Relaxed Balance for Search Trees with Local Rebalancing

Kim S. Larsen[1*], Thomas Ottmann[2], Eljas Soisalon-Soininen[3]

[1] Department of Mathematics and Computer Science, Odense University, Campusvej 55, DK-5230 Odense M, Denmark. E-mail: kslarsen@imada.ou.dk.
[2] Institute of Computer Science, Freiburg University, Am Flughafen 17, D-79110 Freiburg, Germany. 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. Search trees with relaxed balance were introduced with the aim of facilitating fast updating on shared-memory asynchronous parallel architectures. To obtain this, rebalancing has been uncoupled from the updating, so extensive locking in connection with updates is avoided. Rebalancing is taken care of by background processes, which do only a constant amount of work at a time before they release locks. Thus, the rebalancing and the associated locks are very localized in time as well as in space. In particular, there is no exclusive locking of whole paths. This means that the amount of parallelism possible is not limited by the height of the tree.

Search trees with relaxed balance have been obtained by adapting standard sequential search trees to this new paradigm; clearly using similar techniques in each case, but no general result has been obtained. We show how any search tree with local bottom-up rebalancing can be used in a relaxed variant preserving the complexity of the rebalancing from the sequential case. Additionally, we single out the one high level locking mechanism that a parallel implementation must provide in order to guarantee consistency.

1 Introduction

Search trees with relaxed balance are interesting for several reasons. First of all, they give answers to a fundamental question about what is probably the most important data structure: is it possible to uncouple rebalancing in search trees from updating without a great loss in efficiency? The answer is yes, and there are some immediate applications. In a sequential environment during bursts of updates, it is possible to postpone rebalancing partly or completely to speed up the

* Some of the work was done while this author was visiting the Department of Computer Sciences, University of Wisconsin at Madison. The work of this author was supported in part by SNF (Denmark), in part by NSF (U.S.) grant CCR-9510244, and in part by the ESPRIT Long Term Research Programme of the EU under project number 20244 (ALCOM-IT).

processing of user requests. The process can then gradually catch up on delayed rebalancing operations after the burst. In a concurrent environment, the relaxed balancing allows for an efficient solution to the concurrency control problem for binary search trees, and makes a higher degree of parallelism possible.

Search trees with relaxed balance have been studied in many papers, including a survey [27]. Here, we just mention the most important, and, for this paper, relevant results. The crucial idea of uncoupling the rebalancing from the updating was first mentioned in [12]. The first partial result, dealing with insertions only, is from [16]. The first relaxed version of AVL-trees [1] was presented in [21] and proofs of complexity for the rebalancing, matching the complexity from the sequential case, was obtained in [17]. The first relaxed version of B-trees [6] is also from [21] with proofs of complexity matching the sequential ones in [18]. Since [18] really treats (a, b)-trees [15], it also provides proofs for 2-3 trees [14, 2], as well as for (a, b)-trees with other choices of a and b. A relaxed version of red-black trees [12] was introduced in [22] and complexities matching the sequential case were established in [9, 8] for a variant of the original proposal.

Both the worst-case logarithmic time results, which hold for all of the sequential structures mentioned above, as well as the amortized constant time results, which only hold for some, were eventually matched in the relaxed versions.

However, in the cases above, in order to obtain efficient rebalancing, the rebalancing operations in the relaxed version of a given tree structure ended up looking somewhat different from the original ones. Using new strategies for insertion and deletion [24], we show how to use exactly the same rebalancing operations in the relaxed version as in the standard version in such a way that the complexity analysis, with respect to the number rebalancing operations applied, carries over.

Our paper explores the limits of this idea. We discuss the question of what properties a search tree must have in order for a relaxed version to exist and for the complexity to carry over. We give a collection of requirements and prove that if a search tree fulfills these requirements, then such a relaxed version exists. The informal conclusion is that this method works for search trees with bottom-up rebalancing, where rebalancing is in the form of a sequence of local transformations removing or moving purely local conflicts. Bottom-up rebalancing means that rebalancing is performed *after* an update, moving from the location of the update towards the root. The alternative method is top-down rebalancing, where rebalancing is performed while searching for the location for the update such that when the update has been performed, no further rebalancing is required.

This means that in addition to the search trees mentioned above, we now have efficient relaxed versions of a number of other types of search trees. It should be pointed out that the present paper does not only contribute to our understanding of fundamental issues in data structures, but also gives a starting point for an efficient implementation. Before [24] and this present paper, a relaxed version of a given search tree paradigm would have to be created from scratch. Now, by applying our result, there is always a starting point, which can then be optimized for efficiency.

