Acta Informatica 33, 547–557 (1996)

Acta Informatica

Chromatic binary search trees

A structure for concurrent rebalancing*

Otto Nurmi[1], **Eljas Soisalon-Soininen**[2]

[1]Department of Computer Science, University of Helsinki, Teollisuuskatu 23, FIN-00510 Helsinki, Finland
[2]Laboratory of Information Processing Science, Helsinki University of Technology, Otakaari 1, FIN-02150 Espoo, Finland

Received December 5, 1991 / May 2, 1995

Abstract. We propose a new rebalancing method for binary search trees that allows rebalancing and updating to be uncoupled. In this way we obtain fast updates and, whenever the search tree is accessed by multiple users, a high degree of concurrency. The trees we use are obtained by relaxing the balance conditions of *red-black* trees. The relaxed red-black trees, called *chromatic trees*, contain information of possible imbalance such that the rebalancing can be done gradually as a shadow process, or it can be performed separately when no urgent operations are present.

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, i.e. rebalancing advances upward the tree, that requires at most three single rotations for an update operation [14, 15]. They have a top-down rebalancing method, i.e. rebalancing advances downward the tree, 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 implements a totally ordered finite set of elements chosen from a given domain; 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*.

* The work is supported by the Academy of Finland. A version of this paper appeared at the 9th ACM Conference on Principles of Database Systems, Denver, Colorado, 1991.

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 need 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.

In our approach, a process locks 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, less nodes need be locked at a time.

In Sect. 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 Sect. 3 and the separate rebalance operations are presented in Sect. 4. A concurrency control method for the operations is discussed in Sect. 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 Sect. 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 B1–B4 given in Definition 1. 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 and so is a tree consisting only of a single leaf.

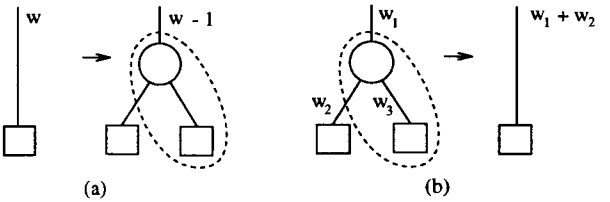

Fig. 1a,b. Update operations. (Only the involved nodes are depicted; the operations have symmetric variants. A circle denotes any node, a square denotes a leaf; a line without an associated weight denotes a black edge.) **(a)** Insertion: an internal node and a leaf are inserted. **(b)** Deletion: a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

3. Updating a chromatic tree

The update operations for a chromatic tree are designed so that they keep the weak balance conditions RB1 and RB2 of Definition 2. 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 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 1.

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 for 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 such that the conditions RB1 and RB2 of a chromatic tree hold and the tree will be modified toward a red-black tree.

A pair $c = ((v, u), (u, t))$ of two consecutive edges of a chromatic tree is a *red-red conflict at* v if both edges (v, u) and (u, t) are red. The node v is the *leading node* of c. A node may have at most 4 red-red conflicts.

If a node v has the child edges $e_1 = (v, u)$ and $e_2 = (v, t)$ and $w(e_1) > 1$ or $w(e_2) > 1$ there are *overweight conflicts* at v. The number of overweight conflicts at v is the sum of the overweight in its child edges, i.e. v has $\max(0, w(e_1)-1)+\max(0, w(e_2)-1)$ overweight conflicts. A node may have both overweight and red-red 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" to the root of the tree. The "distance" to the root from a red-red conflict $c = ((v, u), (u, t))$ is measured by the number of nodes that lie outside the subtree rooted at u. Some of these nodes may appear between v and the root of the tree during the rebalancing while, as we shall see, the nodes in the subtree rooted at u will never be moved above v. The new conflict is left for another 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.

The rebalancing process searches nodes that have conflicts. When the process has found such a node it will remove the conflict by using one of the transformation rules defined below. Note that the transformations are designed such that a red-red conflict at node v is not found if all adjacent edges of v are red. In that case the conflict at the parent node of v must first be removed.

The rebalancing transformations defined below are illustrated in Fig. 2a–e. In figures a broken line denotes a red edge, a double line an overweighted edge; for other denotations see Fig. 1.

Definition 3. *Let T be a chromatic tree and v one of its nodes that have conflicts. The rebalancing transformation at v is defined depending 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 non-zero 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 1. Note that in this case at least one of the grandchild edges of v is red, because it was assumed that there is a conflict at v.

Case 2. The left (resp. right) child edge (v, u) of v is red, its other child edge is not red, and the left (right) child edge of u is red: Perform a single rotation to the right (left) at node v. See Fig. 2(b).

In this case the number of red-red conflicts in the subtree initially with root v is decreased by 1. Notice that if the weight of right child of node u is 0, then a new red-red conflict will be created at u. But then two red-red conflicts at v will be removed by the rotation and thus the total number of red-red conflicts in this subtree will be decreased.

