Relaxed Balanced Red-Black Trees

S. Hanke[1], Th. Ottmann[2], and E. Soisalon-Soininen[3]

[1] Institut für Informatik, Universität Freiburg, Am Flughafen 17, D-79110 Freiburg e-mail: hanke@informatik.uni-freiburg.de

[2] Institut für Informatik, Universität Freiburg, Am Flughafen 17, D-79110 Freiburg e-mail: ottmann@informatik.uni-freiburg.de

[3] Laboratory of Information Processing Science, Helsinki University of Technology, Otakaari 1 A, FIN-02150 Espoo, Finland. e-mail: ess@cs.hut.fi

Abstract. *Relaxed* balancing means that, in a dictionary stored as a balanced tree, the necessary rebalancing after updates may be delayed. This is in contrast to *strict* balancing meaning that rebalancing is performed immediately after the update. Relaxed balancing is important for efficiency in highly dynamic applications where updates can occur in bursts. The rebalancing tasks can be performed gradually after all urgent updates, allowing the concurrent use of the dictionary even though the underlying tree structure is not completely in balance. In this paper we propose a new scheme of how to make known rebalancing techniques relaxed in an efficient way. The idea is applied to the red-black trees, but can be applied to any class of balanced trees. The key idea is to accumulate insertions and deletions such that they can be settled in arbitrary order using the same rebalancing operations as for standard balanced search trees. As a result it can be shown that the number of needed rebalancing operations known from the strict balancing scheme carry over to relaxed balancing.

1 Introduction

A *dictionary* is a scheme for storing a set of data such that the operations *search*, *insert*, and *delete* can be carried out efficiently. Standard implementations of dictionaries using balanced search trees like red-black trees, AVL-trees, half-balanced trees, and others presuppose that each update operation is followed by a sequence of rebalancing steps which restore the respective balance condition. Maintaining the balance conditions assures that the trees cannot degenerate into linear lists and search and update operations can be performed in a number of steps which is always logarithmic in the number of keys stored in a tree.

In a concurrent environment, however, uncoupling the updating (insertion and deletion) from the rebalancing transformations may increase the possible amount of concurrency and speed up updates considerably. This leads to the notion of *relaxed balance*. Instead of requiring that the balance condition is restored immediately after each update operation the actual rebalancing transformations can be delayed arbitrarily and interleaved freely with search and update operations.

Relaxed balancing was first suggested in [8] for red-black trees. The first actual solution, presented by Kessels [10], is for relaxed balancing in AVL-trees [1] where the allowed updates are only insertions. Nurmi et al. [14] extend the work of Kessels to the general case in which deletions are also allowed. In [11] the solution of [14] is analyzed, and it is shown that for each update operation in a tree with maximum

size n, $O(\log n)$ new rebalancing operations are needed. Relaxed balanced B-trees are introduced in [14] and further analyzed in [12]. In [13] Nurmi and Soisalon-Soininen propose a relaxed version of red-black trees which they call a *chromatic tree*. Boyar and Larsen [5] analyze the proposal of [13] and show that after a minor modification the number of rebalancing operations per update is $O(\log(n+i))$, if i insertions are performed on a tree which initially contains n leaves. Boyar et al. [3] prove for a slightly modified set of rebalancing operations that only an amortized constant amount of rebalancing is necessary after an update in a chromatic tree. In [21] Soisalon-Soininen and Widmayer propose a relaxed version of AVL-trees which fulfills, despite the local nature of its operations, some global properties. For example, they show that in their solution no rotation will be performed if the underlying search tree happens to fulfill the AVL tree balance condition before any rebalancing has been done.

Except for the recent paper [21], all previous solutions are not wholly based on the standard balancing transformations but require a large number of different new transformations.

In this paper we propose a new technique of how to make known rebalancing algorithms relaxed in an efficient way. We show that essentially the same set of rebalancing transformations as used in the strict case can also be used for the relaxed case, and that the number of needed rebalancing operations known from the strict balancing scheme carry over to relaxed balancing.

