Concurrent Perfect Balancing of Binary Search Trees*

Jürgen Eckerle† and Otto Nurmi‡

Abstract. When a balanced data structure is updated and searched concurrently, updating and balancing should be decoupled so as to make updating faster. The balancing is done by special maintenance processes that run concurrently with the search and update tasks. We show that it is not necessary to use a weak balance condition like *AVL* or *red-black* condition, since balancing a binary tree perfectly so that the search paths become as short as possible is not much more expensive, that is, a process must lock only 5 nodes at a time even when perfect balance is desired. In contrast to other algorithms that perfectly balance a binary search tree, our algorithm keeps the tree (weakly) balanced during the further balancing. This is important if the data structure is used by concurrent search and update processes.

*The work is partially supported by the Academy of Finland and Deutsche Forschungsgemeinschaft.

†Institut für Informatik, Universität Freiburg, Rheinstraße 10-12, D-79104 Freiburg, Germany

‡Department of Computer Science, University of Helsinki, Teollisuuskatu 23, SF-00510 Helsinki, Finland

1. Introduction

In some applications updating and querying a database must be extremely fast. Such applications are, for example, databases of a telecommunication exchange. The databases must be stored in main memory. They are queried and updated concurrently by several processes.

Balanced binary search trees are good search structures for main memory since they allow fast access to data and they are easy to update. Simultaneous writing of the contents of the same node is not desired. A common way to prevent different processes from reading and writing in the same node is to lock the node when it is read or written; other processes must wait until the writing or reading is complete and the node is unlocked. The waiting times should be as short as possible. It is, thus, important to minimize the number of nodes a process have to lock at a time.

The search paths should be short. We say that a binary tree is *perfectly balanced* if the length of the longest and shortest path leading from the root to a leaf differs at most by one and show how a binary tree can be balanced perfectly by a separate maintenance process without disturbing too much the concurrent search and update processes. The process must lock at most 5 nodes at a time. The structure remains as a search tree with height $O(\log n)$ all the time during the further balancing (n is the number of keys stored in the tree) if its initial height was $O(\log n)$.

For efficiency, the updating and searching processes do no rebalancing. They only leave certain information in the tree in the places where a rebalancing transformation may be needed. The maintenance process, the *rebalancer* visits nondeterministically the nodes of the tree, checks whether a balancing transformation is needed in the visited node, and if needed, performs it. There can be several such maintenance processes working concurrently in the tree.

Notice the essential difference between the quality criteria of sequential and concurrent balancing algorithms: In sequential algorithms it is important that only a few transformations are needed and the balancing is completed rapidly while in concurrent balancing the number of transformations has a minor role if any. It is desired that the rebalancing process does not terminate but continues searching possible balance violations forever. The process does not, in general, know whether the tree is balanced or not. It cannot traverse the tree in any (deterministic) order since in that case it had to block all user tasks for a long time. The nodes the rebalancer visits must be unlocked before the rebalancer can enter them. It must wait until these nodes are unlocked, and when locked by the rebalancer, other processes cannot enter these nodes. Thus, it is important that the rebalancer has to *lock as few nodes as possible* at a time and for as short time as possible. It must lock the nodes for inspecting whether a transformation is needed also when no transformation is required, and it does this even if the tree were already balanced.

The separation of updating and rebalancing was proposed already by Guibas and Sedgewick in [9]. Their solution seems to allow only insertions. For *AVL-trees* [1] a solution without deletions was presented by Kessels in [10]. It was completed with deletions in [16]. AVL-condition requires that 5 nodes are locked at a time. A full solution for *red-black* trees was presented in [15], in which 4 nodes must be locked at a time. A solution with different rebalancing transformations was presented by Boyar and Larsen in [3].

The AVL and red-black conditions are weak, that is, they allow the length of a search path be far away from the optimum (thought bounded by $O(\log n)$). If no updaters are active no new balance violations arise. If the tree is balanced (in the weak sense) the rebalancer still continues searching balance violations but finds none. The idea behind our results is to use this "lazy" time of the rebalancer for further shortening the search paths.

Certain problems arising from the concurrency were solved in [15, 16] by using *leaf-oriented search trees* that contain keys only in leaves. As a byproduct of our rebalancing technique, we describe a simple new technique that enables using *internal* search trees in which keys are stored in all nodes. Related techniques can be found in [7, 8, 12, 13].

Other algorithms that modify an arbitrary binary tree to a perfectly balanced tree (see e.g. [4, 5, 14, 17]) are inefficient in a concurrent environment. One of the reasons for this is that they either modify the tree first to a list (called *vine* in [17]), require a traversal of the tree, or destroy the search structure for a while.

In Section 2, we recapitulate the definition of a chromatic tree [15] and a red-black tree [2, 9, 18, 19], and define a new class of binary trees, *perfect red-black trees*, whose members are perfectly balanced. We further define the invariants that our trees satisfy during the balancing. The details of the simple updating operations are explained in Section 3, and our rebalancing transformations are described in Section 4. A concurrency control method for the operations is discussed in Section 5. A method that allows the use of internal search trees (keys stored in internal nodes and in leaves) is shortly discussed in Section 6. Section 7 contains some concluding remarks.