2 Requirements

In this section, we state sufficient conditions which a balanced search tree, along with its rebalancing operations, must fulfill in order for us to be able to equip this structure with a relaxed balancing scheme. In the following, we refer to the given search tree, which is in fact a parameter to this whole set-up, as the T-tree.

For convenience, we assume that the T-tree is binary. However, all results hold for multi-way search trees as well, with minor modifications.

Requirement 1. *The T-tree must be a binary search tree allowing for several occurrences of the same key. It must offer insertion and deletion operations as well as a finite collection of rebalancing operations.*

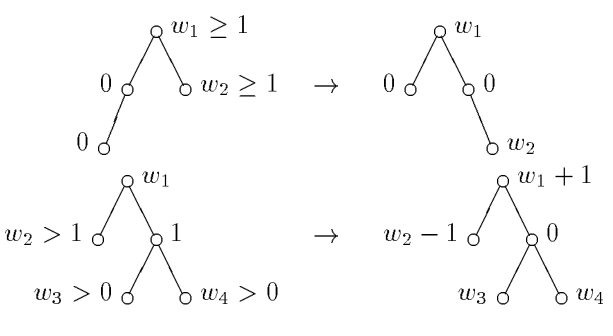
Explanation 1. Nodes can be leaves, or unary or binary internal nodes. The children of a node are referred to as the left and the right child. Each node contains a key, and for any node it must be the case that all keys in its left subtree are smaller than or equal to the key in the node, and all keys in its right subtree are larger than the key in the node.

Nodes may contain additional variables for balance information. These are called the *balance variables*.

Informally, rebalancing after an insertion or deletion must be bottom-up and local, i.e., a rebalancing operation transforms one local problem to another, possibly of the same type, but higher up in the tree, until the problem can be taken care of completely and removed from the tree.

Definition 1. A tuple (t_1, P, t_2, L) is called a *local transformation* if t_1 is a T-tree, P is a boolean predicate over the variables in the nodes of t_1, t_2 is a T-tree with the same number of leaves as t_1, and L is a list of assignments, where the left hand side of each assignment is a variable in a node of t_2 and the right hand side is an expression over the variables in the nodes of t_1.

Below, we give two typical examples of how local transformations are usually given. In these examples, there is only one balance variable that P and L refer to. Call this variable W. Then in the first example, the predicate P is $(u_1.W \geq 1) \wedge (u_2.W = 0) \wedge (u_3.W \geq 1) \wedge (u_4.W = 0)$, where the nodes (u_1, u_2, u_3, u_4) are numbered top-to-bottom and left-to-right. The assignment list L, also in the first example, is $v_1.W := u_1.W$; $v_2.W := 0$; $v_3.W := 0$; $v_4.W := u_3.W$, where the nodes of t_2 (v_1, v_2, v_3, v_4) are also numbered top-to-bottom and left-to-right.



Definition 2. A *rebalancing operation* on a tree T is an operation that replaces one subtree S with another S'. It is the execution of a local transformation (t_1, P, t_2, L) such that the subtree can be written as t_1 with subtrees taking the place of the leaves of t_1. S' is t_2 assigned the values specified in L, and with all the subtrees of t_1 attached to t_2 taking the place of its leaves and in the same order, from left to right, in which they appeared in S.

Also insertion and deletion must be local. This can be formulated in different ways. We choose to use the already defined locality of the rebalancing operations.

Requirement 2. *Insertion and deletion must be rebalancing operations.*

Explanation 2. Insertion must take place in two steps. The new key is inserted into the location of the T-tree, where it should go according to the search tree invariant, and then a rebalancing operation involving the new node is carried out.

For deletion, the node containing the key in question is deleted, and then a rebalancing operation involving the parent of the deleted node is carried out.

Rebalancing must be bottom-up, i.e., after an update causing problems, the problem is taken care of by applying a sequence of rebalancing operations, gradually moving the problem towards the root until it can be removed.

Requirement 3. *Rebalancing must be bottom-up.*

Explanation 3. Rebalancing must be carried out as a finite sequence of rebalancing operations, where for each two consecutive local transformations in this sequence, (t_1^a, P^a, t_2^a, L^a) and (t_1^b, P^b, t_2^b, L^b), t_1^b is rooted at the same node, or at a node above the node where t_1^a is rooted.