In order to illustrate the key ideas and to clarify the ideas underlying our solution as much as possible, we restrict ourselves to the case of red-black trees. But we emphasize that the ideas of marking items for deletions, allowing trees to grow randomly below the balanced part, and to settle accumulated update and rebalancing requests in top-down manner using the standard rebalancing operations carry over to many other classes of search trees as well. The aim of our proposal is to extend the constant-linkage-cost update algorithm for red-black trees [19] in such a way that updates and local structural changes are uncoupled and may be arbitrarily delayed and interleaved. A key idea in our solution is the assumption that the deletion of a key in a tree leads to a *removal request* only; the actual removal of a leaf is considered to be a part of the structural change to restore the balance condition. In this way we put completely aside the problem of deletion which has complicated the previous solutions of relaxed balancing.

2 Red-black trees

The trees in this paper are leaf-oriented binary search trees, which are full binary trees (each node has either two or no children). The nullary nodes of a tree are called the external nodes or leaves while the other nodes are said to be internal nodes. We assume that the keys (chosen from a totally ordered universe) are stored in the leaves of the binary tree. The internal nodes contain routers, which guide the search from the root to a leaf.

For simplicity, we do not distinguish between search trees with stored items (in left-to-right order at the leaves) and their underlying graph structure.

A binary search tree is *balanced* if its height is bounded by $O(\log N)$, where N is the number of keys stored in the tree.

In the following we consider the variant of red-black trees proposed by Sarnak and Tarjan [19]. The nodes of the tree are coloured red or black, respectively. As the balance condition it is required that

1. each search path from the root to a leaf consists of the same number of black nodes,
2. each red node (except for the root) has a black parent, and
3. all leaves are black.

We give now a short review of the update algorithms for strictly balanced red-black trees. Then we will explain in Section 3 how to realize relaxed balancing by using the same rebalancing transformations.

The rebalancing transformations of red-black trees need only a constant number of structural changes (at most two rotations or a rotation plus a double rotation) to restore the balance condition after an update operation.

2.1 Insertions

In order to insert a new key x into a strictly balanced red-black tree we first locate its position among the leaves and replace the leaf by an internal red node with two black leaves. The two leaves now store the old key (where the search ended) and the new key x.

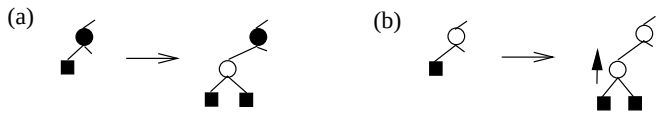

Fig. 1. Insertion of a new item and call of the rebalancing procedure $up\text{-}in$ (denoted by "↑"). Filled nodes denote black nodes and nodes without fill denote red nodes.

If the parent of the new internal node p is red (as well as p itself) then the resulting tree is no longer a red-black tree. In order to correct this and to restore the balance condition we call the rebalancing procedure *up-in* for the new inner node p, cf. Figure 1.

Whenever the rebalancing procedure *up-in* is called for a node p then p is a red node. Note that if p's parent is red as well then the grandparent of p must be black (if it exists). The task of the rebalancing procedure is to achieve that p obtains a black parent while the number of black nodes on any search path from the root to a leaf is not changed. For this the *up-in* procedure flips the colours of some nodes in the immediate vicinity above p and

1. either performs a structural change (at most one rotation or double rotation) involving a few nodes occuring in the immediate vicinity above p in order to restore the balance condition and stops, cf. Figure 2a–d,

2. or (exclusively) calls itself recursively for p's grandparent and performs no structural change at all, cf. Figure 2e.

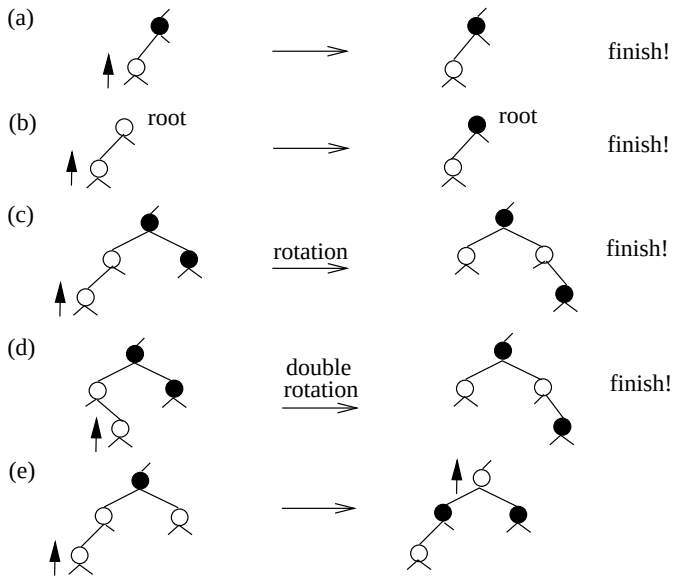

Fig. 2. Local structural changes by the rebalancing procedure *up-in*

For a node p with an up-in request Case 2 above applies if and only if p's parent and the sibling of p's parent are both red.

