# UNCOUPLING UPDATING AND REBALANCING IN CHROMATIC BINARY SEARCH TREES*

OTTO NURMI AND ELJAS SOISALON-SOININEN
*Department of Computer Science, University of Helsinki*
*Teollisuuskatu 23, SF-00510 Helsinki, Finland*

**Abstract.** In order to gain maximal efficiency of the concurrent use of search trees the number of nodes to be locked at a time should be as small as possible, and the locks should be released as soon as possible. We propose a new rebalancing method for binary search trees that allows rebalancing to be uncoupled from updating, so as to make updating faster. The trees we use are obtained by relaxing the balance conditions of *red-black* trees. When not involved with updating, the rebalancing task can be performed as a shadow process being active all the time, or it can be performed outside rush hours, at night, for example.

## 1. INTRODUCTION

*Red-black trees* [6] are balanced binary search trees with several properties that make them a good choice for an in-core structure whenever fast random access of data is desired. They have $O(n)$ size and $O(\log n)$ access time, and they can be updated in $O(\log n)$ time, where $n$ is the number of keys stored in the tree. After insertions and deletions the tree must be rebalanced by *rotations* in order to keep these time bounds. For red-black trees there exists a bottom-up rebalancing method that requires at most three single rotations for an update operation [14, 15]. They have a top-down rebalancing method that needs $O(\log n)$ time for an update operation and $O(\log n)$ rotations (see [6]). When red-black trees are used in *priority search trees* [10], the trees can be updated in $O(\log n)$ time. They make the *persistent trees* [12] efficient. Red-black trees are called *symmetric binary B-trees* in [2] and *balanced trees* in [14, 15].

We assume that the data structure is used as a *dictionary;* its operations are *search, insert,* and *delete.* Insert and delete operations are called *update operations.* Each individual operation is assigned to a separate process. The processes that perform updating are called *updaters.*

If several processes operate concurrently in a data structure there must be a way to prevent simultaneous writing and reading the same part of the structure. A common strategy for concurrency control in tree structures is that a process *locks* some parts of the tree; other processes cannot access a locked part. For efficiency, only a small part of the structure should be locked at a time. The sooner the parts are unlocked the sooner the individual processes terminate.

In a conventional bottom-up rebalancing method rebalancing transformations are carried out when an updater returns from the inserted or deleted node to the root. If a bottom-up method is used in a concurrent environment, the path from the root to a leaf must be locked for the time a writer operates; otherwise the process can lose the path to the root. During the time the root is locked by an updater, no other process can access the tree. Thus, at most one updater can be active at a time.

There exists a *top-down* balancing method for red-black trees [6] in which an updater modifies the tree on the way from the root to the leaf to be inserted or deleted. Since no further rebalancing is necessary after an operation, an updater needs never to return the path to the root. Only a constant number of nodes must be locked at a time. The height of the tree is $O(\log n)$ all the time, and any key can thus be found in $O(\log n)$ time.

We shall take a different approach to the rebalancing problem by uncoupling the rebalancing and updating. The updaters perform no rebalancing but leave certain information for separate rebalancing processes, which will later retain the balance. A rebalancing process can run as a shadow process concurrently with other processes (cf. *on-the-fly* garbage collection [3]) or it can be activated when there are only few other active processes. Several rebalancers can work concurrently.

*The work is supported by the Academy of Finland.

In our approach, a process must lock a small constant number of nodes at a time. Since the updaters do no rebalancing and the separate rebalance operation is divided into several small steps the nodes can be unlocked rapidly. The tree may temporarily be out of balance, *i.e.*, its height is not necessarily bounded by $O(\log n)$.

The separation of updating and rebalancing was proposed already by Guibas and Sedgewick in [6]. Their solution seems to allow only insertions. For *AVL-trees* [1] a solution without deletions was presented by Kessels in [7]. It was completed with deletions in [11]. As we shall see the uncoupling is much simpler for red-black trees and, thus, a fewer nodes must be locked at a time.