2. Perfect red-black trees

We use weighted trees called *chromatic trees* that are defined in [15]. Since we need a kind of height information in the nodes we modify the definition to Tarjan's terminology [18, 19] in which the information is called the *rank* of a node.

A chromatic tree with certain further conditions is a *red-black tree* [9]. These trees are called *symmetric binary B-trees* in [2] and *balanced trees* in [19]. The height of a red-black tree is $O(\log n)$, where n is the number of nodes ([2, 9, 18]). We call such trees *balanced.* With some stronger balance conditions we get a class of trees we call *perfect red-black trees.* They are binary trees of minimal height as shown in Theorem 2.

Our rebalancing process displaces some subtrees. We introduce a notion of *mail nodes* that are roots of these subtrees. The process *sends mail* by making an ordinary node to a mail node (the corresponding subtree is "sent" to another place), *carries mail* by moving mail nodes (and the corresponding subtrees) stepwise towards their destination, and *receives mail* by returning a mail node to an ordinary node. The ranks serve as "addresses" of the mail nodes and the associated subtrees.

A binary tree is *full* if its nodes have either two or no children. With each node v of a tree we associate a non-negative integer $rank(v)$.

Definition 1. A *chromatic tree* is a full binary tree in which the ranks have the following properties:

CH1: If v is any node with a parent, then $rank(v) \leq rank(w)$, where w is the parent of v.

CH2: If v is a leaf, $rank(v) = 0$, and if w is the parent of v then $rank(w) > 0$.

The definition corresponds to that of [15] in the following way: Let v be a node with parent w. If $rank(w) = rank(v)$, the edge (w, v) is *red*; if $rank(w) = rank(v)+1$, the edge (w, v) is *black;* and if $rank(w) > rank(v)+1$, the edge (w, v) is an *overweight edge.*

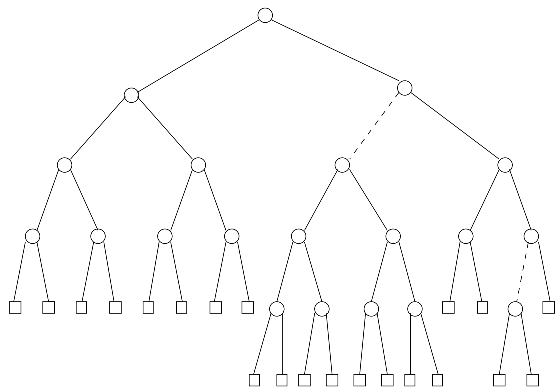

FIG. 1. A perfect red-black tree. Broken lines denote red edges.

The updaters leave a tree as a chromatic tree that may be out of balance. The algorithm of [15] modifies a chromatic tree to a red-black tree:

Definition 2. A full binary tree with the following conditions is a *red-black* tree:

RB1: If v is a leaf, $rank(v) = 0$, and if w is the parent of a leaf, $rank(w) = 1$.

RB2: If v is any node with a parent w, then $rank(w) \leq rank(v) + 1$.

RB3: If v is any node with a grandparent x, then $rank(v) < rank(x)$.

From RB2 it follows that a red-black tree cannot have overweight edges; from RB3 it follows that the tree cannot contain two consecutive red edges in any root-to-leaf path; and from RB2 and RB3 it follows that the number of black edges is the same in each path from a node to a leaf. The height of a red-black tree is $O(\log n)$ where n is the number of its nodes (see [2, 9, 18, 19]).

We call a binary tree *perfectly balanced* if the length of any pair of its root-to-leaf paths differs at most by one. (Notice the difference of our definition and the one of *complete* trees used in a heap structure in which the leaves have to be close to the lower left corner of the tree.) The next definition gives a class of trees that are perfectly balanced red-black trees.

Definition 3. A red-black tree is a *perfect red-black tree* if it contains either no red edges or all its red edges are left child edges of the nodes on the path from the root to the rightmost leaf.

An example of a perfect red-black tree is shown in Fig. 1.

The following theorem tells us that the perfect red-black trees can be used as search structures for any number of keys. In contrast to general red-black trees (and AVL-trees), there is only one perfect tree that contains a given number of nodes.

Theorem 1. Let $n \geq 0$ be an integer. There exist a unique perfect red-black tree with n leaves.

Proof. Let $h > 0$ be an integer. Consider a red-black tree of height h whose all edges are black. The tree contains 2^h leaves. If the left edge of a node v on the rightmost root-to-leaf path is colored red, some nodes must be inserted on the lowest level of the tree so as to get the red-black conditions fulfilled. The number of the new leaves is $2^{rank(v)}$ (here $rank(v)$ is equal to the height of v) but $2^{rank(v)-1}$ leaves of the original tree are changed to internal nodes. Thus, the tree contains $2^h + 2^{rank(v)} - 2^{rank(v)-1} = 2^h + 2^{rank(v)-1}$ leaves.

