to bound the number of times weight decreasing operations can be applied:

Lemma 6.2 At most s weight decreasing operations can occur.

Proof Only deletions introduce overweight and at most one unit is added each time. As weight decreasing operations remove one unit of overweight when they are applied, at most s such operations can occur. $\square$

7 Conclusion

Theorem 7.1 Assume that a red-black tree T initially has $|T|$ nodes. Furthermore, assume that k insertions, s deletions, and a number of rebalancing operations are performed. Then the number of rebalancing operations is no more than $(k + s)(\lfloor \log_2(N + 1) \rfloor - 1) - s$, where $N = |T| + 2k$ is the obvious bound on the number of nodes in the tree. In addition, the number of rebalancing operations which change the structure of the tree is at most $k + s$.

Proof Let us summarize how many times each type of operation can be used:

Operation	Bound	From
blacking	$k(\lfloor \log_2(N + 1) \rfloor - 2)$	corollary 5.2
red-balancing	k	lemma 5.3
push	$s(\lfloor \log_2(N + 1) \rfloor - 3)$	lemma 6.1
weight decreasing	s	lemma 6.2

The results follow by adding up the bounds. $\square$

Acknowledgements

We would like to thank the anonymous referee for some very useful suggestions.

References

[1] G. M. Adel'son-Vel'skiĭ and E. M. Landis, An algorithm for the organisation of information, *Dokl. Akad. Nauk SSSR* **146** (1962) 263–266, (in Russian; English translation in *Soviet Math. Dokl.* **3** (1962) 1259–1263).

[2] R. Bayer and E. McCreight, Organization and maintenance of large ordered indexes, *Acta Inf.* **1** (1972) 97–137.

[3] J. Boyar and K. S. Larsen, Efficient rebalancing of chromatic search trees, *in* "Algorithm Theory – SWAT '92" (O. Nurmi and E. Ukkonen, Eds.), pp. 151–164, Lecture Notes in Computer Science, Vol. 621, Springer-Verlag, Berlin, 1992.

[4] L. J. Guibas and R. Sedgewick, A dichromatic framework for balanced trees, *in* Proceedings, Nineteenth Annual Symposium on Foundations of Computer Science", pp. 8–21, IEEE Computer Society Press, Long Beach, 1978.

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

[6] H. T. Kung and P. L. Lehman, A concurrent database manipulation problem: binary search trees, *in* Proceedings, Fourth International Conference on Very Large Data Bases, p. 498, IEEE Computer Society Press, Long Beach, 1978, (abstract; full version in Concurrent manipulation of binary search trees, *ACM TODS* **5** (1980) 354–382).

[7] L. Malmi, An efficient algorithm for balancing binary search trees, Technical Report, TKO-B84, Department of Computer Science, Helsinki University of Technology, 1992.

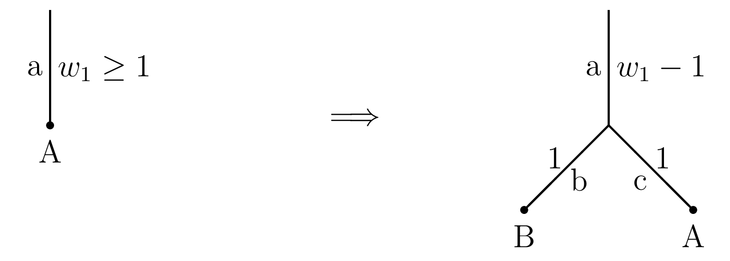
[8] O. Nurmi and E. Soisalon-Soininen, Uncoupling updating and rebalancing in chromatic binary search trees, *in* Proceedings of the Tenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, pp. 192–198, ACM Press, New York, 1991.

[9] O. Nurmi, E. Soisalon-Soininen and D. Wood, Concurrency control in database structures with relaxed balance, *in* Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, pp. 170–176, ACM Press, New York, 1987.

