International Journal of Foundations of Computer Science
© World Scientific Publishing Company

VARIANTS OF (A,B)-TREES WITH RELAXED BALANCE

LARS JACOBSEN KIM S. LARSEN
Department of Mathematics and Computer Science
University of Southern Denmark, Odense
Campusvej 55, DK-5230 Odense M, Denmark
{*eljay,kslarsen*}@*imada.sdu.dk*

Received (received date)
Revised (revised date)
Communicated by Editor's name

ABSTRACT

New variants of (a, b)-trees with relaxed balance are proposed. These variants have better space utilization than the earlier proposals, while the asymptotic complexity of rebalancing is unchanged. The proof of complexity, which is derived, is much simpler than the ones previously published. Through experiments, some of the most interesting applications of this data structure are modeled, and it is demonstrated that the new variants are competitive.

Keywords: search trees, (a, b)-trees, relaxed balance, amortized analysis.

1. Introduction

For integers a and b such that $a \geq 2$ and $b \geq 2a - 1$, an (a, b)-tree [12, 24] is a multi-way search tree where each node has between a and b children, and where all leaves are at the same depth. For large values of a, this guarantees a small height.

B-trees, the most popular type of (a, b)-tree where $b = 2a - 1$, were proposed in [3]. In [26], a relaxed version of B-trees was introduced. The requirements are relaxed in the following way: nodes are allowed to have fewer than a children, and leaves are no longer required to be at the same depth. The obvious disadvantage is that logarithmic heights can no longer be guaranteed. However, new applications are made possible from the fact that updating can now be performed without the subsequent rebalancing, and this can be allowed without violating the relaxed criteria.

In a parallel (shared-memory) setting, it therefore becomes possible to make updates without locking more than one leaf. This is as opposed to a naive implementation where significant parts of the whole search path must be locked, thereby decreasing parallelism in the system. In a sequential setting, rebalancing can be "turned off" during bursts of updates to increase the processing rate.

If these options are used, it is of course of great interest to be able to rebalance efficiently, and with as few operations as possible get back to a standard tree, where the strict constraints are fulfilled such that logarithmic height is again guaranteed. It turns out that this can be done independently in small steps, which is important for the locking in the parallel setting, and which implies in the sequential setting that rebalancing can be turned off at any time. However, in the sequential setting, the most important property is that each update only gives rise to an amortized constant number of rebalancing operations, meaning that it is relatively easy to catch up with the rebalancing while still processing requests at a slower pace after the burst. Proofs that (a, b)-trees with relaxed balance can be implemented efficiently were given in [19, 20].

An account of the history of relaxed balance from the first ideas in [9, 15] can be found in [18]. Here is a very brief account of structures other than multi-way trees.

AVL-trees [1] were made relaxed in [26, 29] with complexity results in [17]. Red-black trees [2, 9, 31] were made relaxed in [27, 28] with complexity results in [4, 5, 6, 7, 18]. In [21], a general relaxed balance result for balanced trees is presented, and in [11, 22], variations are developed. Locking techniques for relaxed structures in a parallel setting are discussed in [5, 28]. Furthermore, an extension of the locking protocol proposed in [28] which is suitable for applications using a problem queue is discussed in [10].

In this paper, we focus on two problems in the proposal for relaxed (a, b)-trees from [20, 26]. The first problem is that when splits, caused by insertions, are handled, new nodes with degree only two may be created. These nodes cannot have their degree increased while they exist, meaning that the remaining space in those nodes is wasted; unless special small nodes are made for this purpose, forcing the program to deal with nodes of different types. The second problem is that rebalancing is somewhat restrictive in that only the topmost of a group of these degree two nodes can be rebalanced.

In response to this, we suggest a new variant based on chaining siblings which avoids that problem of low degree nodes. This can be seen as the obvious relaxed version of B^{link}-trees [16, 23, 30]. However, if updating is very non-uniform, this proposal may result in longer search paths on average in the relaxed structure. Therefore, we also suggest another variant for very skew applications. This second variant is a further development of [20] where degree two nodes may be created, but where the number of children can increase gradually instead of being fixed at two until the node disappears. In addition, the rebalancing is more flexible in that the constraints on when the rebalancing operations can be applied impose fewer restrictions on the ordering of the rebalancing operations.