Let $k = \lfloor \log n \rfloor$ and $b_k b_{k-1} \ldots b_0$ be the binary representation of n. A perfect red-black tree with n leaves can be constructed as follows: Take the red-black tree of height k with only black edges. For each $i = k-1, \ldots, 0$, if $b_i = 1$ color the left edge of its rightmost node v with $rank(v) = i+1$ red and add one level leaves to the subtree pointed by the red edge. Color the father edges of the new leaves black. From the discussion above the tree contains n leaves. The representation is unique since the binary representation of n is unique. $\square$

Since the number of black edges of a red-black tree is the same on any root-to-leaf path and such a path contains at most one additonal red edge in a perfect red-black tree, we have:

Theorem 2. A perfect red-black tree is perfectly balanced.

Our rebalancing process moves red edges of a chromatic tree towards the nodes on the rightmost path. Special marks called *rightmost marks* are kept in the nodes of that path. A special node that will never be removed is kept as the rightmost leaf. The roots of subtrees that are displaced during the rebalancing are marked either by an *up-mail* or a *down-mail* mark. If a node contains an up-mail (down-mail) mark, it is called an up-mail (down-mail) node. Our updating and rebalancing operations are designed so that they keep the invariants of the next definition.

Definition 4. A *mail-tree* is a full binary tree in which the ranks and mail marks have the following properties:

MT1: If v is a mail node, then v is neither the root nor a leaf.

MT2: If v is any node and w is its nearest ancestor that is not a mail node, then $rank(v) \leq rank(w)$.

MT3: If v is any mail node and w is its nearest ancestor that is not a mail node, then $rank(v) < rank(w)$.

MT4: If v is a leaf, $rank(v) = 0$, and if w is the parent of v then $rank(w) > 0$.

MT5: Let v be a node and w its ancestor. If v and w are both up-mail (down-mail, resp.) nodes then a non-mail node is between them.

MT6: If v is a node on the rightmost root-to-leaf path then v has a *rightmost* mark.

Let (v, w) be an edge of a mail tree. If either v or w is a mail node the color of the edge is undefined. For other edges, the color is defined as in chromatic trees.

A nonempty chromatic tree is a mail tree. A mail tree that contains no mail nodes is a chromatic tree, and if it contains no balance violations (overweight edges and consecutive red edges) then it is a red-black tree. If the only red edges are left child edges of the nodes on the rightmost root-to-leaf path, then the tree is a perfect red-black tree.

As in [15, 16], we first assume that all keys are stored in the leaves. The internal nodes contain *routers* that guide the search in the structure. The router stored in a node v must be greater than or equal to any key stored in the left subtree of v and smaller than any key stored in the right subtree of v. The use of leaf search trees make deletions simple but also a successful search must advance down to the leaves. In Section 6. we propose a new method for deletions that enables keys to be stored also in internal nodes.

3. Search and update operations

The ranks and mail marks have no influence to search operations; searching is carried out exactly as in any leaf-oriented search tree (see e.g. [11]).

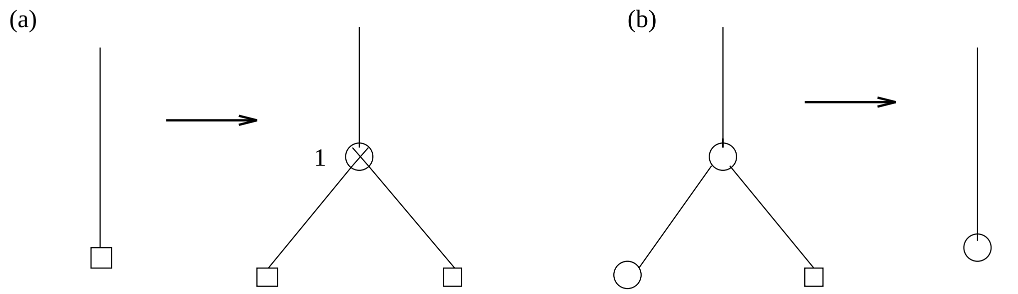

FIG. 2. Update operations. (Only involved nodes are shown; the operations have symmetric variants. A circle denotes any node; a square denotes a leaf. The ranks of the nodes are shown on the right of them.) (a) Insertion: an internal node and a leaf are inserted; if the right child of the internal node has a *rightmost* mark then the new internal node is marked, too. (b) Deletion: a leaf and an internal node are removed.

The updating operations are as in [15]. They are designed so that they keep the invariants of a chromatic tree. By a trivial modification (marking the nodes on the rightmost path) the invariants of a mail tree will also be kept. The details of the update operations are:

Insertion: The new key is searched from the tree. If found, the process terminates. Otherwise, the search ends up in a leaf u. Since the tree must be a full binary tree an insert operation adds an internal node v and a leaf w into the tree. u and w will be the children of v and v's parent will be original parent of u. The internal node is marked with a *rightmost* mark if u has a mark. The ranks are assigned as shown in Fig. 2. The children of the new internal node are ordered such that the one containing the smaller key is the left child. The router of the new internal node is a copy of the key in its left child.

Deletion: The key is searched from the tree. If it is not found, the operation terminates. Otherwise, the leaf containing the key is removed and its parent is replaced with the parent's other child. The ranks of the remaining nodes need not be changed. Since the rightmost leaf is a special node that will never be removed, the *rightmost* marks of the remaining nodes need not be updated.

An insertion may produce a red edge, and a sequence of them may produce consecutive red edges. A delete operation may produce an overweight edge.