It is not sufficient that rebalancing is bottom-up in this simple sense. There is no reason to do the following in a sequential search tree, but if we imagine allowing new updates to be carried out *before* finishing the rebalancing of another, we can create a situation where there are a number of separate problems of imbalance in the tree at the same time. Also under these circumstances must rebalancing be bottom-up, i.e., one problem of imbalance must move towards the root (or possibly remain at the same distance from the root) even in a context where there are other problems around it in the tree. At first, one might think that this stronger requirement must follow from the requirement above, but as we discuss in section 5, this is not the case.

Definition 3. Let h be the maximal height of the t_1 trees in all of the local transformations used in the rebalancing operations including the insertion and deletion rebalancing operations.

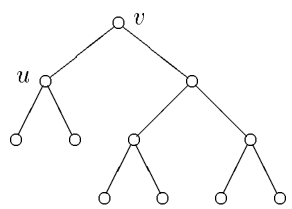
Requirement 4. *Rebalancing must be strongly bottom-up.*

Explanation 4. Assume that we allow updates to be performed with problems of imbalance still present in the tree (the tree is still a search tree, so this is meaningful) as long as the update is performed at a distance at least h below any problem of imbalance. We also allow rebalancing operations to be carried out provided that the problem of imbalance is located at a distance at least h nodes below any other problem of imbalance. Then rebalancing must still be bottom-up, i.e., no matter which problem of imbalance we decide to apply a rebalancing operation to, the problem of imbalance must remain at the same distance from the root or move closer to the root.

Rebalancing is based on local transformations which change the tree locally (parts of the tree which are structurally identical to one of the t_1's). The fact that the t_1's from different local transformations are different (of different shape and possibly of different height), makes it difficult to formulate and prove results concerning the behavior of the rebalancing as a whole. It is also a problem in guaranteeing that two separate problems of imbalance do not interfere. For these reasons, we embed all these different shapes in some common shapes, large enough to contain each of the problems of imbalance. We introduce an *operation area*, which is as much of the tree which it can possibly be necessary for a rebalancing operation to inspect, and a *synchronization area*, which is a larger area used to avoid deadlocks when different problems of imbalance progress towards a common ancestor in the tree.

Definition 4. A *triangle* rooted at a node u in the T-tree is the part of the tree at a distance at most h edges below u. The *operation area* of a triangle rooted at u is rooted at the node v, h edges above u (or at the root of the T-tree if u is fewer than h edges below the root of the T-tree). Let $d_o = depth(u) - depth(v) + h$. The operation area is the part of the tree at a distance at most d_o edges below v, except that nothing in the subtree of u belongs to the operation area. The *synchronization area* of a triangle rooted at u is rooted at the node v, h edges above u (or at the root of the T-tree if u is fewer than h edges below the root of the T-tree). Let $d_s = depth(u) - depth(v) + 2h$. The synchronization area is the part of the tree at a distance at most d_s edges below v, except that nothing in the subtree of u belongs to the synchronization area.

Below, assume that u and its two children constitute a triangle. Then the rest of the edges and nodes shown form u's synchronization area.



The final requirement is that all information pertinent to the rebalancing process must be kept in the tree, i.e., in the balance variables of the nodes, as opposed to, for instance, in the procedure that repeatedly applies the rebalancing operations.

Requirement 5. *Rebalancing must be oblivious.*

Explanation 5. If a rebalancing operation is carried out by applying the local transformation (t_1^a, P^a, t_2^a, L^a), where t_1^a is rooted at u, it must be possible to determine which local transformation to apply next, and where, by inspecting only nodes in the triangle rooted at u or in its operation area. In addition, the next local transformation (t_1^b, P^b, t_2^b, L^b) must have t_1^b rooted at u or at a node in its operation area.

If a local transformation can be applied somewhere, rebalancing cannot yet have terminated.

Some presentations of standard search trees do not fulfill these requirements at all. However, some implementations violate the requirements for trivial reasons, and can easily be *brought* to fulfill these. As an example, when a node is deleted from a red-black tree [12], one node may temporarily be double black. In some presentations, the procedure carrying out the rebalancing operations remembers which node is double black. So, rebalancing is not oblivious. However, it is trivial to change this to registering in the node that it is double black, thereby conforming to the requirements. In fact, any finite control that such a procedure may maintain can just as well be built into the nodes in the tree.

3 A general scheme for relaxed balancing

Now we assume that we are given a T-tree fulfilling the requirements listed in the previous section. Based on this, we define a *relaxed* T-tree. For emphasis, if we need to refer to the T-tree which is not relaxed, we use the term *standard* T-tree. Searching and updating, to be described below, is handled as in [24].