Case 3. The left (resp. right) child edge (v, u) of v is red, its other child edge is not red, and the right (left) child edge of u is red: Perform a double rotation to the right (left) at node v; see Fig. 2(c).

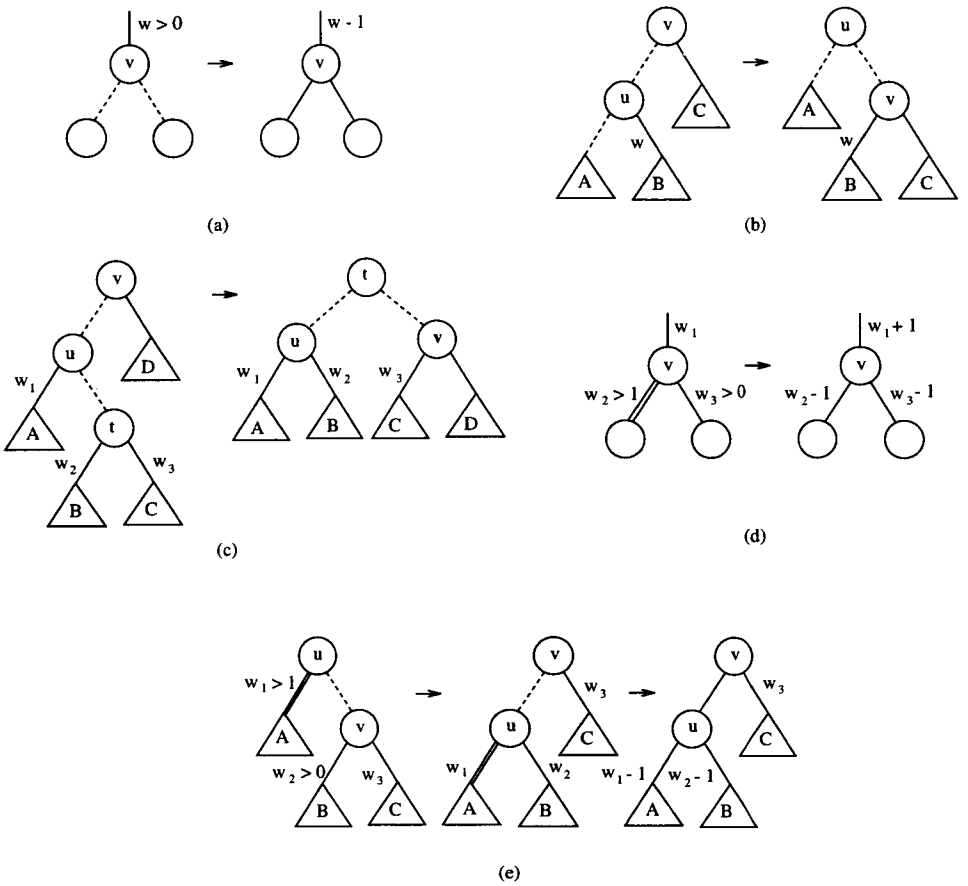

Fig. 2a–e. Rebalance operations. (All operations but (**a**) have symmetric variants. A broken line denotes a red edge, a double line denotes an overweighted edge; for other denotations see Fig. 1.) (**a**) Case 1 (the red child edges of the lowermost nodes are not shown): the weights are adjusted. (**b**) 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.

As in Case 2 the number of red-red conflicts in the subtree initially with root v is decreased by 1. In this case new red-red conflicts may appear at node t, the right child node of u, but, as is easily counted from Fig. 2(c) the total number of red-red conflicts in this subtree is decreased.

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. See Fig. 2(d).

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 left (right) at node v, and after that perform the operation defined in Case 4 in the left (right) child. See Fig. 2(e). □

Next we want to show that the rebalancing transformations as defined above indeed will retain the conditions of a chromatic tree.

Lemma 1. *Let T be a chromatic tree and assume that any one of the transformations given in Definition 3 is applied to T. Then the transformed tree T' is a chromatic tree.*

Proof. From illustrations it is clear that the weighted lengths of the paths form the root to the leaves remain the same. Thus the condition RB2 is fulfilled. The only possibility to obtain a violation of condition RB1 is in Case 4 (and thus in Case 5) when the weights of the child edges of the node v are decreased. But then if T' had a violation of condition RB1, T would already have had a violation of the condition RB2. □

We shall show that any sequence of the rebalancing transformations, long enough, will ultimately lead to a red-black tree. This is important since we shall not fix the order in which the transformations are to be applied. If we had to, we had to lock more than a constant number of nodes at a time. We first need the following lemma.