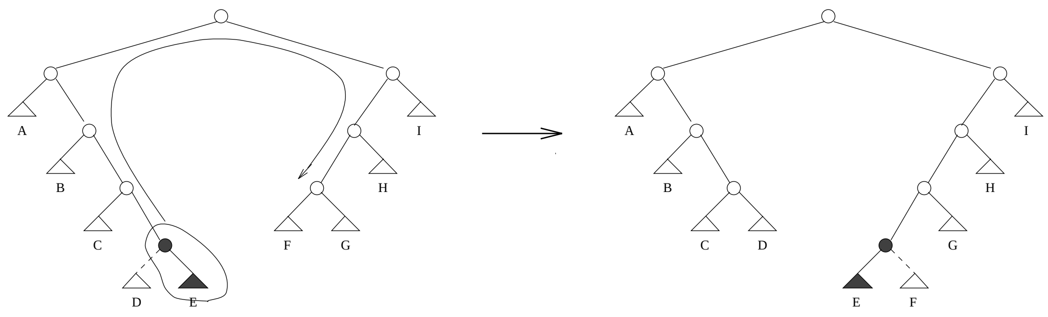

FIG. 3. A large rotation.

These possible violations of (perfect) red-black conditions are later removed by the rebalancing process.

4. Rebalancing transformations

We describe now the rebalancing transformations that modify a mail tree towards a perfect red-black tree. The overweight edges are removed as in [15]. The red edges are either removed or moved stepwise towards the rightmost root-to-leaf path.

Balancing a binary tree perfectly requires, in general, large rotations involving more than a constant number of nodes. A separate rebalancing process has time to do such rotations. Since the involved nodes must be locked (see Section 5.), the large rotations must be split into small pieces involving only a constant number of nodes at a time. We have split them into steps consisting of single rotations and forwarding the balance information (updating the ranks) and changing routers in some nodes. Between such small steps the nodes are unlocked and the search and update processes are free to advance in the tree. The operations are designed so that when applied to a red-black tree, the height of the tree will remain $O(\log n)$ where n is the number of stored keys.

To get an intuitive idea how the rebalancing advances let us assume that the tree is a red-black tree, (v, w) is its only red edge, and the edge is not in its final position. Let x be the successor of v in the symmetric order (see [11]) of the nodes. The rebalancer first marks v with an up-mail mark, and moves v and v's right subtree stepwise upwards until v is the left child of x.

At that point v is moved to the right so that it becomes x's right child. The former right subtree of v is now its left subtree. The routers stored in v and x are swapped, and v is marked with a down-mail mark. Then what follows is a concatenate operation (see [19]) of v's right subtree (including v) and the left subtree of v. The node v is moved downwards until it is the left child of a node z that has $rank(z) \leq rank(v)$. Then v is made again to an ordinary node. This may color an edge red. The new red edge stays to the right of the one which disappeared as v was first time moved upwards. We call the displacing of v a *large rotation* (see Fig. 3). The process moves the red edge towards the rightmost root-to-leaf path by repeating such large rotations. If a red edge appears on the rightmost path, it is moved to its final position by a single left rotation. Then the tree is a perfect red-black tree.

The rebalancing process searches possible balance violations, i.e. overweight edges and red edges. Whenever an overweight edge is encountered the process decreases the overweight or moves it towards the root (where the overweight disappears) by one of the transformations defined below (Case 1). When a red edge is encountered either the ranks are adjusted (cf. [18]), or one of the nodes incident to the red edge is made to an up-mail node and a large rotation is started.

The details of the transformations are divided in 5 cases according to the node the rebalancer has encountered. In all of them, a transformation will not be carried out if it would lead to a situation in which a descending path would contain three consecutive mail nodes or two consecutive mail nodes of the same type. This condition must be checked before performing any of the transformations defined below. It can easily be done by inspecting a few nodes around the place where the transformation should be performed. For brevity, we have left it away from the description of the operations.

Case 1. (Overweight edge.) The rebalancer has encountered a node that has an overweight edge as a child edge. If one of the children is a mail node then no transformation apply. Otherwise, the transformation depends on the color of its other child edge. If the other child edge is not red, then a demotion ([18, 19]) is performed as depicted in Fig. 4(a). Otherwise, the transformation is a single rotation followed by a demotion (Fig. 4(b)).

Case 2. (Red edge: send mail.) The rebalancer has encountered a node v with a red child edge. Assume v contains no *rightmost* mark (see Case 5). No rebalancing action is carried out at v if v's parent edge is red or if v's parent is a mail node, v's both child edges are red, and the rank of the parent

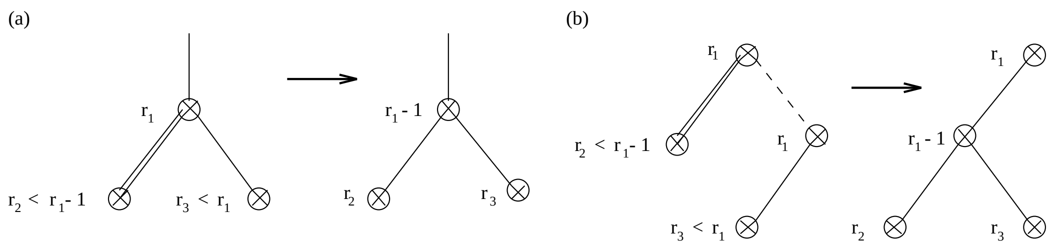

FIG. 4. Rebalancing transformation in Case 1 (Overweight edge). (The symbol $\otimes$ denotes a node that has no mail mark; a broken line denotes a red edge; a double line denotes an overweight edge. The ranks are shown on the left side of the nodes.) (a) A demotion. (b) A rotation followed by a demotion.