We add the following variables to the nodes of the tree: `insreq` and `delreq`, assuming that the nodes already have the variables `key`, `left`, `right`, `parent`, `external`. It is clear how these variables are intended to be used (`external` is set if the node is a leaf, and unset otherwise) and below, we only discuss the use of the new variables.

We use the tree as a *leaf-oriented* tree, i.e., all keys are kept in the leaves (also called external nodes) and the internal nodes merely contain routers that guide the search through the tree. This generally requires that the tree can contain several identical keys, which is ensured by requirement 1. Routers are of the same type as the keys (and are stored in the `key` variable), but a router is not necessarily present as a key in the tree, since we do not want to update routers when a deletion occurs. A router is greater than or equal to any key in the subtree to its left and less than any key in the subtree to its right.

The tree will at times physically contain keys which should not be there. In those nodes, the deletion request `delreq` is set. A search operation for a key k is performed as follows: First we search for the correct node as usual in a search tree. We will end at a node which is a leaf, because the tree is leaf-oriented. If the leaf has key k and `delreq` is unset, then a positive response is returned. Otherwise the search was negative.

Apart from the nodes that should be deleted, the nodes of the tree are divided into two groups: the ones that are in, what we will refer to as, the *structured part* of the tree, and the ones below the structured part. The ones below are organized in a binary tree without any balance information whatsoever. We call these parts the *unstructured parts* of the tree. The structured part may not actually be balanced (in the sense that all paths have lengths asymptotically logarithmically bounded in the size of the structured part of the tree) since rebalancing may be in progress. It merely reflects that the process of carrying out the necessary rebalancing operations caused by the insertion of the key in a node has begun. If the variable `insreq` is set in a node, then that node belongs to the unstructured part of the tree; otherwise it belongs to the structured part. When we discuss updating and rebalancing below, we also specify the values of `insreq` for the nodes involved.

Updating

Insertion of a key k is performed as follows: First the location for the insertion, a leaf, is found just as for insertion into a standard search tree. If the key in this leaf is k, then `delreq` is unset. Otherwise, the leaf l is replaced by three nodes: an internal node with two leaves as children. One leaf is the leaf l and the other leaf is new and contains the key k and `delreq` is unset. The leaf with the smaller key is made the left child, and the new internal node is given this smaller key as its router. The variable `insreq` in the internal node is set, indicating an insertion request.

Deletion of a key k is performed as follows: First the location for the deletion is found just as for a deletion of a key in a standard search tree. If the key is not found, nothing happens. Otherwise, the key will be in a leaf, since the tree is leaf-oriented. If the parent u of this leaf has `insreq` unset, then the variable `delreq` in the leaf is set. Otherwise, we are in the unstructured part of the tree and the node u is replaced with u's other child (the child which is *not* the leaf with key k).

Notice that the updating outlined above handles redundant updates correctly, i.e., an attempt to insert a key which is already present or delete a key which is not present (or has `delreq` set) will not alter any `insreq` or `delreq` values.

Rebalancing

After some insertions and deletions have been performed, the following problems might be present: there could be leaves, which should be deleted (`delreq` is set), and there could be whole subtrees of nodes that should be moved into the structured part of the tree (all the internal nodes where `insreq` is set) such that balance can be obtained. In this section, we describe how this is done. The insertion and deletion requests create triangles; each update can create one triangle. How this is done is described at the end of this section. First we explain the conditions under which triangles are moved around in the tree.

We say that a relaxed T-tree is *in balance* if no rebalancing operations can be carried out, i.e., for each local transformation (t_1, P, t_2, L), there is no connected set of nodes in the tree matching t_1 such that P holds. If the tree is in balance, then the relaxed T-tree is also a standard T-tree. The goal of the rebalancing process is to work towards obtaining this.

By requirement 5, the local part of the tree to be transformed next (one of the t_1's in a local transformation) can be associated with a triangle as defined in definition 4. When the local transformation is carried out, the next location for a local transformation (if any) can be determined (also by requirement 5) and another triangle can be associated with *this* location. It turns out that in the rules for rebalancing and the complexity proofs in the next section, it is sufficient to deal with triangles moving from one location to the next as outlined above, from when they are created by an update until they disappear.

The restrictions given below for when triangles can be moved deal with the problems of interference and deadlock. We must avoid interference between different rebalancing operations. This can be handled by forbidding that triangles move into each other. However, if this was the only requirement, then a deadlock situation could arise because neither of two triangles located side-by-side would be able to move without entering the other triangle.