We prove that the asymptotic complexity of both of the new proposals match the results achieved in [20], and the proofs are much simpler than the one from [20]. Additionally, we verify experimentally that the improvements we set out to make are in fact obtained.

2. Preliminaries

The trees we consider are leaf-oriented, meaning that all keys are kept in the leaves. Internal nodes contain routers, which are of the same type as the keys and often copies of some of these. However, the only purpose of the routers is to guide the searches to the correct leaves. Leaf-oriented trees are often the choice in large database-oriented applications, and the preferred choice in designing relaxed structures, since there does not seem to be a good way of carrying out deletions in a local manner if the tree is not leaf-oriented. In the rest of the paper, we only consider leaf-oriented trees. We assume that the leaves contain the keys and references to the actual data associated with the keys. For convenience, these references are also referred to as children.

If $a \geq 2$ and $b \geq 2a-1$, then an (a,b)-tree can be defined as a multi-way search tree where:

- The root has at most b children and at least 2 children.
- All other nodes have at most b children and at least a children.
- All leaves have the same depth.

The number of children of a node is often referred to as the *degree* of the node.

An internal node u with m children (pointers to subtrees) also stores $m-1$ distinct routers in increasing order $k_1, k_2, \ldots, k_{m-1}$. Let $k_0 = -\infty$ and $k_m = \infty$, then the following search tree invariant is maintained by all operations on the tree: all keys in the range $[k_i, k_{i+1})$, $0 \leq i \leq m-1$, in u's subtree are stored in the ith subtree of u.

When insertions are made into an (a,b)-tree, the correct leaf for the insertion is found and the element is inserted. Then, if the maximal degree constraint is violated, the leaf is *split*, creating an extra child of the parent. If the parent's degree now exceeds b, it is in turn also split, and so on. If the root is split, a new root with two children is created.

Deletions are handled similarly, also bottom-up. Once the leaf is found and the element deleted, the leaf might contain fewer than a elements. In this case, one considers *sharing* with either of its siblings, provided that the leaf and one of its siblings together have at least $2a$ elements, in which case the elements are distributed evenly among the two nodes. If there are fewer than $2a$ elements total among the node and either one of its siblings, all the elements from the two nodes are moved to one leaf, and the reference to the other node is removed from the parent. This is referred to as *fusion*. A fusion may change the parent's degree to $a-1$, in which case again either a sharing or a fusion is performed, and so on. If the minimal degree of the root is violated, implying that the root has become unary, the root is removed making its only child the new root.

Since we are going to consider relaxed (a,b)-trees, we may, for emphasis, refer to the data structure just defined as *standard* (a,b)-trees.

In designing a relaxed version of an (a,b)-tree, there are some obvious requirements based on our discussion in the introduction concerning the intended parallel or sequential use of the data structure.

- The balance constraints should be relaxed such that updates can be performed without immediate rebalancing.
- It should be possible to gradually change the relaxed tree back into a standard tree of (guaranteed) logarithmic height.
- Rebalancing must be efficient in terms of the number of operations which are required in response to an update.

There are several ways of designing operations which fulfill these criteria. In this paper, we propose two such designs and compare these to a third from [20]. All three proposals have the optimal asymptotic complexity [32] of a constant number of rebalancing operations per update. For all three proposals, as well as the original non-relaxed version, this result holds only provided $b \geq 2a$.

3. Relaxation via Chaining

In the following, we define a relaxed collection of balance constraints. Nodes are equipped with an extra pointer which may or may not be used, and we refer to these as chain-pointers. These pointers go to the right in the sense that if u points to v, then all keys in the subtree of v are larger than all keys in the subtree of u. Such a node v which is referenced by a chain-pointer in u does not have a parent. Thus, searching must be via u. This idea is very similar to B^{link}-trees [16, 23, 30], where every node has such a pointer. In fact, this collection of operations can be seen as the obvoius way to divide the rebalancing in B^{link}-trees up into constant-sized steps.

We refer to any node heading a chained list as *referenced*, since it is referenced by its parent, whereas nodes only referenced via a chain-pointer are called *unreferenced.* A separating router is kept with the chain-pointer to be used when the unreferenced node does again get a parent. We let $p(u)$ denote the parent of u, if u is referenced; otherwise it denotes its predecessor in the chained list of siblings. Furthermore, $c(u)$ denotes the number of children of u.