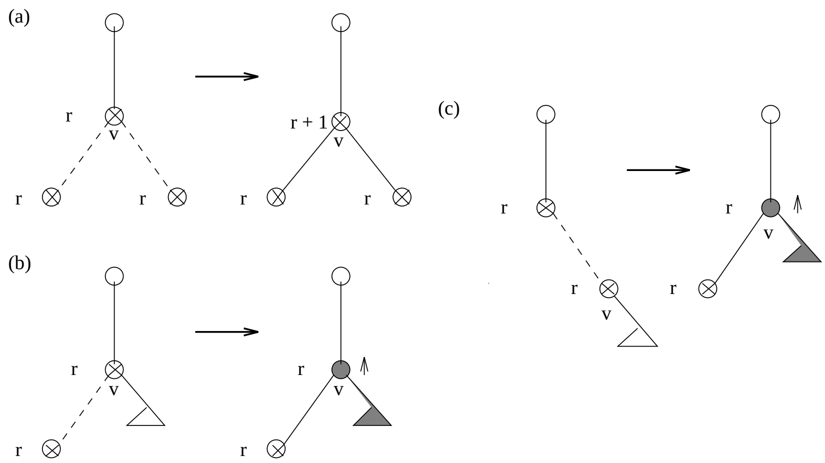

FIG. 5. Rebalancing transformations in Case 2 (Red edge). (A mail node is denoted by a black circle with an arrow on its left side that denotes the direction of the mail. The corresponding subtree is denoted by a black triangle. For other denotations, see Fig. 4.) (a) A promotion is carried out. (b) The node is marked with an up-mail mark. (c) The right son of the node is marked with an up-mail mark and a simple rotation is performed.

is smaller than $rank(v)$. Otherwise, if v's both child edges are red, then a promotion is carried out (see Fig. 5(a)). If the red edge is v's left child edge, v will be marked with an up-mail mark (see Fig. 5(b)). Finally, if the red edge is v's right edge, then v's right child is marked with an up-mail mark and moved one step upwards by a single rotation (see Fig. 5(c)).

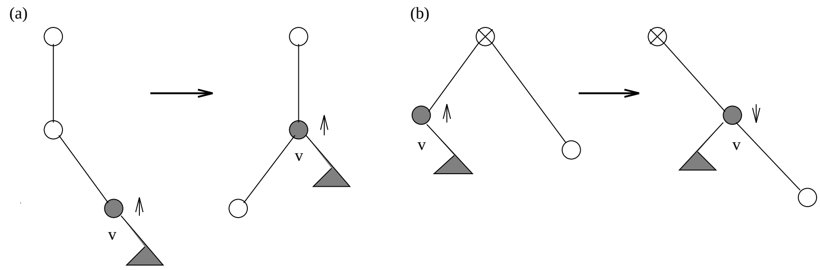

FIG. 6. Rebalancing actions in Case 3 (Up-mail node). (a) The up-mail node v is moved higher by a single rotation. (b) The up-mail node v is moved to the right and changed to a down-mail node. The routers of v and its parent are swapped.

Case 3. (Up-mail node: carry mail.) Let v denote the up-mail node that the rebalancer has encountered. First, if v is the right child of its parent, then v is moved higher by a single rotation (see Fig. 6(a)). Let v then be the left child of its parent. In this case v has reached its topmost position. It will be moved to the right by a single rotation. For simplicity of the explanation, we assume that the root node of the subtree is not changed but v has been moved so that it is the right child of the root. The former right subtree of v is the left subtree of v after the rotation (see Fig. 6(b)). The keys of the root of the subtree and v are swapped. This will save the symmetric order of the routers. Finally, v is marked as a down-mail node, and it starts its way downwards in the tree.

Case 4. (Down-mail node: carry mail or receive mail.) The rebalancer has encountered a down-mail node v. The action depends on the right child of v. Let us first assume that v's right child is an up-mail node or the child has greater rank than v. Then we move v downwards by a single rotation (see Fig. 7(a)). Let us then assume that the rank of v's right child is smaller than or equal to $rank(v)$ and that the child is not a mail node. If the rank of the right child is *smaller* than $rank(v)$ then the mail mark v is removed. If the rank of the right child is *equal* to $rank(v)$ and neither one of the child edges of v's right child are red, nor one of the children of v's right child is a mail node, then the mail mark of v is removed, too (see Fig. 7(b)). If the rank of the child was equal to $rank(v)$, the operation colors an edge red. That edge is closer to the rightmost path than the one that disappeared when the mail node was created.