2.2 Deletions

In order to delete a key from a strictly balanced red-black tree, we first locate the leaf where the key is stored and then remove the leaf together with its parent. (Note that the balance condition implies that the remaining sibling of the leaf is either a leaf as well or a red node which has two leaves as its children.) If the removed parent was black then the red-black tree structure is now violated. It can be restored easily if the remaining node is red (change its colour). Otherwise, the removal leads to the call of the rebalancing procedure *up-out* for the remaining leaf, cf. Figure 3.

Whenever the rebalancing procedure *up-out* is called for a node p then p is a black node and each search path to a leaf of the subtree with root p has one black node too few. The task of the procedure *up-out* is to increase the black height of the subtree rooted at p by one. In order to achieve this the *up-out* procedure changes the colours of some nodes in the immediate vicinity beside and above p and

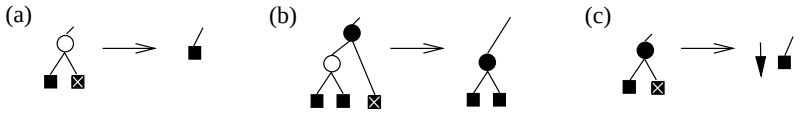

Fig. 3. Deletion of an item (marked by "$\times$") and call of the rebalancing procedure $up\text{-}out$ (denoted by "$\downarrow$")

1. either performs a structural change (at most two rotations or a rotation plus a double rotation) involving a few nodes occuring in the immediate vicinity beside and above p in order to restore the balance condition and stops, cf. Figure 4a–d,
2. or (exclusively) calls itself recursively for p's parent and performs no structural change at all, cf. Figure 4e.

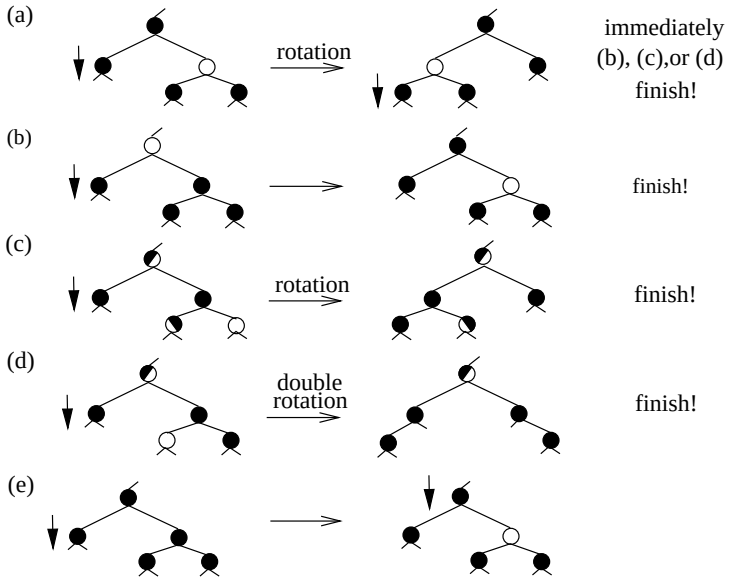

Fig. 4. Local structural changes by the rebalancing procedure *up-out*. Half filled nodes denote nodes that are either black or red.

For a node p with an up-out request Case 2 above applies if and only if p's parent, p's sibling, and both children of p's sibling are all black.

3 Relaxed balancing

In this section we show how to realize relaxed balancing by using the same rebalancing transformations as in the case of strict balancing.

We first uncouple the real insert and delete operations (i.e. searching of a leaf and the insertion or removal of an item, respectively) from the rebalancing procedures *up-in* and *up-out*. Instead of calling the rebalancing procedure immediately after the update operation we only deposit an *up-in request* "↑" or an *up-out request* "↓", respectively, at the corresponding node as shown in Section 2 and do nothing.