Now, a triangle Δ can be moved from one location up to another, if the following two conditions are fulfilled:

- the location Δ is moving to is completely available, i.e., no other triangle occupies any part of the new location.
- if Δ is in the synchronization area of some triangle Δ', this Δ' is located at a depth greater than or equal to the depth of Δ.

If a triangle cannot be moved as a rebalancing operation requires because of the rules above, the operation cannot be performed at that point in time. Note that these rules are meaningful since by requirement 4, rebalancing is strongly bottom-up. Also observe that the second rule implies that it is not possible to move into another triangle's operation area (since triangles can move up a distance of at most h at a time), so there cannot be any interference from other operations when the location for the next triangle following a rebalancing operation has to be decided upon.

Now we comment on the creation and removal of triangles. Dealing with the latter first, when the last rebalancing operation is carried out, the triangle disappears.

Triangles are created when the top node in an unstructured part of the tree is moved into the structured part (a node with an insertion request) or when a node with a deletion request is treated. Since, by requirement 2, insertions and deletions are merely rebalancing operations, the t_1 in the first local transformation to be carried out decides the location of the first triangle. That triangle may be created when the location it wants to occupy is completely available, and does not overlap with the current operation area of any previously generated triangle.

4 Complexity

If a search tree with relaxed balance is to be used in a concurrent environment, a locking mechanism must be applied to ensure the atomicity of rebalancing operations. Even though operations may take place in parallel, for the sake of the complexity argument, we may assume that operations take place one at a time (notice that our goal is merely to count the number of rebalancing operations; not to make any direct statement about the parallel complexity). This is also the approach taken in [9, 17, 8, 18].

So, insertions, deletions, creations of triangles, removal of triangles, and moving of triangles are the basic operations in the following, and the assumption is that they can be interleaved in any way in accordance with the rules given in the previous section.

The following theorem establishes the connection between the operations in a relaxed T-tree and a standard one. In short, the implication is that the rebalancing operations carried out as a response to a number of updates in a relaxed T-tree are *exactly* the same ones that would be carried out in a standard T-tree for some ordering of the updates. The only assumption is that rebalancing in the standard T-tree after any update is finite.

In the following, a triangle rooted at u is said to be *above* another triangle rooted at v, if v is a descendent of u in the tree.

Additionally, we shall talk about nodes *having belonged* to a triangle. For the purpose of the following proof, we say that triangle movements are carried out in the following steps. Given a certain location of a triangle, at some point the next rebalancing operation for the triangle is carried out, and then the triangle is moved to its new location. All the nodes in the triangle immediately before it is moved are said to having belonged to the triangle.

Theorem 5. (Serializability) *Assume that some updates $u_1, \ldots, u_k$ and rebalancing operations $r_1, \ldots, r_p$ are carried out on a relaxed T-tree T, which is in balance. Then there exists an ordering of the updates $u_{i_1}, \ldots, u_{i_k}$ such that the multiset $\{r_{i_1}^1, \ldots, r_{i_1}^{k_{i_1}}, \ldots, r_{i_k}^1, \ldots, r_{i_k}^{k_{i_k}}\}$ contains the multiset $\{r_1, \ldots, r_p\}$, where $r_{i_j}^1, \ldots, r_{i_j}^{k_{i_j}}$ are the rebalancing operations carried out, in that order, after the update u_{i_j} on the corresponding standard T-tree, created by carrying out the operations*

$$u_{i_1}, r_{i_1}^1, \ldots, r_{i_1}^{k_{i_1}}, \ldots, u_{i_{j-1}}, r_{i_{j-1}}^1, \ldots, r_{i_{j-1}}^{k_{i_{j-1}}}$$

on T in the listed order.

Proof. We define an ordering $<$ as follows:

- if the update u_j takes place after all rebalancing operations arising from u_i have been carried out, we define $u_i < u_j$.
- if, at any point in time, a triangle arising from u_i is above a triangle from u_j, we define $u_i < u_j$.
- if a triangle arising from u_j is moved on top of a node that have belonged to a triangle arising from u_i, we define $u_i < u_j$.

– if a triangle arising from u_j is unable to move because it is in the synchronization area of another triangle arising from u_i which is located at a smaller depth, we define $u_i < u_j$.

Clearly, the graph corresponding to $<$ is acyclic, so we can choose a total ordering $u_{i_1}, \ldots, u_{i_k}$ consistent with $<$ (also known as a topological ordering of the graph).