In Section 2 we review the definition of (balanced) red-black trees and extend the definition to include certain unbalanced trees. The implementation of the update operations is discussed in Section 3 and the separate rebalance operations are presented in Section 4. A concurrency control method for the operations is discussed in Section 5. Section 6 contains conclusions and some remarks.

## 2. CHROMATIC BINARY TREES

In this section we recapitulate the balance conditions for a red-black tree and then relax the conditions so that certain unbalanced trees will satisfy them. The loose conditions are needed since we shall permit for an updater to leave the tree in an unbalanced shape.

We shall only consider *leaf-oriented* binary search trees, which are full binary trees (each node has either two or no children) with the keys stored in the leaves. The internal nodes contain *routers*, which guide a search through the structure. The router stored in a node $v$ must be greater than or equivalent to any key stored in the leaves of $v$'s left subtree and smaller than any key in the leaves of $v$'s right subtree. We do not require the routers to be keys present in the tree. The loose definition of the routers enables a very simple strategy for concurrency control, as will be shown in Chapter 5. The routers far away from a leaf need not be updated even after deleting a key.

With each edge $e$ of the tree we associate a non-negative integer $w(e)$, called the *weight* or *color* of $e$. An edge $e$ is *red* if $w(e) = 0$ and *black* if $w(e) = 1$. If $w(e) > 1$, the edge is *overweighted*. If $e = (u, v)$ is an edge of the tree, $w(e)$ is stored in $v$, *i.e.*, the weight of the edge between a node and its child is stored in the child.

The *weighted length* of a path is the sum of the weights of its edges. The *weighted level* of a node is the weighted length of the path from the root to the node. The weighted level of the root is 0.

The definition of a (balanced) red-black tree is adopted from [6]:

**Definition 1.** A full binary tree $T$ with the following balance conditions is a *red-black* tree:

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

B2: The weighted level of all leaves of $T$ is the same.

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

B4: $T$ has only red and black edges. □

Red-black trees, as defined above, are *balanced*, *i.e.*, their height is bounded by $O(\log n)$ where $n$ is the number of their nodes (or the number of their leaves) (see Bayer [2], in which red edges are called *horizontal* and black edges *vertical*).

Tarjan [14, 15] has defined update operations for red-black trees in which rebalancing is carried out immediately when a leaf has been inserted or deleted. The rebalancing transformations may propagate from the inserted or deleted node towards the root of the tree. Although the method of [14, 15] never requires more than a constant number of rotations after an update operation, the number of other needed rebalancing actions (called *promote* and *demote* in [14, 15]) can be $\Theta(\log n)$ where $n$ is the number of nodes in the tree. The method is inefficient if high degree concurrency is desired. In the update operations of Guibas and Sedgewick [6] the balancing is done before inserting or deleting a leaf. In their method, the need for balancing transformations may propagate only in the top-down direction. The method can be used in a concurrent environment in such a way that only a few nodes need to be locked at a time. It does not support separation of updating and balancing when deletions are present.

The trees we shall use will be defined by relaxing the red-black balance conditions. We withdraw the conditions B3 and B4 and allow all non-zero weights in the edges closest to the leaves. The condition B2 is needed as such in the relaxed definition.

**Definition 2.** A full binary tree $T$ with the following conditions is a *chromatic* tree:

RB1: The parent edges of $T$'s leaves are not red.

RB2: The weighted level of all the leaves of $T$ is the same. □

A chromatic tree can be out of balance but any red-black tree is a chromatic tree. An empty tree is a red-black tree as well as a tree consisting only of a single leaf.