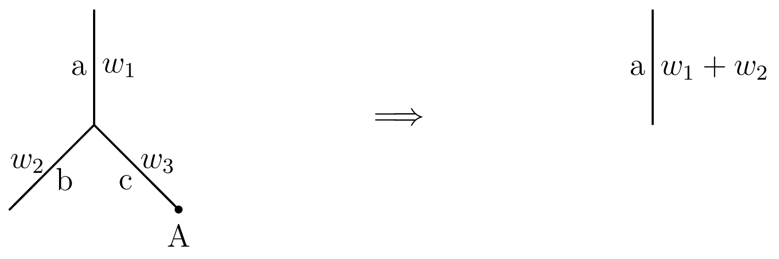
[10] W. Pugh, Skip lists: A probabilistic alternative to balanced trees, *in* "Algorithms and Data Structures", (Proceedings of WADS '89, F. Dehne, J.-R. Sack and N. Santoro, Eds.), pp. 437–449, Lecture Notes in Computer Science, Vol. 382, Springer-Verlag, Berlin, 1989.

[11] W. Pugh, Concurrent maintenance of skip lists, Technical Report, CS-TR-2222.1, University of Maryland, College Park, 1989, (Revised June, 1990).

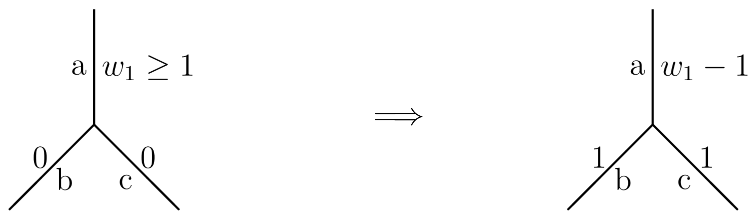
Appendix



Insertion.
Comment: the leaf B is inserted to the left of the leaf A.

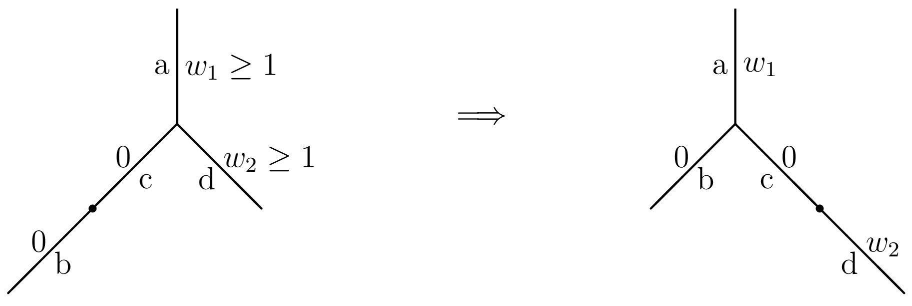


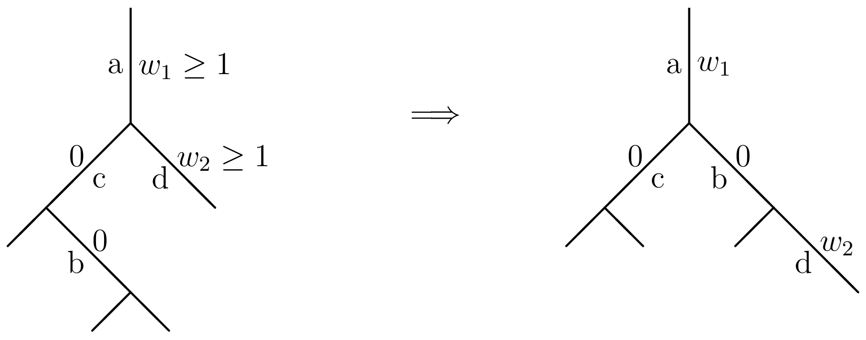
Deletion.
Comment: the leaf A is deleted.