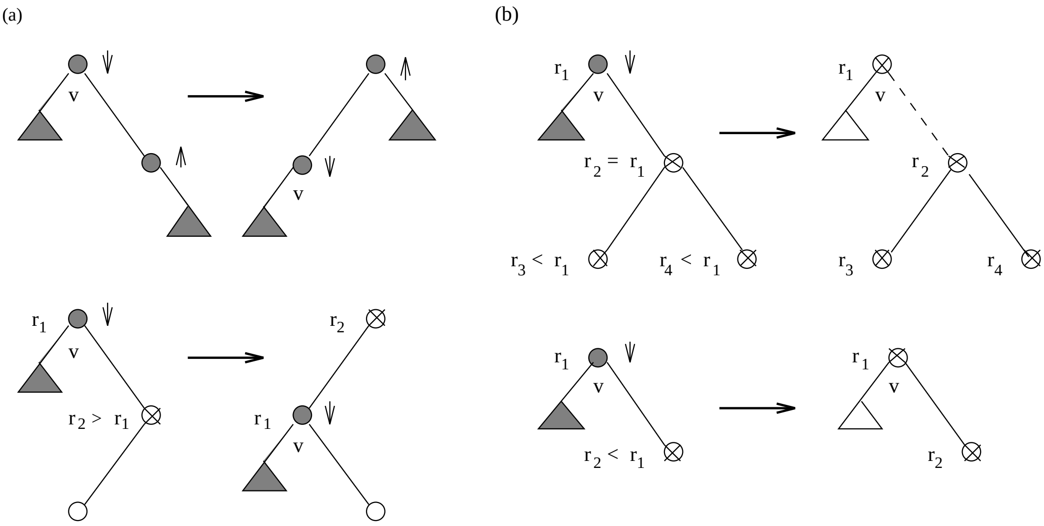

FIG. 7. Rebalancing transformations in Case 4 (Down-mail node). (a) The down-mail node v is moved downwards by a single rotation. (b) The mail mark of v is removed.

Case 5. (Special cases close to the rightmost path.) The red edges of the rightmost root-to-leaf path are removed by promotions and simple or double rotations. Let us assume, that the rebalancer has encountered a node v that contains an *rightmost* mark. If v's parent edge is red, no transformation is performed at v. Otherwise (v is the root or v's parent edge is not red), we have several subcases. First, if v's right child edge is red, its left child edge is not red, and no child edge of v's right child is red, then a single left rotation is performed (see Fig. 8(a)). If both of v's child edges are red, then a promotion is carried out as in Case 2 (see Fig. 5(a)). If v starts a descending path with two consecutive red edges, a single or a double rotation followed by a promotion is carried out (see Figs. 8(b) and 8(c)). If v is a mail node, the mail mark is removed. This may introduce a descending path with two consecutive red edges. In that case, a single or a double rotation followed by a promotion is immediately carried out.

For the correctness of the transformations, we first prove that whenever the tree is not a perfect tree, it contains at least one node where a transformation can be carried out. Since we do not fix the order in which the rebalancer traverses the tree, we have still to show that even when the nodes

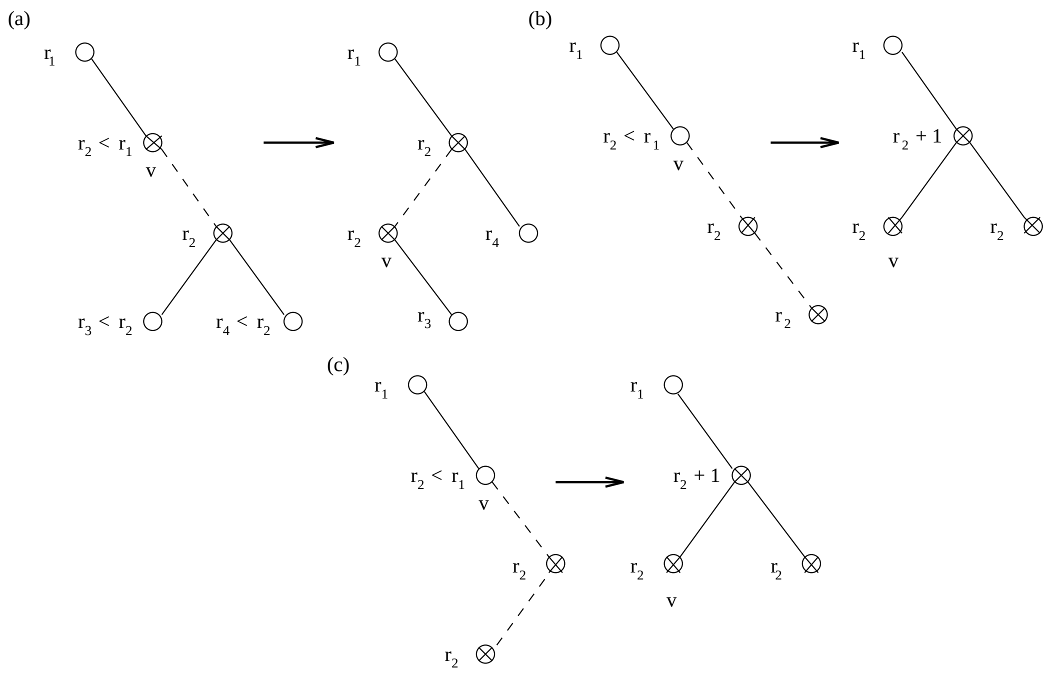

FIG. 8. Rebalancing transformations of Case 5 (Special cases close to the rightmost path). The subcases have reflections that are not shown. (a) A single rotation (b) A single rotation followed by a promotion. (c) A double rotation followed by a promotion.

are chosen randomly, we will finally get a perfect red-black tree.