In relaxed balancing we allow to interrupt the restructuring procedure after each single restructuring step. We also allow to carry out several of single restructuring steps concurrently at different locations in the tree. The only requirement will be that no two of them interfere at the same location. In order to make this more precise we first specify what we mean by a one-step restructuring at a node p with an up-in or an up-out request: To handle such a request means either to settle it by a local structural change and halt, as shown in Figure 2a–d (Figure 4a–d), or to shift the request to the grandparent (parent) and handle it at a later time, as shown in Figure 2e (Figure 4e).

Now we define a relaxed balanced red-black tree as follows.

A *relaxed balanced red-black tree* is a full binary tree with red and black nodes where the red nodes may have up-in requests, the black nodes may have up-out requests, and as the relaxed condition it is required that

1. For each search path from the root to a leaf, the sum of the number of black nodes plus the number of up-out requests is the same,
2. each red node (except for the root) has either a black parent or an up-in request, and
3. all leaves are black.

The idea here is that each one-step rebalancing operation keeps the tree as a relaxed balanced red-black tree, and gradually transforms it towards a red-black tree. As we will see in the case of accumulated insertions the relaxed balance condition can be guaranteed very easily. But in the case of accumulated deletions we run into many problems if we allow that arbitrary leaves can be removed together with their (possibly black) parents. (Recall that the rebalancing procedure *up-out* increases the black height of a subtree rooted at a black node with an up-out request only by one.)

This leads to the simple but crucial idea of our balancing scheme. Instead of removing a leaf together with its parent immediately after locating it among the leaves we only deposit a *removal request* "$\times$" at that leaf. That is, the leaf is not physically removed but only marked to be deleted later after eventually conflicting up-in or up-out requests have bubbled up in the tree. The actual removal is considered to be part of the rebalancing transformation. To handle such a removal request is only allowed if thereby the relaxed balance condition is not violated. Therefore, we may have to handle or bubble up conflicting requests first, before we can actually remove the leaf with the removal request.

It should now be obvious how to settle a removal request. If the removal request applies to a leaf which has a red parent or a red sibling it can easily be settled (remove the leaf together with its parent and, if necessary, colour the sibling black). Otherwise, the removal of the leaf leads to an up-out request. This is as depicted in Figure 3.

3.1 Interleaving updates

As we have seen, a single insertion replaces a leaf by a red node with two leaves as its children and may lead to an up-in request for the new red node. Now observe that we can do the same also in the case when an (accumulated) insertion falls into a leaf which has a red parent with an up-in request as the result of a previous insertion. In this way, a sequence of insertions may lead to a growth below the red-black tree as in the case of natural (unbalanced) binary search trees. Each of the newly created internal nodes with a red parent has an up-in request attached to it.

If an insertion falls into a leaf which has a removal request and/or an up-out request, then if the leaf has a removal request abandon the removal request and (re-)insert the key at that leaf. Otherwise, if the leaf has an up-out request remove the up-out request and replace the leaf by an internal black node with two leaves.

Several deletions can be accumulated by depositing a removal request at a leaf for each single deletion. Only if a deletion falls into a leaf which has a red parent with an up-in request as the result of a previous insertion then we delete the leaf immediately and abandon the up-in request.

3.2 Concurrent handling of rebalancing transformations

First we observe that several removal requests can be settled concurrently in an arbitrary order as long as it is assured that no two of them interfere at the same nodes. So, a removal request for a node which itself or the parent of which has an up-out request—as the result of a previously settled removal request—cannot be handled before the up-out request has been settled or bubbled up in the tree. This assumption assures that two removal requests appended to both leaves of a black node are always handled correctly.

Analogously, if the sibling of a leaf with a removal request has an up-in request then the up-in request must be settled or bubbled up in the tree before the removal request can be handled.