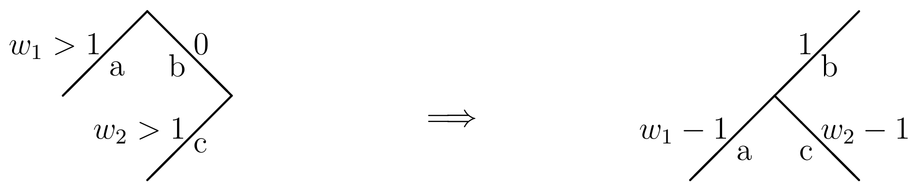
Blacking.

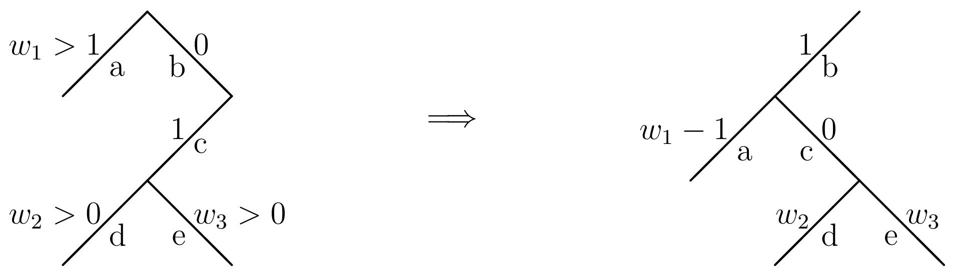
Restriction: at least one edge must be in a red-red conflict.

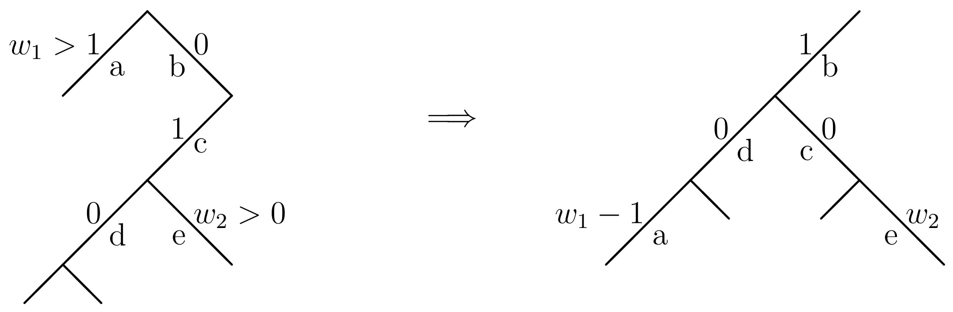


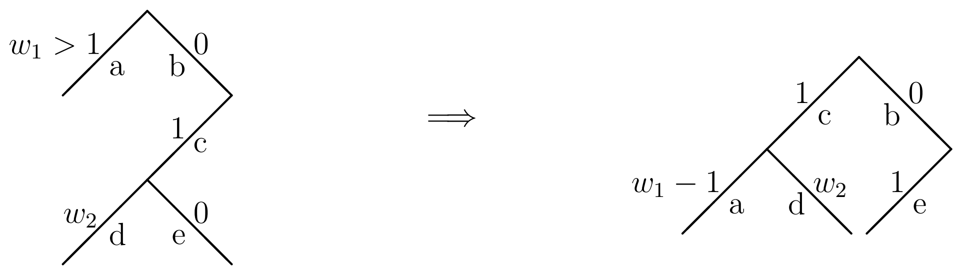


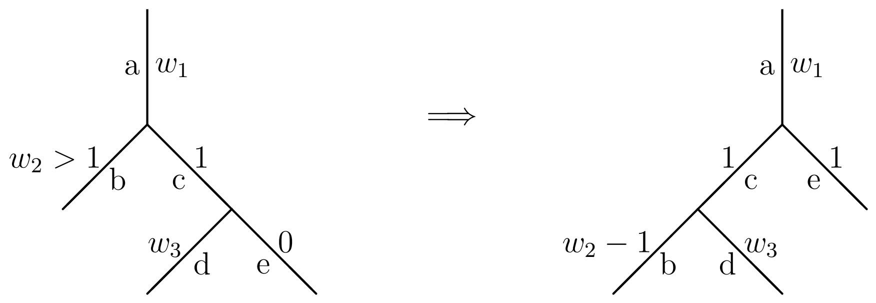
Red-balancing.