We call a sequence of large rotations that moves a red edge to its final position a *sequence of large rotations*. The sequence of parent nodes that a node v has during a sequence of large rotations is called v's *parent sequence*. A node may appear in the sequence at most 3 times.

Theorem 3. Let T be a mail tree that is not a perfect red-black tree. Then T has at least one node at which a rebalancing transformation can be carried out.

Proof. We have three cases. Assume first that T has neither mail nodes nor overweight edges. Then it contains at least one red edge that is not in its final position. Let v be the parent node of the topmost red edge in T. We can apply either a transformation of Case 2 or Case 5 at v. If T contains overweight edges but no mail-nodes, then the overweight can be decreased by a transformation of Case 1. Assume finally that T contains mail-nodes. Let v be the mail-node v that has the minimal rank. Let w denote the last mail node in v's parent sequence (or v if the sequence contains no mail nodes). In w one can readily apply a transformation of Case 3, Case 4, or Case 5. □

If the tree contains no overweight, new red edges can be created only by promotions or when a mail node is made to an ordinary node. In a promotion, two old red edges disappear, and thus the number of red edges decreases. Let us consider a node with only one red child edge. We associate the left red edge to its parent node and the right red edge to its child node. We can now imagine that a mail node "contains" a red edge and consider how the red edges move during the rebalancing.

The operations do not change the symmetric order of the keys and routers. We show that a finite number of our transformations will move a mail node (and the red edge) at least one step forward in the symmetric ordered sequence of keys and routers.

Let n be the number of nodes in the tree. A mail node is moved upwards until a point where the node is moved to the right and the routers are swapped (Case 3). This path may contain, in the worst case, all n nodes of the tree, but since two up-mail nodes cannot pass each other, any node can be passed only once. When routers are swapped, the mail node (and the red edge) moves forward in the symmetric sequence. Then the node starts its way downwards. The path downwards does not contain more than n nodes since two down-mail nodes do not pass each other. When the node has reached its bottommost place the mail mark is removed. This may create a new red edge that is associated to its parent or child.

From the definition of our operations we see that they either move a mail node, mark a node with the up-mail mark, or remove a down-mail node. Since the marks are swapped only when the node has been moved to the right in the symmetric sequence, the red edge has been moved forward before the mail mark is removed.

From the discussion above and Theorem 3 we get:

Theorem 4. Any sequence of rebalancing transformations that is long enough, will modify a mail tree to a perfect red-black tree.

The next theorem tells us that searching is efficient also during the rebalancing.

Theorem 5. Let T be a red black tree with n nodes and T' a mail tree that is obtained from T by a sequence of rebalancing transformations. Then the height of T' is $O(\log n)$.

Proof. Since a rebalancing transformation that could cumulate red edges in a root-to-leaf path is not carried out, and since any such path may contain at most two consecutive mail nodes, the length of the longest root-to-leaf path cannot be longer than 3 times the length of any root-to-leaf path of a red-black tree with n nodes. The length of the longest root-to-leaf path of a red-black tree with n nodes is $O(\log n)$. □

5. Concurrency control

We present a simple locking strategy for our trees. It prevents simultaneous writing and reading the same data location but still allows a high degree of concurrency. The strategy is developed by augmenting the one of [15, 16] that is based on the one of Ellis [7, 8].

In [15] a strategy for concurrency control is called *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 [7, 8, 12, 13, 15, 16] 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 [7, 8, 15, 16] we use three kinds of locks, which we call, according to [15], *r-locks, w-locks,* and *x-locks* (they are called ρ-, α-, and ξ-locks in [7, 8]). 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.

The 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. The nodes are always locked in the top-down direction in order to avoid dead-lock situations.

As in [15], 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. The process keeps at most two locks in the tree at a time (for details, see[15]).

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, and performs the deletion as in [15]. At most 3 nodes must be locked 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 ranks and marks must be investigated. In the worst case it must w-lock 5 nodes (Case 5(a)). 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. Thus, 5 nodes must be w-locked and 4 nodes must be x-locked in the worst case. If a rebalancing process decides that the chosen rebalancing transformation does not apply, it releases immediately all locks and continues searching for other conflicts.

The correctness of the concurrency control follows (as in [15]) form the facts that the nodes are locked only in top-down order, and that the nodes whose contents are to be modified are x-locked during the modification.

6. Internal search trees

We have assumed that the trees are full binary trees and that they are used as leaf-oriented search trees. By using the idea of separate maintenance processes we sketch a technique that enables the use of internal search trees in which keys are stored as well in the internal nodes as in the leaves.

The difficulty in the use of internal search trees in a concurrent environment is that a delete operation must lock more than a constant number of nodes after removing a key from an internal node. The deleted key must be replaced with its symmetric predecessor that may be far away from the node. All nodes between these nodes should be locked.