Now observe that several up-in and up-out requests can also be settled concurrently in an arbitrary order as long as it is assured that no two of them interfere at the same nodes. If we assure that a request is never shifted into the (large enough) surrounding area of another request, it is always possible to carry out the one-step rebalancing operations correctly. If a transformation shifts a request to an ancestor node occurring in the surrounding area of another request, this transformation cannot be carried out before the other request has been settled or bubbled up in the tree. That means, the conflict of two requests can simply be avoided by handling requests in top-down manner if they would meet in the same area, otherwise, in arbitrary order. There is always a request that can be carried out, since the topmost of those requests can be settled without any interference of the other ones.

In this way we get a simple algorithm for relaxed balancing by expanding the standard balancing technique. Now we show how to tune this algorithm for relaxed balanced red-black trees by choosing the surrounding area as small as possible. As result we will obtain the rebalancing transformations for chromatic trees proposed by Boyar et al. [3].

Let the *vicinity* of an up-in or an up-out request of a node p denote the set of neighbouring nodes the colours of which have to be changed or which will obtain new subtrees by the transformation explained in Section 2 that settles the request of p. (Note

that in accordance with our definition nodes that must be considered in order to decide which kind of transformation applies, but which are not affected by the transformation itself, do not lie in the vicinity of p's request.) A request of a node p *is in conflict with* a request of a node q if q lies in the vicinity of p's request.

Obviously, the rebalancing transformations for red-black trees have the following properties:

Fact 1 *The vicinity of an up-in or an up-out request of a node p never contains nodes of the subtree rooted at p.*

Fact 2 *For each node q occuring in the vicinity of an up-in request of a node p, $depth(q) < depth(p)$. Furthermore, if q is black then q must be p's grandparent.*

Our aim is to show that for each given relaxed balanced red-black tree which contains up-in or up-out requests there is always a request that is not in conflict with other requests so that at least one rebalancing transformation can be carried out. We want to show that if a request of a node p is in conflict with other requests then at least one of these other requests is not in conflict with the request of p.

From Fact 1 and Fact 2 it follows

Fact 3 *If an up-in or up-out request of a node p is in conflict with a an up-in or up-out request of a node q and $depth(q) < depth(p)$ then the q's request is not in conflict with the request of p.*

Fact 4 *If a request of a node p is in conflict with a request of a node q and $depth(q) = depth(p)$ then p has an up-out request. Furthermore, if q has an up-in request then q's request is not in conflict with the request of p.*

Fact 5 *Assume that a request of a node p is in conflict with a a request of a node q, $depth(q) > depth(p)$, and there is no other request in p's vicinity with smaller depth than q. If q has an up-in request, then the request of q is not in conflict with any other requests.*

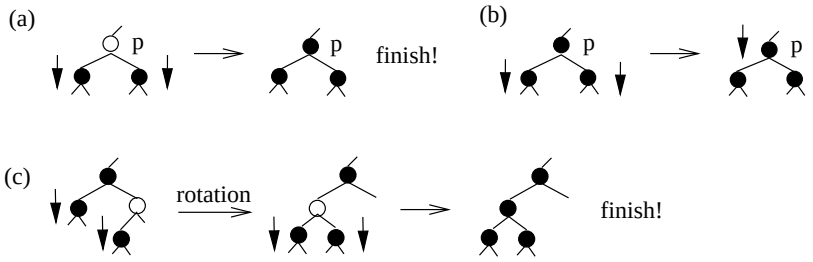

Fig. 5. Additional transformations.

Observe that two up-out requests of sibling nodes are in conflict with each other meaning that we cannot settle one of the requests by using the restructuring *up-out* without any interference of the other one. However, this problem can be solved by using the simple transformation shown in Figure 5a,b. If the parent p of those nodes is red then we colour p black and remove the up-out requests from both children of p. Otherwise, we replace the up-out requests of p's children by an up-out request for p (recursive shift). As the result of this transformation we have abandoned at least one up-out request by using only colour changes.

This additional transformation can be combined analogously with the rotation shown in Figure 4a as the operations shown in Figure 4b–d. If a node p with an up-out request has a red sibling the child of which has an up-out request, the transformation shown in Figure 5c settles both up-out requests.

Lemma 1. *Given a relaxed balanced red-black tree which does not satisfy the balance condition for red-black trees, the tree has at least one node at which one of the rebalancing operations shown in Figure 2—Figure 5 can be carried out while preserving the relaxed balanced red-black tree structure.*

Next we observe that any valid sequence of one-step rebalancing operations which is long enough, will ultimately transform a relaxed balanced red-black tree into a strictly balanced red-black tree again.