## 3. UPDATING A CHROMATIC TREE

The update operations for a chromatic tree are designed so that they keep the weak balance conditions. The conditions are loose enough to prevent the need of immediate rebalancing. Since the tree must be a full binary tree an insert operation adds a new internal node

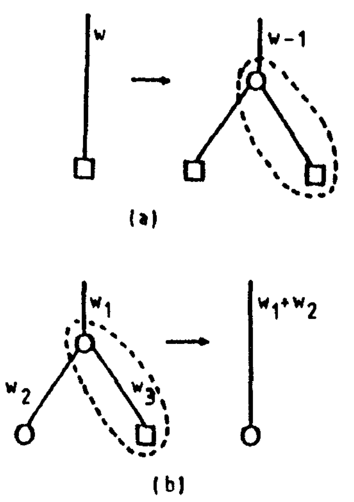

FIG. 1. Update operations. (Only the involved nodes are depicted; the operations have symmetric variants. A circle denotes any node, a square denotes a leaf; a line without an associated weight denotes a black edge.) (a) Insertion: an internal node and a leaf are inserted. (b) Deletion: a leaf and an internal node are deleted.

and a new leaf into the tree. (For simplicity, we do not consider the trivial cases in which the tree is initially empty or consist of a single leaf.) A delete operation removes a leaf and an internal node. Since the routers are not necessarily keys present in the tree, even a delete operation does not need to modify routers far away from the deleted nodes. The operations are described below (cf. Fig. 1).

**Insertion:** The new key is searched from the tree. If the key is found the process terminates. An unsuccessful search ends up in a leaf, say $v$. A new internal node $u$ is inserted in the structure in the place of $v$, and $v$ and a new leaf containing the new key are made children of $u$. The children are ordered such that the one containing the smaller key will be $u$'s left child. The router for $u$ is a copy of the key contained in its left child.

The parent edge of $u$ gets the weight of $v$'s old parent edge $-1$. The weights of the child edges of $u$ are set to one.

**Deletion:** The key to be deleted is searched from the structure. If it is not found, the process terminates. Otherwise, the leaf containing the key is removed. Its parent is replaced by the parent's other child, say $u$.

The weight of $u$'s new parent edge is the sum of the weight of $u$'s old parent edge and the weight of the parent edge of the removed internal node.

The new weights are assigned in such a way that the conditions RB1 and RB2 hold true.

An insertion may introduce a new red edge and several insertions may introduce a sequence of consecutive red edges to the path from the root to a leaf. A deletion can introduce a new overweighted edge or it can increase an existing overweight.

## 4. REBALANCING A CHROMATIC TREE

This section describes the actions performed by a rebalancing process. The process searches violations of the red-black conditions and when one is found it updates the weights of a few edges, and it may additionally perform a single or a double rotation. The operations are designed so that RB1 and RB2 hold and the tree will be modified towards a red-black tree.

A node $v$ has a *red-red conflict* if one of the paths from it to a leaf begins with two consecutive red edges. It has an *overweight conflict* if one of its child edges is overweighted. A node can have several conflicts.

The operations for red-red conflicts are similar with those defined in [14, 15]; the difference is that we do not care if a new conflict will arise closer the root of the tree. The new conflict is left for a new separate rebalancing action.

The operations for overweighted edges resemble the operations that are performed after a deletion in [14, 15]. As in the case of a red-red conflict, we do not immediately remove the new conflict that may arise closer to the root. The operations may create new red-red conflicts below the node in which the operation was performed. Some of them must be removed immediately (see Case 5 of Definition 3).

The rebalancing process searches nodes that have conflicts. When it has found such a node it will remove the conflict by using one of the transformation rules defined below. There is one exception: if the rebalancer founds a red-red conflict in a node whose all adjacent edges are red, the conflict cannot be removed before the conflict in its parent.

The rebalancing transformations defined below are illustrated in Fig. 2.

**Definition 3.** Let $T$ be a chromatic tree and $v$ one of its nodes that have a conflict. The *rebalancing transformation* in $v$ depends on the weights of some edges close to $v$ (if several of the cases apply, choose one of them):

**Case 1.** Both of the child edges of $v$ are red, and either $v$ is the root or the parent edge of $v$ has a nonzero weight: Set the color of the child edges black; if $v$ is not the root, then decrease the weight of the parent edge of $v$ by one.