The proof of this theorem is by induction. We prove for any $j \in \{1, \ldots, k\}$ that if only the updates $u_{i_1}, \ldots, u_{i_j}$ are performed, then the set of rebalancing operations carried out in the relaxed T-tree is contained in

$$\{r_{i_1}^1, \ldots, r_{i_1}^{k_{i_1}}, \ldots, r_{i_j}^1, \ldots, r_{i_j}^{k_{i_j}}\}.$$

The base case $j = 1$ is trivial, since with only one update, we have standard T-tree rebalancing. For the induction step, assume that up until the index j, everything holds. We now want to include the update $u_{i_{j+1}}$. Clearly rebalancing from the first j updates will progress as without update $u_{i_{j+1}}$ unless this update changes some nodes that later appear in the operation area of a triangle arising from an update u_{i_h}, $h \leq j$. However, this cannot happen since then by definition, update u_{i_h} would be after update $u_{i_{j+1}}$ in the ordering.

It remains to be shown that the set of rebalancing operations due to update $u_{i_{j+1}}$ carried out in the relaxed T-tree is a subset of $\{r_{i_{j+1}}^1, \ldots, r_{i_{j+1}}^{k_{i_{j+1}}}\}$. If the triangles from $u_{i_{j+1}}$ arrive last to any node that they visit, the result is immediate, since those parts of the tree will appear exactly as if all other rebalancing had been carried out completely. The question is whether it is possible for the triangle Δ arising from $u_{i_{j+1}}$ to arrive at a location containing a node which ought to be visited by a triangle from one of the other updates, u_{i_h}, before the triangle from $u_{i_{j+1}}$ was supposed to arrive. Since the triangle from u_{i_h} will progress up the tree as long as everything looks the way it did before we took update $u_{i_{j+1}}$ into account, it will eventually reach the location where nodes that have belonged to Δ are to be found, or it will be unable to move because it is in the synchronization area of Δ. Both of these possibilities would contradict the ordering. □

With this proof, we have bounded the number of rebalancing operations which can be carried out in the relaxed version by the bound from the standard case. This could, of course, also have been obtained by defining a relaxed version in which no rebalancing operations can be applied. Thus, we also have to prove that all the triangles containing the problems of imbalance can be moved and eventually removed.

The problem is that triangles from different subtrees can meet at a node and one triangle has to move up before the other. If both triangles continue up towards the same node, a potential deadlock situation arises, where none of the triangles can move because the other is in its way. The rules involving the operation and synchronization areas have been designed to avoid this situation.

Theorem 6. (Progress) *If updating in a relaxed T-tree stops, the relaxed T-tree will eventually become a standard T-tree.*

Proof. Assume for the sake of contradiction that there are triangles in the tree, but none of them can move. Consider a triangle Δ at a smallest depth d. Since Δ is at a smallest depth, the two rules for when triangles can move imply that Δ cannot move because the location it wants to move to is partly occupied by another triangle Δ'. This Δ' must be located at a depth no larger than $d + h$; otherwise it cannot possibly occupy the next location for Δ.

Since triangles move at most h edges up at a time (requirement 5), the move leading to this situation could not have been made by Δ', since Δ' could not have moved while in the synchronization area of Δ because of its larger depth. So, assume that the last move was made by Δ. If Δ had smaller depth than Δ' before its last move, it must have moved more than once since Δ' moved. Consider the earliest point where the depth of Δ was no larger than the depth of Δ'. The move by either Δ or Δ' immediately prior to this must have been carried out while in the other triangle's synchronization area with that triangle at a smaller depth. This is impossible according to the rules, and therefore a contradiction.

Since theorem 5 bounds the number of rebalancing operations that can be carried out, the result follows. □

5 Examples

At this point, it is possible to take a concrete search tree, check the requirements from section 2, and conclude that a relaxed version exists, which is as efficient, in terms of the number of rebalancing operations necessary, as the standard version. If all requirements are not fulfilled, one can try to determine if minor changes can be made, rectifying the situation (as in the example given at the end of section 2). This inspection is easy and tedious, and documentation of the requirements being fulfilled is very space consuming, since, among other things, it includes listing all the rebalancing operations.

Here, we merely list some known search trees that fulfill the requirements and summarize the main complexity results which carry over to the relaxed version. This is also an opportunity to comment on the relative strength of worst-case and amortized results which changes in the scenario of relaxed balance.