Let p be a node of a relaxed balanced red-black tree T. By $\varphi(p)$ we denote the parent node of p in T. The *relaxed black depth* of p, $rbd(p)$ is defined by

$$rbd(p) = \begin{cases} rbd(\varphi(p)) & \text{if } p \text{ is red} \\ rbd(\varphi(p)) + 1 & \text{if } p \text{ is black and has no up-out request} \\ rbd(\varphi(p)) + 2 & \text{if } p \text{ is black and has an up-out request} \end{cases}$$

where we set $rbd(\varphi(\text{root})) := 0$. Let $R(T)$ denote the number of rebalancing requests in T (i.e. the number of up-in, up-out, and removal requests), and let $RD(T)$ denote the sum of all relaxed black depths of nodes in T which have an up-in or an up-out request.

We characterize the balance of a relaxed balanced red-black tree by the tuple $(R(T), RD(T))$ by saying that a relaxed balanced red-black tree T' is *closer to a red-black tree* than another relaxed balanced red-black tree T, denoted $T' < T$, if in accordance to the lexicographic order the tuple $(R(T'), RD(T'))$ is smaller than the tuple $(R(T), RD(T))$. Note that the smallest elements in this relation are red-black trees.

Obviously, the one-step rebalancing operations shown in Figure 2—Figure 5 either decrease the number R of rebalancing requests by settling a request or decrease the sum RD by shifting a request to a node with a lower relaxed black depth. (The relaxed black depth of nodes in the subtree below are not affected by the transformation.) Therefore, we obtain

Lemma 2. *Let T be a relaxed balanced red-black tree with at least one rebalancing request, and let T' be the relaxed balanced red-black tree that has been obtained from T by applying one of the one-step rebalancing transformations shown in Figure 2—Figure 5. Then $T' < T$.*

We get easily an upper bound on the necessary one-step rebalancing operations. One update operation causes at most one rebalancing request. Such a rebalancing request is—in accordance to the relaxed black depth—bubbled up in the tree by using only colour flips until it is finally removed by using a constant number of colour flips and pointer changes. This leads to the main theorem of this paper:

Theorem 3. *Let T be a red-black tree and assume that insertions, deletions and rebalancing transformations (possibly interspersed) are applied to T. Then the number of structural changes (pointer changes) needed to restore the balance condition is $O(i+d)$ where i is the number of insertions and d the number of deletions. The number of colour changes is $O(\log(n+i))$ where n is the size of T.*

We also state the following obvious but important property for relaxed balanced red-black trees and their update scheme.

Theorem 4. *Let T be a red-black tree and assume that insertions, deletions but no rebalancing transformations are applied to T such that a tree T' is obtained. Assume further that T' has no leaves with (unsettled) removal requests and that the sequence of updates has not changed the shape of the original tree. Then T' does not contain any unsettled up-out or up-in requests.*

A possible sequence of those updates is a sequence of insertions of keys $k_1, \ldots, k_r$ which is followed by a sequence of deletions of $k_1, \ldots, k_r$ (or vice versa). Note, that the relaxed balancing scheme proposed in [12] generates even for such a sequence of updates a tree which is full of balance conflicts.

The expanded set of one-step rebalancing operations we have obtained by tuning the basic algorithm for relaxed balancing is essentially the same set of rebalancing operations proposed by Boyar et al. [3], if we assume that the weight of a node in [3] is always $w \leq 2$, and, whenever a rebalancing transformation is carried out, the nodes with a lower depth in the vicinity of the handled request always have $w \leq 1$. It should be clear how to adapt this transformations to the general case $w \geq 0$ proposed in [3] by allowing the accumulation of several up-out requests at single nodes and to move up-out requests from one node to another even if they are not handled by such a transformation.

4 Conclusion

Our aim was to keep the relaxed balancing scheme conceptually as simple as possible. For this reason we have restricted our presentation to the class of red-black trees.