**Case 2.** The left (resp. right) child edge of $v$ is red, its other child edge is not red, and the left (right) child edge of $v$'s left (right) child is red: Perform a single rotation to the right (left).

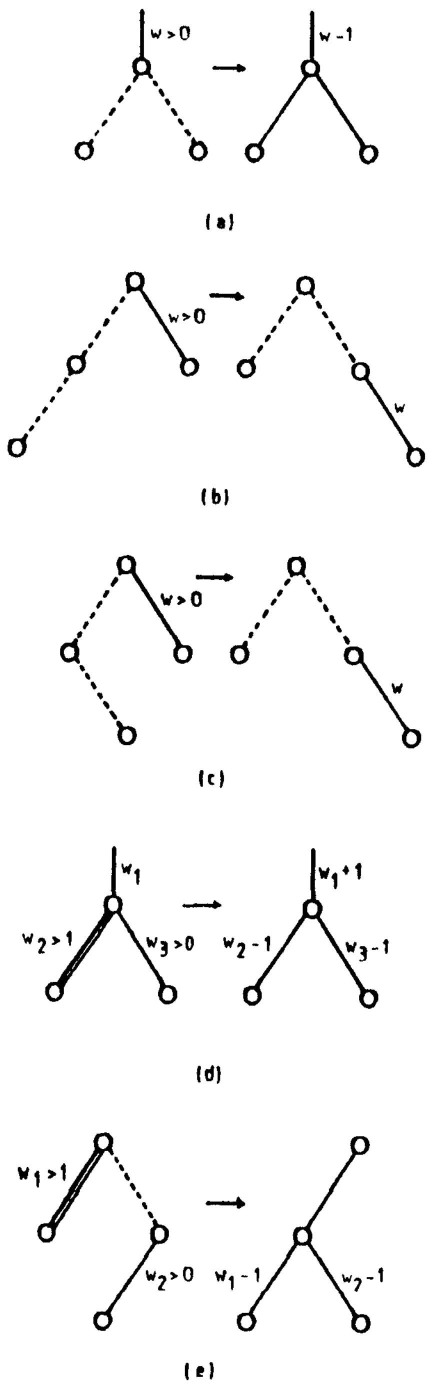

FIG. 2. Rebalance operations. (All operations but (a) have symmetric variants. A broken line denotes a red edge, a double line denotes an overweighted edge; for other denotations see Fig. 1.) (a) Case 1 (the red child edges of the lowermost nodes are not shown): the weights are adjusted. (b) Case 2: a single rotation. (c) Case 3: a double rotation. (d) Case 4: the weights are adjusted. (e) Case 5: a single rotation is performed and the weights are adjusted.

**Case 3.** The left (resp. right) child edge of $v$ is red, its other child edge is not red, and the right (left) child edge of $v$'s left (right) child is red: Perform a double rotation to the right (left).

**Case 4.** One of the child edges of $v$ is overweighted and the other is not red: Decrease the weights of the child edges by one; if $v$ is not the root then increase the weight of the parent edge of $v$ by one.

**Case 5.** The left (resp. right) child edge of $v$ is overweighted, its other child edge leading to a node $u$ is red, and the left (right) child edge of $u$ is not red: Perform a single rotation to the right (left) and after that, perform the operation defined in Case 4 in the left (right) child. □

The transformations clearly keep the conditions for a chromatic tree. (At the first glance, it may seem that in Cases 3 and 4 the condition RB1 could be violated by decreasing the parent edge of a leaf from one to zero. In that case the tree would, however, have a violation of the condition RB2 before the transformation.)