We can now define the *relaxed level* of a node u.

$$rl(u) = \begin{cases} 0 & u \text{ is the root,} \\ rl(p(u)) + 1 & u \text{ is referenced,} \\ rl(p(u)) & u \text{ is unreferenced} \end{cases}$$

Finally, if $a \geq 2$ and $b \geq 2a - 1$, then a relaxed (a, b)-tree can be defined as a multi-way search tree as follows.[a]

- For any pair of leaves ℓ_1 and ℓ_2, we have that $rl(\ell_1) = rl(\ell_2)$.
- If ℓ is a leaf, then $0 \leq c(\ell) \leq b$.
- If u is an internal node, then $1 \leq c(u) \leq b$.

[a] Everywhere in this paper, we consider a tree containing a total of at most two elements, stored at the root, to be a standard (a, b)-tree.

- If u is not the root, then u is pointed to by either its predecessor sibling or its parent, but not by both.

Thus, the problems, also called *conflicts*, we may have in such a relaxed tree which distinguishes it from a standard (a, b)-tree are the following. Nodes may be *underfull*, meaning that they have fewer than a (fewer than 2, if it is the root) children, and nodes may be *unreferenced* by a parent if they are instead referenced by a chain-pointer. In this case, we associate the conflict with the node from where the chain-pointer originates.

We will see how these problems arise through updating and later how we can remove the problems. The philosophy in the rebalancing is the usual that problems are fixed immediately if possible and otherwise moved closer to the root (where they can always be fixed). The great difference from standard rebalancing is that rebalancing can be carried out independently in small steps and can be interrupted at any point (between these steps).

We define all operations by figures, explaining notation the first time it is used. We show only the pointers (or groups of pointers); it is clear that if pointers are moved around, keys should be moved accordingly. Greek letters are used for sets of pointers whereas Roman letters and explicitly drawn pointers ($\boxed{\bullet}$) represent single pointers.

In general for all the operations, the relative order of the keys and therefore also the references must be preserved. Thus, when references are moved from one node to its right sibling, it is the references with the largest keys which are moved.

Another matter which cannot be inferred from the illustrations is which nodes that are new. Here, we decide that when a new chain-pointer is created, the node pointed to by the chain-pointer is new. So, all references to the old node, identified as the origin of the chain-pointer, are still valid.

Similarly, when a chain-pointer is removed by removing a node, it is the node which is pointed to by the chain-pointer which is removed.

Figure 1 shows how insertions are handled.

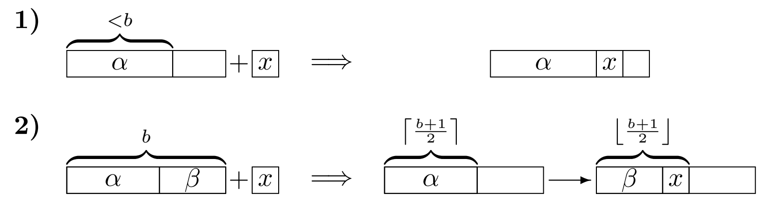

Figure 1: *Insert* operations; inserting x.

Thus, if x should be inserted into a node u with fewer than b children, the insertion can be performed directly. If, on the other hand, u is full, then a new node is created. The new node is referenced by a chain-pointer from u. Out of the total of $b+1$ references (the b references along with the new x), the $\left\lfloor \frac{b+1}{2} \right\rfloor$ references with the largest keys are moved to the new node.

Figure 2 shows deletion, and Figure 3 shows how rebalancing near the root is handled.

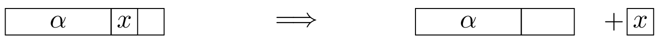

Figure 2: *Delete*; deleting x.

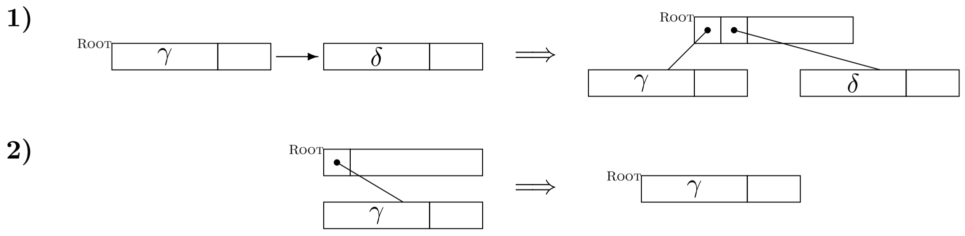

Figure 3: *Root* operations.