The basic idea of our relaxed balancing scheme (accumulation of insertions and deletions and settling them in an arbitrary order, using the same rebalancing operations as for standard balanced search trees) is applicable to other classes of search trees as well. The idea is simply to let the tree grow below the original leaves as a random binary search tree and to mark an original leaf, if it has to be deleted. Then this set of unsettled updates can be handled just as a sequence of standard updates which are followed immediately by rebalancing transformations. All what is necessary is to assure

that the postponed updates are settled top-down and no two restructurings interfere at the same nodes. Of course, this simple scheme has also opened a lot of room for tuning the individual update algorithms.

References

1. Adel'son-Vels'kii, G.M, Landis, E.M.: "An algorithm for the organisation of information"; Soviet Math. Dokl., 3 (1962), 1259–1262.
2. Bayer, R., McCreight, E.: "Organization and maintenance of large ordered indexes"; Acta Informatica, 1 (1972), 173–189.
3. Boyar, J., Fagerberg, R., Larsen, K.: "Amortization Results for Chromatic Search Trees, with an Application to Priority Queues"; 4th International Workshop on Algorithms and Data Structures, Lecture Notes in Computer Science, 955 (1995), 270–281.
4. Boyar, J., Fagerberg, R., Larsen, K.: "Chromatic priority queues"; Preprint 15, Department of Mathematics and Computer Science, Odense University (1994).
5. Boyar, J., Larsen, K.: "Efficient rebalancing of chromatic search trees"; Journal of Computer and System Sciences, 49 (1994), 667–682.
6. Ellis, C.S.: "Concurrent search in AVL-trees"; IEEE Transactions on Computers, C-29 (1980), 811–817.
7. Ellis, C.S.: "Concurrent search and insertions in 2–3 trees"; Acta Informatica, 14 (1980), 63–86.
8. Guibas, L.J., Sedgewick, R.: "A dichromatic framework for balanced trees"; Proc. 19th IEEE Symposium on Foundations of Computer Science, (1978), 8–21.
9. Keller, A., Wiederhold, G.: "Concurrent use of B-trees with variable-length entries"; SIGMOD Record, 17, 2 (1988), 89–90.
10. Kessels, J.L.W.: "On-the-fly optimization of data structures"; Comm. ACM, 26 (1983), 895–901.
11. Larsen, K.: "AVL trees with relaxed balance"; Proc. 8th International Parallel Processing Symposium, IEEE Computer Society Press, (1994), 888–893.
12. Larsen, K., Fagerberg, R.: "B-trees with relaxed balance"; Proc. 9th International Parallel Processing Symposium, IEEE Computer Society Press, (1995), 196–202.
13. Nurmi, O., Soisalon-Soininen, E.: "Chromatic binary search trees: A structure for concurrent rebalancing"; Acta Informatica 33 (1996), 547–557.
14. Nurmi, O., Soisalon-Soininen, E., Wood, D.: "Concurrency control in database structures with relaxed balance"; Proc. 6th ACM Symposium on Principles of Database Systems, (1987), 170–176.
15. Olivié,H.J.: "A new class of balanced search trees: half-balanced binary search trees"; R.A.I.R.O. Theoretical Informatics, 16, 1 (1982), 51–71.
16. Ottmann, Th., Soisalon-Soininen, E.: "Relaxed Balancing Made Simple"; Techn. Report 71, Institut für Informatik, Universität Freiburg, Germany (1995), also appeared as electronic version, anonymous FTP ftp.informatik.uni-freiburg.de, in /documents/reports/report71/, also http://hyperg.informatik.uni-freiburg.de/Report71.
17. Ottmann, Th., Wood, D.: "Updating binary trees with constant linkage cost"; International Journal of Foundations of Computer Science, 3, (1992), 479–501.

18. Sagiv, Y.: ”Concurrent operations on B*-trees with overtaking”; Journal of Computer and System Sciences, 33, 2 (1986), 275–296.
19. Sarnak, N., Tarjan, R.E.: ”Planar point location using persistent search trees”; Comm. ACM, 29 (1986), 669–679.
20. Shasha, D., Goodman, N.: ”Concurrent search structure algorithms”; ACM Transaction on Database Systems, 13, (1988), 53–90.
21. Soisalon-Soininen, E., Widmayer, P.: ”Relaxed balancing in search trees”; Advances in Algorithms, Languages, and Complexity: Essays in Honor of Ronald V. Book (eds. D.-Z.Du and K.-I. Ko), Kluwer Academic Publishers, Dordrecht (1997). To appear.