The transformation defined in Case 1 removes at least one red-red conflict. It may introduce a new red-red conflict in a node closer to the root. In the root, no new conflicts can arise. In Cases 2 and 3 a red-red conflict disappears and no new conflicts can arise. Each transformation of Cases 4 and 5 decreases the overweight of at least one edge. The transformation of Case 4 may increase the overweight in an edge closer to the root. The transformations of Cases 4 and 5 may also create new red-red conflicts.

For the correctness of the rebalancing transformations, we first prove the following lemma:

**Lemma 1.** Given a chromatic tree that does not satisfy the balance conditions for red-black trees, the tree has at least one node in which a rebalancing transformation can be carried out.

**Proof.** The only case in which no rebalancing action is possible is a red-red conflict in a node whose all adjacent edges are red. The tree must, however, have another node closer to the root that has a red-red conflict and whose parent edge is not red (at least the root is such a node). In that node, one of the transformations of Cases 1, 2, or 3 can be used. □

It is easy to see that no rule can move overweight downwards in the tree (in the sense that the number of nodes below the overweight would decrease). The overweighted edges will thus disappear by a sequence of transformations defined in Cases 4 and 5. If the tree has no overweight conflicts a new red-red conflict may only appear above a removed red-red conflict and it will disappear in the root at the latest. Thus, for a given chromatic tree, there is at least one sequence of rebalancing transformations defined above that modifies the tree such that it will finally be a red-black tree.

For a stronger result we shall need we first give some definitions that will simplify its proof.

Let $T$ be a chromatic tree with $n$ nodes and $E$ the set of its edges, and let $e = (u, v)$ be its edge. Then, by $nodes(e)$ we denote the number of nodes in the subtree whose root is $v$. The *red-red unbalance* of the edge $e$ is

$$ru(e) = \begin{cases} n - nodes(e), & \text{if } e \text{ and the parent edge} \\ & \text{of } u \text{ are red;} \\ 0, & \text{otherwise.} \end{cases}$$

The red-red unbalance of a red edge that causes a conflict is the "distance" between the conflict and the root of the tree. The red-red unbalance of the tree is the sum of the red-red unbalance of its edges, *i.e.*, $ru(T) = \sum_{e \in E} ru(e)$. Similarly, the *overweight unbalance* of an edge $e$ is

$$ou(e) = \begin{cases} (w(e) - 1)(n - nodes(e)), & \text{if } w(e) > 1; \\ 0, & \text{otherwise.} \end{cases}$$

The overweight unbalance of an edge is the "distance" between the edge and the root multiplied by the overweight. Finally, the overweight unbalance of the tree is $ou(T) = \sum_{e \in E} ou(e)$.

$T$ is clearly a red-black tree if and only if $ru(T) = ou(T) = 0$. The influence of our rebalancing transformations to the unbalance of a tree is described by the following lemma:

**Lemma 2.** Let $T$ and $T'$ be chromatic trees such that $T'$ is obtained from $T$ by using the transformation of Case $i$ $(1 \leq i \leq 5)$ of Definition 3. Then

$$\begin{cases} ru(T) > ru(T') \text{ and } ou(T) = ou(T'), & \text{if } i = 1, 2, \text{ or } 3; \\ ou(T) > ou(T'), & \text{if } i = 4, \text{ or } 5. \end{cases}$$

**Proof.** Directly from definitions. □

The overweight unbalance of a tree cannot be increased by the transformations. In Cases 4 and 5 the red-red unbalance may be increased. The added unbalance must later be removed by the transformations of Cases 1, 2 and 3.

The following theorem tells us that we do not need to fix the order in which a rebalancer removes the conflicts.

**Theorem 1.** Given a chromatic tree, any sequence of rebalancing transformations that is long enough, modifies the tree into a red-black tree.