It is no surprise, of course, that AVL-trees [1], B-trees [6] (as well as 2-3 trees [14, 2] and the more general (a, b)-trees [15]), and red-black trees [12] fulfill the requirements and have relaxed versions, since results on these data structures have already been published [21, 22] with complexity results in [9, 17, 8, 18] including worst-case logarithmic rebalancing per update, and for red-black trees and (a, b)-tree variants with $b \geq 2a$ also amortized constant rebalancing per update, just as it was obtained for the standard case in [25, 15]. For AVL-trees, we also get the amortized constant rebalancing results in insertion-only or deletion-only situations that was proven for the standard case in [19]. However, the relaxed versions that come out from applying the method from this paper are different. Another relaxed version of red-black trees has been defined in [13].

In that paper, problems that we solve in general have been worked out for the specific case of red-black trees.

Also symmetric binary B-trees [5] and half-balanced trees [23] fulfill the requirements, so worst-case logarithmic rebalancing per update is obtained. This is not too surprising since these are similar to red-black trees, in the sense that the shape of the trees arising from these two paradigms can always be colored in such a way that a red-black tree is obtained. However, the rebalancing operations and criteria are different, so they are not the same as red-black trees (rebalancing operations in response to an update would be different).

Of completely new data structures (that is new to the world of relaxed balancing), BB[α]-trees [20] fulfill the requirements and the logarithmic worst-case rebalancing per update as well as the amortized constant rebalancing per update [7] carry over.

Among search trees that were not assumed to have (efficient) relaxed versions using the methods from the earlier papers are splay trees [26]. This data structure also fulfills the requirements and we obtain amortized logarithmic rebalancing for relaxed splay trees. Notice that amortized logarithmic rebalancing is in fact no weaker than worst-case logarithmic rebalancing in this scenario because our statements only deal with the *number* of rebalancing operations carried out and not *when* they are carried out. They can be carried out significantly later than the update that caused them, since they can be interleaved with updates and rebalancing operations caused by other updates.

Note that randomized search trees [4] (also called treaps) fulfill all requirements; except that they are not strongly bottom-up. They are indeed bottom-up. However, if two problems of imbalance exist simultaneously, one above the other, then the one farthest from the root can move *down* when a rebalancing operation is applied. It is easy to construct such an example and we leave this to the reader. This means first of all that the proofs in this paper do not hold. This does not rule out, of course, that the structure could have an efficient relaxed version. However, the naive approach does not work [10] ($\Theta(k)$ rebalancing in the sequential case can become $\Theta(k^2)$ rebalancing in the relaxed version), and there is no obvious way of avoiding the problem. The problems seems to be that problems of imbalance are not *purely* local, since an incorrectly placed priority can have a conflict with not just its neighbors, but with nodes on a whole path.

Some search trees cannot possibly be brought to fulfill the requirements. For instance, search trees that apply global rebuilding, such as general balanced trees [3] and scapegoat trees [11].

6 Comments on implementation

When a search tree with relaxed balance is to be used on a parallel shared-memory architecture, a locking scheme is necessary in order to ensure consistency. Many different locking scheme for tree restructuring have been made (one can be found in [22]) and one of these can be adapted to meet our demands.

However, the important observation is that also this problem can now be solved once and for all. We need a safe mechanism for dealing with triangles: their creation, moving, and removal, as well as simple insertion and deletion into the unstructured parts of the tree. Once this has been accomplished in connection with one relaxed search tree, it can be reused in later implementations of other relaxed search trees. So, also here there is a standard starting point, which can then be optimized for a specific structure. Without any optimization, the result will probably not be satisfactory because of the size of the operation and synchronization areas.

At the algorithmic level, optimization of this standard proposal for rebalancing towards higher efficiency in a concrete implementation could involve combining updates before they are moved into the structured part of the tree, or allowing for some controlled interference between problems in triangles, as has been done in earlier proposals for relaxed search trees, the idea being that combined problems can sometimes be handled in fewer rebalancing operations total, and sometimes two or more problems of imbalance cancel out.

Acknowledgements

The first author would like to thank Rolf Fagerberg for numerous interesting discussions on this and related topics, and Joan Boyar for discussions and for comments on an earlier draft of the paper.