Instead of immediately removing the deleted key, we perform a weak deletion, that is, the key is only marked as deleted. Then the symmetric predecessor is searched, and when found, the predecessor is marked with a *key-mail* mark. The maintenance process moves the marked node stepwise towards the (weakly) deleted key and finally replaces it with the predecessor contained in the marked node. The nodes are unlocked between these steps.

Let us assume that the maintenance process has found a key-mail node. If the node is the right child of its parent, the process moves it upwards by a single left rotation. If the node is a left child, then the key is stored into the parent, and the node is removed. This is an easy task since the node is always unary. The search processes are modified so that they also inspect whether the found keys are marked as deleted.

The key-mail nodes have initially at most one child. If they would get a right child by an insert operation, the new key is stored in the key-mail node and the insert operation proceeds further with the key that was earlier in the mail node. Thus the key-mail nodes remain unary.

The maintenance process must x-lock at most 3 nodes at a time. The delete operation uses w-lock coupling when advancing downwards in the tree. Just before marking a node the w-locks are changed to x-locks. The insert operation must x-lock a node when it changes its contents.

At most each other node along a search path may be a key-mail node. Thus the influence of key-mail nodes to the searching time is only a constant factor. Since nodes are not removed immediately when a key is deleted, the tree may contain more nodes than keys. However, since nodes are inserted only when keys are inserted, we have:

Theorem 6. Let T be a balanced internal binary search tree with n keys. The height of T is $O(\log n)$ after any sequence of delete operations defined above.

7. Conclusions

We have presented a method to balance a binary search tree perfectly. The method allows updating and searching the tree concurrently with the balancing. The balancing task is left for a separate process that performs maybe several *local* modifications in the tree. The tree can temporarily be out of balance after updates as in [15, 16]. When applied to a (weakly) balanced tree, the tree remains (weakly) balanced during the balancing. Thus, the further balancing does not have a considerable influence on the efficiency of searching. The operations defined in [15] that modify a tree to a red-black tree, can be used together with our transformations. One can give a higher priority to them such that the tree will first become weakly balanced, and then our operations will slowly improve the balance until the tree will become optimal.

We have split the 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. Boyar and Larsen [3] could decrease the number of necessary transformations by combining the transformations of [15] into larger pieces with the cost that more nodes need to be locked at a time. We suppose that the number of our operations might be decreased in a similar way, but it is questionable whether that would improve the performance of the system.

Problems arising if keys are stored in internal nodes are avoided in [15, 16] by using leaf-oriented search trees in which the routing information of the internal nodes need not be keys present in the structure. For the efficiency of a successful search, the keys should be stored in internal nodes, too. Our solution is, following the idea of separated rebalancing, to let a separate maintenance process to solve these problems.

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. J. Boyar and K. S. Larsen, Efficient rebalancing of chromatic search trees. *Proc. 3rd Scandinavian Workshop on Algorithm Theory (SWAT '92),* LNCS vol. 621, Springer-Verlag, 1992, 151–164.
4. H. Chang and S. S. Iyengar, Efficient algorithms to globally balance a binary search tree. *Commun. ACM* **27** (1984), 695–702.
5. A. C. Day, Balancing a binary tree. *Comput. J.* **19** (1976), 355–361.
6. 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. *Commun. ACM* **21** (1978), 699–975.
7. C. S. Ellis, Concurrent search in AVL-trees. *IEEE Trans. Computers* **C-29** (1980), 811-817.
8. C. S. Ellis, Concurrent search and insertions in 2–3 trees. *Acta Inform.* **14** (1980), 63–86.
9. L. J. Guibas and R. Sedgewick, A dichromatic framework for balanced trees. *Proc. 19th IEEE Symp. Foundations of Computer Science,* 1978, 8–21.
10. J. L. V. Kessels, On-the-fly optimization of data structures. *Comm. ACM* **26** (1983), 895–901.
11. D. E. Knuth, *The Art of Computer Programming, Vol.3: Sorting and Searching.* Addison-Wesley, Reading, Mass., 1973.
12. H. T. Kung and P. L. Lehman, A concurrent database manipulation problem: Binary search trees. *ACM Trans. Database Syst.* **5** (1980), 339–353.
13. U. Manber and R. E. Ladner, Concurrency control in a dynamic search structure. *ACM Trans. Database Syst.* **9** (1984), 439–455.
14. W. A. Martin and D. N. Ness, Optimal binary trees grown with a sorting algorithm. *Commun. ACM* **15** (1972), 88–93.

15. O. Nurmi and E. Soisalon-Soininen, Uncoupling updating and rebalancing in chromatic binary search trees. *Proc. 10th ACM Symp. Principles of Database Systems*, 1991, 192–198.

16. O. Nurmi, E. Soisalon-Soininen, and D. Wood, Concurrency control in database structures with relaxed balance. *Proc. 6th ACM Symp. Principles of Database Systems*, 1987, 170–176.

17. Q. F. Stout and B. L. Warren, Tree rebalancing in optimal time and space. *Comm. ACM* **29** (1986), 902–908.

18. R. E. Tarjan, Updating a balanced search tree in $O(1)$ rotations. *Inf. Proc. Lett.* **16** (1983), 253–257.

19. R. E. Tarjan, *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, Pa., 1983.