**Proof.** Let $T$ be a chromatic tree with $n$ nodes, and let $\pi = T_1, T_2, \ldots$ be a sequence of chromatic trees such that $T_1 = T$ and $T_{i+1}$ is obtained from $T_i$ by one of the rebalancing transformations. Let us assume further that $\pi$ is as long as possible, *i.e.*, it is finite if and only if no rebalancing transformations can be applied to its last element.

For the sake of contradiction, let us assume that $\pi$ is infinite. The total weight of the edges of a chromatic tree is always non-negative. No transformation defined in Cases 2–5 can increase the total weight of the tree. The transformation of Case 1 increases the total weight by one, but it never produces an overweighted edge. Thus the total weight of the transformed trees is also bounded from above. There exist only a constant number of binary trees with $n$ nodes and a different shape.

We can conclude that we can get only a finite number of different trees by using our rebalancing transformations. Since $\pi$ is infinite, it must contain at least two equivalent trees. Let $T_p$ and $T_q$ be two trees in $\pi$ such that $T_p = T_q$ and $p < q$, and let $r$ be an index such that $p \leq r < q$.

Assume first that a transformation rule for an overweight conflict (Case 4 or 5) has been used in the tree $T_r$. Both rules decrease the overweight unbalance $ou(T_r)$. Since no other transformations can increase the overweight unbalance, $T_r$ cannot be transformed to $T_q$ by using the rules of Definition 3. Thus $T_p$ cannot be equivalent to $T_q$, which is a contradiction. This means that the sequence between $T_p$ and $T_q$ can contain no applications of Cases 4 and 5.

Assume then that a transformation rule for a red-red conflict (Case 1, 2, or 3) has been used in $T_r$. The red-red unbalance $ru(T_r)$ is decreased. Since the unbalance can only be increased by a rule for an overweight conflict, and they cannot be applied for trees between $T_p$ and $T_q$, $T_r$ can no more be transformed to $T_q$. Thus $T_p$ cannot be equivalent to $T_q$. The contradiction proves that $\pi$ must be finite.

Since $\pi$ was as long as possible and it is finite, the theorem follows from Lemma 1. □

One can easily construct a procedure that rebalances a chromatic tree by visiting its nodes $O(1)$ times. The procedure traverses the tree in the inorder (see [15], e.g.) and whenever it encounters a conflict, it makes the appropriate rebalancing transformations. The traversal must be extended in such a way that possible new red-red conflicts in already visited left subtrees are removed immediately. Since red-red conflicts do not propagate downwards this can be accomplished by visiting a constant number of already visited nodes. After a rotation, the procedure must take care not to go to an already traversed subtree that has been moved from the left.

There is another algorithm that takes an arbitrary binary search tree and transforms it to a complete binary tree (in which the level of the leaves differs at most by one) by using $\Theta(n)$ time and $O(1)$ working space [13]. That algorithm always rebuilds the whole tree whereas the one sketched above does only some local rebuilding when the tree has only some balance violations. The red-black balance conditions are, of course, much weaker than the ones for a complete binary tree.

In an concurrent environment, we cannot fix the order in which a rebalancer traverses the tree. Otherwise, the rebalancer should lock paths from the root to the leaves of the tree. The length of such a path is is not bounded by a constant. In the next section we shall show that by allowing a nondeterministic traversal for the rebalancer only a small constant number of nodes must be locked at a time.

## 5. CONCURRENCY CONTROL IN A CHROMATIC TREE

In this section we present a simple locking strategy for chromatic trees. It prevents simultaneous writing and reading the same data but still allows a high degree of concurrency. The strategy is based on the one of Ellis [4, 5], which was originally developed for AVL and 2-3 trees.

As discussed earlier, we define that a strategy for concurrency control is ***efficient***, if, at a point of time, any process prohibits the access of other processes only to a constant number of nodes. The strategies of [4, 5, 8, 9] are all efficient in that sense.

A process that performs search operations is called a *reader* and a process that may modify the tree (update and rebalance operations) is called a *writer*.