Figure 4 shows how splits, necessitated by insertions, are performed. If the top node has fewer than b children, *Split 1* is applied; otherwise, *Split 2* is used. Figure 5 shows how sharing, necessitated by deletions, is handled. Notice that $2a$ is used as the threshold since this is the minimum number of pointers required to fill two nodes to the minimum degree. In Figure 6, fusion is illustrated.

Note that we are not concerned with where pointers are actually inserted in nodes nor in which half of a split node depicted pointers are placed; we merely show one among several analogous cases.

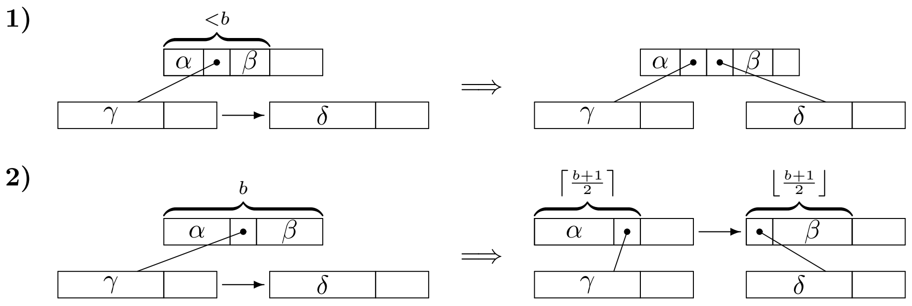

Figure 4: *Split* operations.

3.1. Correctness and Complexity

If u is a node in an (a, b)-tree, we define

$$a_u = \begin{cases} 2, & \text{if } u \text{ is the root} \\ a, & \text{otherwise} \end{cases}$$

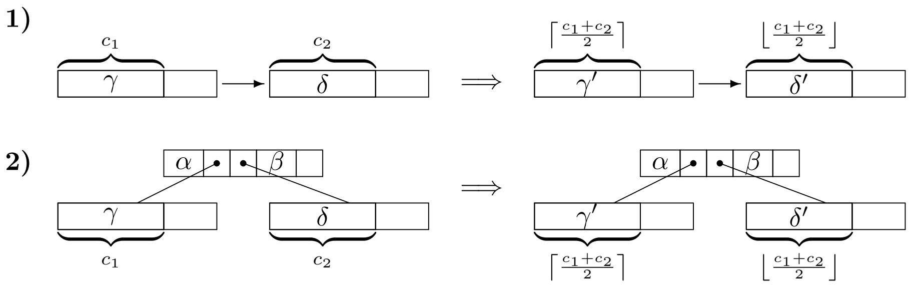

Figure 5: *Share* operations. Requirement: either $c_1 < a$ or $c_2 < a$, and $c_1 + c_2 \geq 2a$.

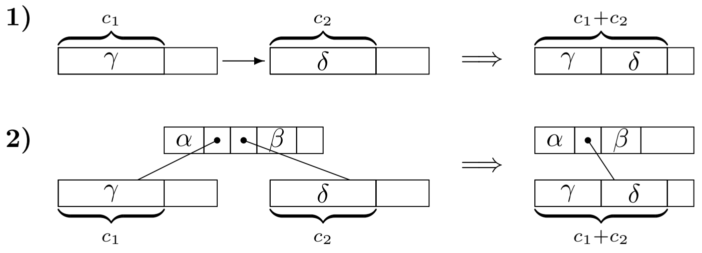

Figure 6: *Fuse* operations. Requirement: $c_1 < a$ or $c_2 < a$, and $c_1 + c_2 < 2a$.

where a is the minimal degree of a non-root node, as already defined.

The following two propositions state the soundness and completeness of the scheme, respectively. Here, soundness means that after an operation has been applied to a relaxed (a, b)-tree, the relaxed criteria are still fulfilled. Completeness means that as long as the tree is not a standard (a, b)-tree, some operation can be applied.

Proposition 1 *Any of the operations turn a relaxed (a, b)-tree into a relaxed (a, b)-tree.*

Proof. Follows easily by inspection of the operations. □

Proposition 2 *If a relaxed (a, b)-tree contains a conflict, at least one of the rebalancing operations can be applied.*

Proof. Assume that the tree contains conflicts. Let u be the topmost and leftmost node containing a conflict. If u is the root, either *Root 1* or *Root 2* can be applied. Assume that u is not the root. If $c(u) < a$ and u has an unreferenced sibling, it can be fused or shared with it. If it has no such sibling, it has a real sibling, since u was a topmost conflict and u's parent therefore has at least two children. Then u can be fused or shared with its real sibling.