References

1. G. M. Adel'son-Vel'skiĭ and E. M. Landis. An Algorithm for the Organisation of Information. *Doklady Akadamii Nauk SSSR*, 146:263–266, 1962. In Russian. English translation in *Soviet Math. Doklady*, 3:1259-1263, 1962.
2. Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.
3. Arne Andersson. Improving Partial Rebuilding by Using Simple Balance Criteria. In Frank Dehne, Jörg-Rüdiger Sack, and Nicola Santoro, editors, 1st International Workshop on Algorithms and Data Structures*, Lecture Notes in Computer Science*, volume 382, pages 393–402. Springer-Verlag, 1989.
4. Cecilia R. Aragon and Raimund G. Seidel. Randomized Search Trees. In *Proceedings of the 30th Annual IEEE Symposium on the Foundations of Computer Science*, pages 540–545, 1989.
5. R. Bayer. Symmetric Binary B-Trees: Data Structure and Maintenance Algorithms. *Acta Informatica*, 1:290–306, 1972.
6. R. Bayer and E. McCreight. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica*, 1:173–189, 1972.
7. Norbert Blum and Kurt Mehlhorn. On the Average Number of Rebalancing Operations in Weight-Balanced Trees. *Theoretical Computer Science*, 11:303–320, 1980.
8. Joan Boyar, Rolf Fagerberg, and Kim S. Larsen. Amortization Results for Chromatic Search Trees, with an Application to Priority Queues. In Fourth International Workshop on Algorithms and Data Structures*, Lecture Notes in Computer Science*, volume 955, pages 270–281, 1994. To appear in *Journal of Computer and System Sciences*.

9. Joan F. Boyar and Kim S. Larsen. Efficient Rebalancing of Chromatic Search Trees. *Journal of Computer and System Sciences*, 49(3):667–682, 1994.
10. Rolf Fagerberg. Personal communication. 1996.
11. Igal Galperin and Ronald L. Rivest. Scapegoat Trees. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 165–174, 1993.
12. Leo J. Guibas and Robert Sedgewick. A Dichromatic Framework for Balanced Trees. In *Proceedings of the 19th Annual IEEE Symposium on the Foundations of Computer Science*, pages 8–21, 1978.
13. S. Hanke, Th. Ottmann, and E. Soisalon-Soininen. Relaxed Balanced Red-Black Trees. In *Proc. 3rd Italian Conference on Algorithms and Complexity*, 1997. To appear.
14. J. E. Hopcroft. Title unknown. Unpublished work on 2-3 trees, 1970.
15. Scott Huddleston and Kurt Mehlhorn. A New Data Structure for Representing Sorted Lists. *Acta Informatica*, 17:157–184, 1982.
16. J. L. W. Kessels. On-the-Fly Optimization of Data Structures. *Communications of the ACM*, 26:895–901, 1983.
17. Kim S. Larsen. AVL Trees with Relaxed Balance. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 888–893. IEEE Computer Society Press, 1994.
18. Kim S. Larsen and Rolf Fagerberg. Efficient Rebalancing of B-Trees with Relaxed Balance. *International Journal of Foundations of Computer Science*, 7(2):169–186, 1996.
19. Kurt Mehlhorn and Athanasios Tsakalidis. An Amortized Analysis of Insertions into AVL-Trees. *SIAM Journal on Computing*, 15(1):22–33, 1986.
20. J. Nievergelt and M. Reingold. Binary Search Trees of Bounded Balance. *SIAM Journal on Computing*, 2(1):33–43, 1973.
21. O. Nurmi, E. Soisalon-Soininen, and D. Wood. Concurrency Control in Database Structures with Relaxed Balance. In *Proceedings of the 6th ACM Symposium on Principles of Database Systems*, pages 170–176, 1987.
22. Otto Nurmi and Eljas Soisalon-Soininen. Uncoupling Updating and Rebalancing in Chromatic Binary Search Trees. In *Proceedings of the Tenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 192–198, 1991.
23. H. J. Olivié. A New Class of Balanced Trees: Half-Balanced Binary Search Trees. *R.A.I.R.O. Informatique Théoretique*, 16(1):51–71, 1982.
24. Th. Ottmann and E. Soisalon-Soininen. Relaxed Balancing Made Simple. Technical Report 71, Institut für Informatik, Universität Freiburg, 1995.
25. Neil Sarnak and Robert E. Tarjan. Planar Point Location Using Persistent Search Trees. *Communications of the ACM*, 29:669–679, 1986.
26. Daniel Dominic Sleator and Robert Endre Tarjan. Self-Adjusting Binary Trees. In *Proceedings of the 15th Annual ACM Symposium on the Theory of Computing*, pages 235–245, 1983.
27. Eljas Soisalon-Soininen and Peter Widmayer. Relaxed Balancing in Search Trees. In D.-Z. Du and K.-I. Ko, editors, *Advances in Algorithms, Languages, and Complexity*, pages 267–283. Kluwer Academic Publishers, 1997.

This article was processed using the LaTeX macro package with LLNCS style