As in [4, 5] we use three kinds of locks, which we call *r-locks, w-locks,* and *x-locks* (they are called $\rho$-, $\alpha$-, and $\xi$-locks in [4, 5]). If a reader holds an $r$-lock in a node, the node cannot be $w$-locked nor $x$-locked by other processes but another reader can $r$-lock it. If a process holds a $w$-lock in a node, the node cannot be $w$-locked nor $x$-locked by other processes but it can be $r$-locked by a reader. Finally, if a node is $x$-locked, no other process can access the node. A reader uses $r$-locks to exclude writers, a writer uses $w$-locks to exclude other writers and $x$-locks to exclude all other processes.

Our strategy is that a reader $r$-locks the node whose contents it is reading. A writer $x$-locks the nodes, whose contents is to be modified. During the search phase of an update operation and when a rebalancer checks whether a transformation should be performed or not, the nodes can be accessed by readers; the writers use $w$-locks instead of $x$-locks during this first phase. (If they used $r$-locks instead, there could arise a dead-lock situation when the $w$-locks are converted to $x$-locks after the search phase.) The nodes are always locked in the top-down direction in order to avoid dead-lock situations.

When a reader advances from the root to a leaf, it uses ***r-lock coupling***, *i.e.*, it $r$-locks the child to be visited next, before it releases the $r$-lock in the currently visited node. Thus, a reader keeps at most two locks at a time.

A writer that will perform an insert operation uses ***w-lock coupling*** during the search phase. When the search terminates at a leaf, the lock in the parent is released and the $w$-lock in the leaf is converted to an $x$-lock. Then the leaf is changed to an internal node with two new leaves as children. The key stored in it is copied into one new leaf; the key to be inserted is stored in the other new leaf. Finally, the router in the internal node and the weights of the edges are assigned as explained in Chapter 3. By using this technique, we do not need to $x$-lock the parent of the node where the search terminated. The process keeps at most two locks in the tree at a time.

During a delete operation, three nodes must be locked on the lowest level: the leaf to be deleted, its sibling, and its parent. To achieve this, a process that will perform a delete operation uses $w$-lock coupling during the search phase. When the leaf to be deleted has been found, its parent is still kept $w$-locked, and the process $w$-locks the sibling of the leaf. Then it $x$-locks the parent of the leaf, the leaf itself, and its sibling. Now the leaf is deleted, the contents of the sibling is copied to the parent and the sibling is deleted, and after adjusting the weights, the only remaining lock is released. If the sibling node was a leaf, the parent must be made to a leaf before the copying. By copying the contents of the sibling to the parent we avoid the need to $x$-lock the grandparent of the deleted leaf. The process locks at most three nodes at a time.

The rebalancing processes traverse the tree nondeterministically. In the node currently visited, a process nondeterministically chooses one of the rebalancing transformations and checks, whether the transformation applies and, finally, if it applies, it is performed by the process. During the checking phase, the process $w$-locks the nodes whose color-fields must be investigated. In the worst case it must $w$-lock four nodes (the visited node, its children, and one of its grandchildren. The nodes are $w$-locked in the top-down direction. Just before the transformation, it converts the $w$-locks of the nodes whose contents will be changed to $x$-locks. If the rotations are implemented by exchanging the contents of nodes, the parent of the conflict node can freely be accessed by other processes. Thus, four nodes must be $x$-locked in the worst case (Case 5).

If a rebalancing process decides that the chosen rebalancing transformation does not apply, it releases immediately all locks and continues searching for other conflicts.

We have not requested that the rebalancer should always perform a transformation if one of them applies. If the rebalancer had to choose a transformation deterministically always when one of them applies it should $w$-lock seven nodes in the worst case in order to select the right type of a transformation.