Now assume that $c(u) \geq a$. Then u has an unreferenced sibling, since there is a conflict. The node u itself must be referenced since the conflict was chosen to be leftmost. Then u's unreferenced sibling can be inserted into u's parent while perhaps splitting that parent. □

Theorem 1 *If $b \geq 2a$, then rebalancing is amortized constant per update.*

Proof. Define the following (not necessarily disjoint) sets:

$$\textsc{min} = \{u \mid c(u) = a_u\} \qquad \textsc{uf} = \{u \mid c(u) < a_u\}$$
$$\textsc{max} = \{u \mid c(u) = b\} \qquad \textsc{nr} = \{u \mid u \text{ is unreferenced }\}$$

Let the potential of a relaxed (a, b)-tree T be

$$\Phi(T) = 3(|\textsc{uf}| + |\textsc{nr}|) + 2|\textsc{max}| + |\textsc{min}|$$

Since any update operation involves only a constant number of nodes, it increases the potential by at most a constant. In the following, we show that any rebalancing operation decreases the potential by at least one. Since the potential is always at least zero, the result follows.

Consider the split operations. In the case of *Split 1*, the potential decreases by three for the newly referenced node, while it might increase by two for the additional pointer in the parent. For *Split 2*, we have that the size of NR remains unchanged, and in the worst case ($b = 2a$), one minimally filled node replaces a full node.

In the case of *Fusion 1* and *Fusion 2*, two nodes, u and v, of which at least one (say u) is underfull, are merged into one. For this proof, we consider the underfull node u to be the one which is removed. The potential of the node which is the merger of u and v contributes with a potential less than or equal to that of v. Additionally, the potential released by removing u is at least one larger than the possible increase in potential from deleting a pointer in the parent. Thus, in total, the potential is decreased by at least one.

For *Share 1* and *Share 2*, at most two minimally filled nodes replace one underfull node. Thus, the potential decreases by at least one. For *Root 1*, an unreferenced node is made referenced while creating a minimally filled root, whereas in *Root 2* an underfull node is removed. In both cases the potential is easily seen to decrease. □

4. Relaxation via Height Adjustments

In this section, we present a different relaxed (a, b)-tree, which is an extension of the one presented in [20]. Also this scheme has improved space utilization compared with [20]. However, by using a tree-shaped buffer area like in [20] instead of a chained list, we are not penalized on searches.

As in [20], nodes are now equipped with tag values. In [20], tags zero and minus one were allowed. Here, we allow nodes to have positive integer tags as well.

In the following, let the *tag* of a node u be denoted $t(u)$. As opposed to the previous section, since there are no chains in the relaxation scheme to be defined now, $p(u)$ always denotes the parent of u, which must exist for any non-root node u. The relaxed level $rl(u)$ of a node u in this new type of relaxed (a, b)-tree is defined as follows:

$$rl(u) = \begin{cases} t(u) & u \text{ is the root,} \\ rl(p(u)) + 1 + t(u) & \text{otherwise} \end{cases}$$

For integers a, b, a relaxed (a, b)-tree must fulfill the following set of constraints:

- For any pair of leaves ℓ_1 and ℓ_2, we have that $rl(\ell_1) = rl(\ell_2)$.
- If ℓ is a leaf, then $0 \leq c(\ell) \leq b$ and $t(\ell) \geq 0$.
- If u is an internal node, then $1 \leq c(u) \leq b$ and $t(u) \geq -1$.

The idea is now the same as in the previous section that updates create problems (conflicts) which locally may violate the standard (a, b)-tree constraints, and rebalancing operations are designed to gradually remove these conflicts or move them closer to the root.

In Figures 7 through 12, we illustrate the operations for this data structure. To show how the tags are adjusted, we write these as superscripts to the nodes. On the left hand side of the operations (the situation before the operation is carried out), if the value zero is displayed, the operation can only be applied to nodes with tag value zero. Similarly, if it reads $t_1 \neq 0$, the tag value must be different from zero. On the right hand side (the situation after the operation is carried out), the new tag values are displayed as functions of the old tag values.