Lemma 2. *Given a chromatic tree that does not satisfy the balance conditions for red-black trees, the tree has at least one node at which a rebalancing transformation can be carried out.*

Proof. The only conflict for which no rebalancing action is possible is a red-red conflict at 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 the latest the root is such a node). In that node one of the transformations of Cases 1, 2, and 3 can be used. □

Let T be a chromatic tree and u its node. By *outside*(u) we denote the number of T's nodes that are not contained in the subtree rooted at u. Let $c = ((v, u), (u, t))$ be a red-red conflict of T. The *red-red distance* of c, $rd(c)$, is defined by $rd(c) = outside(u)$ and the *total red-red distance* in T, $rd(T)$ by

$$rd(T) = \sum_{c \in R(T)} rd(c),$$

in which $R(T)$ is the set of red-red conflicts of T.

If $e = (v, u)$ is an overweight edge, the *overweight distance* $od(e)$ of e is defined by $od(e) = (w(e) - 1) \cdot outside\ (u)$. The *total overweight distance* of T, $od(T)$, is defined by

$$od(T) = \sum_{e \in E(T) \text{ and } w(e) > 1} od(e),$$

in which $E(T)$ is the set of edges of T.

We shall characterize the balance of a tree T by a quadruple $(o(T), od(T), r(T), rd(T))$, in which $o(T)$ is the number of overweight conflicts in T, $r(T)$ is the number of red-red conflicts of T, and $od(T)$ and $rd(T)$ are as defined above. Let $\lessdot$ denote the alphabetic order of the quadruples (that is, $(a', b', c', d') \lessdot (a, b, c, d)$ if $a' < a$ or $a' = a$ and $b' < b$ or $a' = a$, $b' = b$ and $c' < c$ or $a' = a$, $b' = b$, $c' = c$ and $d' < d$). We say that a tree T' is *closer* to a red black tree T, $T' \prec T$, if

$$(o(T'), od(T'), r(T'), rd(T')) \lessdot (o(T), od(T), r(T), rd(T)).$$

Notice that the smallest elements in this relation are red-black trees.

We shall show that whenever a rebalancing transformation is applied in a chromatic tree T it will result a tree T' with $T' \prec T$.

First we show that the red-red conflicts can be removed using the transformation of Cases 1, 2, and 3 without creating new overweight in the tree.

Lemma 3. *Let T be a chromatic tree with at least one red-red conflict, and let T' be a chromatic tree that has been obtained from T by applying one of the transformations of Cases* 1, 2, *and* 3 *of Definition* 3. *Then $T' \prec T$.*

Proof. When an operation as defined in any of Cases 1, 2, and 3 is applied at node v (which is the leading node of the conflict $c = ((v,u),(u,t))$ to be removed) then the number of red-red conflicts will be decreased in the subtree S originally rooted with the node v.

One conflict with leading node v is removed, and if a new red-red conflict c_1 is created within this subtree then another conflict c_2, uniquely depending on c_1, has been removed. Altogether we can conclude that within the subtree S the number of red-red conflicts has decreased.

Some red-red conflicts of the tree may be pushed lower in Cases 2 and 3, but whenever this happens for a conflict $c' = ((v',u'),(u',t'))$ the number of nodes in the subtree rooted at u' does not change.

In Case 1, if the grandparent edge of the leading node is red and the parent edge becomes red a new conflict arises. But this conflict has smaller red-red distance than the removed one. Denote by S' the tree obtained from S by the transformation. If the edge leading to the root of S' is red, we have one new red-red conflict in Cases 2 and 3. However, the red-red distance of this new conflict is smaller than the red-red distance of the removed conflict. Hence we can conclude that either $r(T') < r(T)$ or $r(T') = r(T)$ and $rd(T') < rd(T)$.

As to the overweight conflicts, it is clear that no new ones are created. Some overweight conflicts may be pushed lower in the tree, but for each overweight conflict e in T $od(c)$ is retained in the transformation. Thus $o(T') \leq o(T)$ and $od(T') \leq od(T)$.

We can conclude that $(o(T'), od(T'), r(T'), rd(T')) \lessdot (o(T), od(T), r(T), rd(T))$ and thus $T' \prec T$. □

For Cases 4 and 5 we have:

Lemma 4. *Let T be a chromatic tree with at least one overweight conflict, and let T' be a chromatic tree that has been obtained from T by applying one of the transformations of Cases* 4 *and* 5. *Then $T' \prec T$.*

Proof. The transformation of Case 4 either removes a conflict or pushes overweight upward. Thus in this case $o(T) \leq o(T)$ and $od(T') < od(T)$. Assume then that for the overweighted edge (v,t) with the weight $w((v,t)) > 1$ the transformation of Case 5 is applied. As the result we may have an overweighted edge (v,t) with weight $w((v,t)) - 1$ lower in the tree. That is, $o(T') < o(T)$. Thus in both cases $T' \prec T$. □

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. In Lemma 1 we have seen that we can always apply a transformation in a chromatic tree that is not a red-black tree. In Lemma 2 we have seen that any transformation modifies a tree T to a tree T' with $T' \prec T$. The last tree in the sequence is a tree S with $(o(S), od(S), r(S), rd(S)) = (0,0,0,0)$, which is a red-black tree. □

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