Let us assume, that a set $S$ of search, insert, and delete operations are executed concurrently with rebalance operations. Only one process can hold an $x$-lock in a node at a time. All nodes modified by a writer are $x$-locked until the operation is complete. Thus, neither a search operation nor the searching phase of an update operation can take a wrong path from the root to a leaf. This means that the search operations of $S$ give the same answers as they would give in some serial execution of $S$. The dead-lock situations are avoided by always locking the nodes in top-down direction, and by excluding other writers before a writer converts $w$-locks to $x$-locks. We can thus conclude that the locking strategy behaves correctly.

## 6. CONCLUSIONS

We have presented a method to update binary search trees in such a way that the rebalancing task can be left for a separate process that performs maybe several *local* modifications in the tree. The trees can temporarily be out of balance but we expect that the update operations insert and delete keys so evenly in the tree that the execution times of the dictionary operations remain tolerable. In that case, however, the conventional updating operations do not need to perform much rebalancing, but they must spend time in checking whether or not a rebalancing transformation must be performed. In a concurrent environment, even the top-down updaters must lock several nodes during the checking phase.

We have split the rebalancing transformations to as small pieces as possible in order to decrease the number of locks needed and to make the processes fast. The sooner the processes unlock the nodes the higher degree of concurrency is obtained. There are, of course, several ways to combine our rebalancing transformations to larger pieces.

Difficult problems that arise if keys can be stored in internal nodes of the tree were avoided by using leaf-oriented search trees in which the routing information of the internal nodes need not be keys present in the structure. Other solutions for the problem can be found in [4, 5, 8, 9].

There is a method to uncouple updating and rebalancing in AVL-trees (see [7, 11]), but it is more complicated than the one for red-black trees, and the rebalancers must lock more nodes at a time. This strengthens the usefulness of red-black trees as an in-core data structure.

## REFERENCES

1. G. M. Adel'son-Vels'kii and E. M. Landis, An algorithm for the organization of information, *Soviet Math. Dokl.* **3** (1962), 1259–1262.
2. R. A. Bayer, Symmetric binary B-trees: Data structure and maintenance algorithms, *Acta Inform.* **1** (1972), 290–306.
3. E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens, On the fly garbage collection: An exercise in cooperation. *Comm. ACM* **21** (1978), 699–975.
4. C. S. Ellis, Concurrent search in AVL-trees, *IEEE Trans. Computers* **C-29** (1980), 811-817.
5. C. S. Ellis, Concurrent search and insertions in 2–3 trees, *Acta Inform.* **14** (1980), 63–86.
6. L. J. Guibas and R. Sedgewick, A dichromatic framework for balanced trees, *19th IEEE Symp. Foundations of Computer Science,* 1978, 8–21.
7. J. L. V. Kessels, On-the-fly optimization of data structures, *Comm. ACM* **26** (1983), 895–901.
8. H. T. Kung and P. L. Lehman, A concurrent database manipulation problem: Binary search trees, *ACM Trans. Database Syst.* **5** (1980), 339–353.
9. U. Manber and R. E. Ladner, Concurrency control in a dynamic search structure, *ACM Trans. Database Syst.* **9** (1984), 439–455.
10. E. M. McCreight, Priority search trees, *SIAM J. Comput.* **14** (1985), 257–276.
11. O. Nurmi, E. Soisalon-Soininen, and D. Wood, Concurrency control in database structures with relaxed balance. *Proc. 6th ACM Conf. Principles of Database Systems*, 1987, 170–176.
12. N. Sarnak and R. E. Tarjan, Planar point location using persistent search trees. *Comm. ACM* **29** (1986), 669–679.
13. Q. F. Stout and B. L. Warren, Tree rebalancing in optimal time and space. *Comm. ACM* **29** (1986), 902–908.
14. R. E. Tarjan, Updating a balanced search tree in $O(1)$ rotations. *Inf. Proc. Lett.* **16** (1983), 253–257.
15. R. E. Tarjan, *Data Structures and Network Algorithms,* Society for Industrial and Applied Mathematics, Philadelphia, Pa., 1983.