Updates are handled as shown in Figures 7 and 8. Notice that since we now do not have chain-pointers, a leaf can only be split easily if there is room for an additional pointer in the parent. Otherwise, an intermediate level must be introduced, and tag values must be adjusted such that the relaxed level does not change. From a correctness point of view, *Insert 2* could be used also when the tag value is larger than zero. The reason for choosing *Insert 3* in that case is motivated by complexity concerns, since positive tag values indicate that levels are missing (if some node has a positive tag value, any path from the root through that node must have fewer nodes on it in order for all leaves to be at the same relaxed level).

The operations for handling conflicts near the root are depicted in Figure 9. Negative tags, which are created by insertions, are treated as shown in Figure 10. Similar to the discussion above, a negative tag of -1 indicates that there are too many nodes. More precisely, the references in a node with tag -1 really belong in the layer above. Thus, each of the operations of Figure 10 basically moves references from a node with tag -1 up into its parent and removes the tag -1 node. However, since the total number of references in the parent may exceed b after such a move, a splitting may be performed as an indivisible part of the operation.

Underfull nodes, created by deletions, can be merged with other nodes as shown in Figure 11. Finally, positive tags, created by some of these operations, can be handled using operations in Figure 12.

Before we treat the properties of this structure formally, we discuss some of the motivating design issues.

Positive tags are introduced when underfull nodes with no siblings are to be rebalanced. In this case, the parent is a unary node and (parts of) the search path is just a chained list. Instead of waiting until the parent has been fused or has shared with one of its siblings, we compress the path to speed up searches to the underlying subtree.

Space utilization is improved over [20] by examining the parent when nodes are split, since sparse tag -1 nodes are not created unless it is necessary, i.e., if the

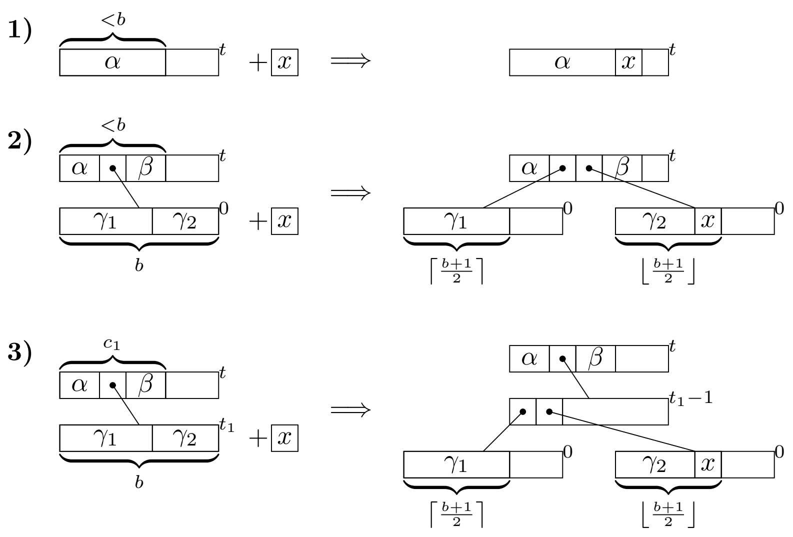

Figure 7: *Insert 1–3.* For *Insert 3*, either $c_1 = b$ or $t_1 > 0$.

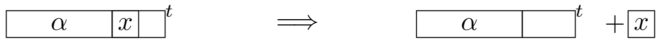

Figure 8: *Delete*; deleting x.

reference to the new node will not fit in its parent. Furthermore, if the pointers from a tag -1 node fit in the parent when we examine the nodes with the purpose of rebalancing, we insert them into the parent regardless of the parents tag. In this way, we can deal with more than just the top-most tag -1 node in a connected component of such nodes.

4.1. Correctness and Complexity

The following two propositions address the soundness and completeness of the scheme, respectively.

Proposition 3 *Any of the operations turn a relaxed (a, b)-tree into a relaxed (a, b)-tree.*

Proof. We prove the proposition by induction on the number of applied operations since the tree was last a standard (a, b)-tree. The base case follows trivially, since any standard (a, b)-tree fulfills the relaxed constraints. Let T be an (a, b)-tree fulfilling the relaxed constraints. We show that after the application of one of the operations, T still fulfills the constraints.