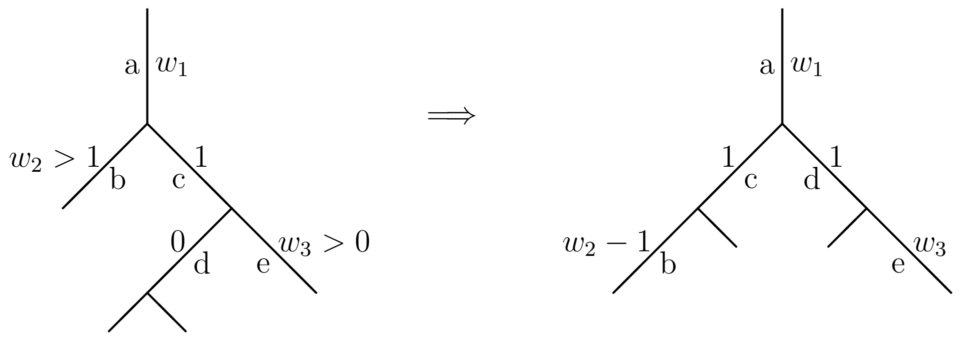

$w_1 > 1$
a
b
0
$w_2 > 1$
c
$\Longrightarrow$
1
b
$w_1 - 1$
a
c
$w_2 - 1$

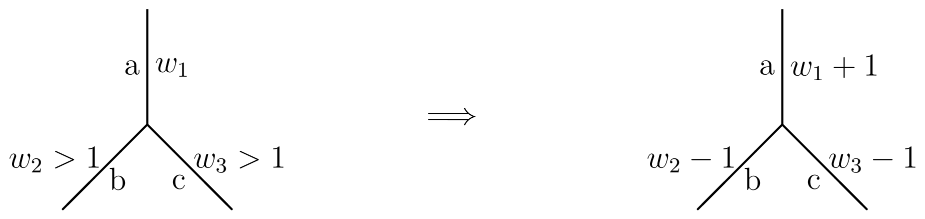

$w_1 > 1$
a
b
0
1
c
$w_2 > 0$
d
e
$w_3 > 0$
$\Longrightarrow$
1
b
$w_1 - 1$
a
c
0
w_2
d
e
w_3


$w_1 > 1$
a
b
0
1
c
0
d
e
$w_2 > 0$
$\Longrightarrow$
1
b
0
d
c
0
$w_1 - 1$
a
e
w_2

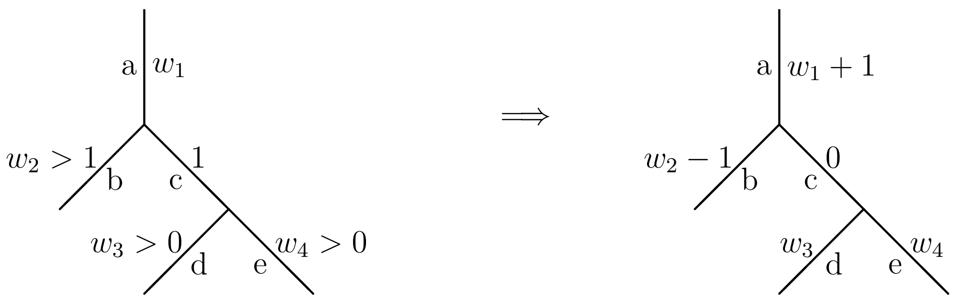

$w_1 > 1$
a
b
0
1
c
w_2
d
e
0
$\Longrightarrow$
1
c
b
0
$w_1 - 1$
a
d
w_2
1
e







Weight decreasing (1-7).



Push.