It is easily verified that no operation can violate the degree constraints. In the following, we are therefore only concerned with the operations that change tags and lengths of search paths.

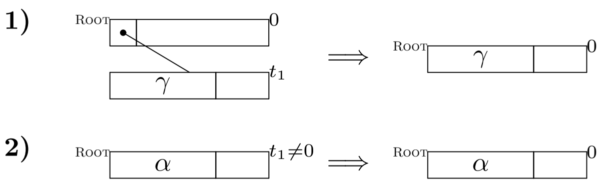

Figure 9: *Root* operations.

First observe that if the length of the search path to some subtree is increased (decreased) by one while the sum of the tags on the exact same path is decreased (increased) by one, the relaxed levels are unchanged. In particular, internal tag -1 nodes can be created or removed without violating any constraints. Furthermore, creating a new root, deleting a unary root, or clearing the root's tag, affects all leaves equally.

Finally, it is also easily verified that any newly created leaf has tag value at least zero, and that any newly created internal node has tag value at least -1. $\square$

Proposition 4 *If a relaxed (a, b)-tree contains a conflict, at least one of the rebalancing operations can be applied.*

Proof. Let T be a relaxed (a, b)-tree. Assume that T contains conflicts and let $\mathcal{C}$ be the set of topmost conflicts. Let r denote the root of T. Assume that $r \in \mathcal{C}$. Then either r is underfull, or r has a non-zero tag. In the first case, *Root 1* can be applied, and in the latter, *Root 2* can be applied. Hence, in the following, we assume $r \notin \mathcal{C}$.

Assume that $\mathcal{C}$ contains a node u with $t(u) = -1$. If the pointers from u fit in u's parent $p(u)$, *Minus One 2* can be applied. Otherwise, if $p(u)$ is the root, *Minus One 1* can be applied, or if $p(u)$ is not the root, then depending on whether the extra pointer obtained from splitting $p(u)$ will fit in u's grandparent or not, either *Minus One 3* or *Minus One 4* can be applied.

Now assume that $\mathcal{C}$ contains no negatively tagged nodes. Assume that $\mathcal{C}$ contains a node u with $t(u) > 0$. Observe that u has at least one neighbor (since u was a topmost conflict, its parent cannot be underfull) and all of u's neighbors have tags greater than or equal to zero. Let v be a neighbor of u. Assume that $t(v) \leq t(u)$; otherwise consider v instead. If $t(v) > 0$, *Positive 1* can be applied. Now assume that $t(v) = 0$. If $c(v) < b - 1$, then depending on whether $p(u)$ has more than two children or not, either *Positive 2* or *Positive 3* can be applied. Finally, if $c(v) \geq b - 1$, *Positive 4* can be applied.

Finally, assume that $\mathcal{C}$ contains no non-zero tagged nodes. Thus, all nodes in $\mathcal{C}$ are underfull. Again u has at least one neighbor, and every neighbor of u has tag zero. Hence, either *Fuse* or *Share* can be applied to u and its neighbor. $\square$

Note that the *Compress* operations are not necessary in order to show completeness, but the conflicts introduced by them can be handled. In the following, we show that allowing nodes to have positive tags, or allowing negatively tagged nodes to have larger degree than two, does not change the asymptotic complexity

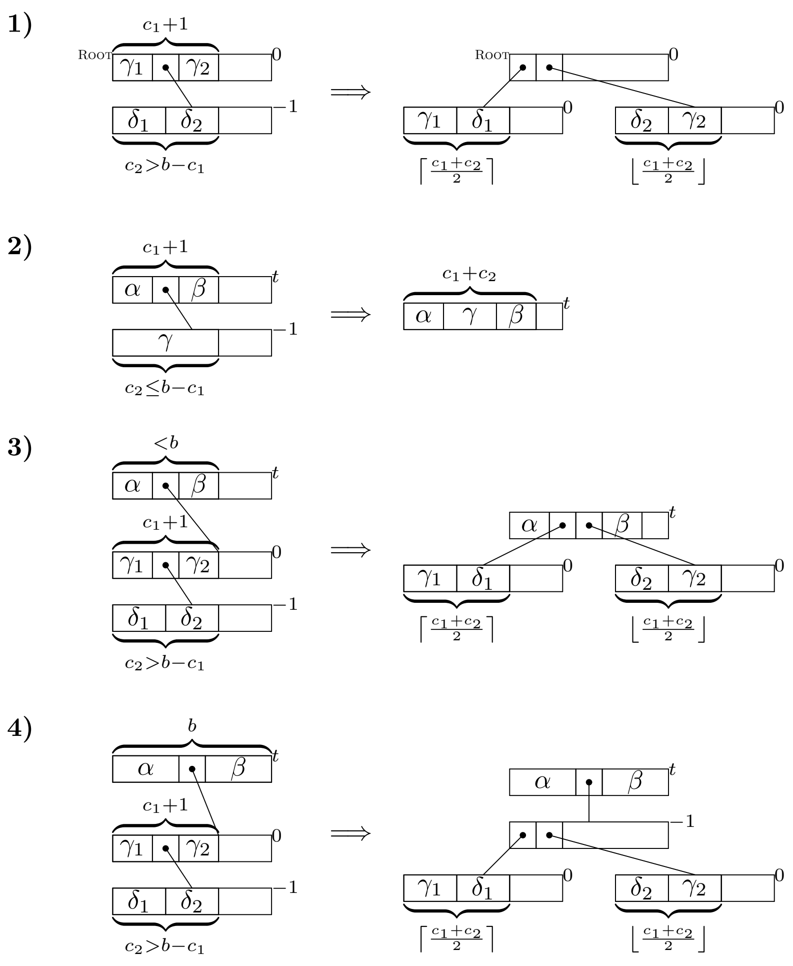

Figure 10: *Minus One* operations.

of rebalancing.

Theorem 2 *If $b \geq 2a$, then rebalancing is amortized constant per update.*

Proof. Define the potential of a node u, denoted $\Phi(u)$, to be:

$$\Phi(u) = \begin{cases} 3t(u) + \begin{cases} 4 & c(u) = 1 \\ 3 & 1 < c(u) < a \\ 1 & c(u) = a \\ 2 & c(u) = b \\ 0 & \text{otherwise} \end{cases} & t(u) \geq 0 \\ 1 + 2(c(u) - 1) & t(u) < 0 \end{cases}$$

Note that the cost of decreasing the number of pointers in any node depends on the value of a. If $a = 2$, the potential is increased by at most 3 if the number of pointers is reduced by one. However, if the number is not reduced to below two,

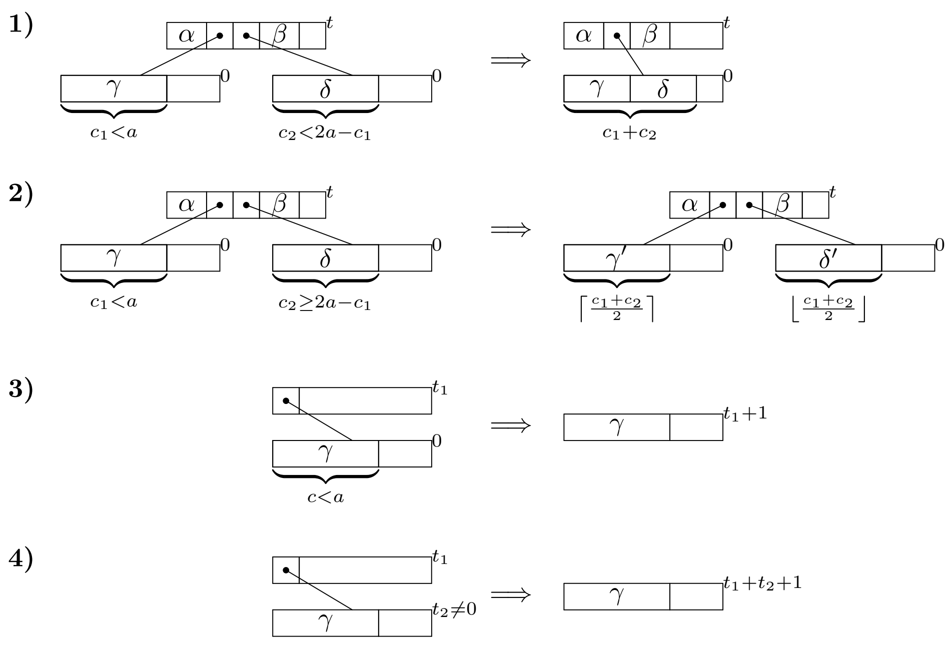

Figure 11: Handling underfull nodes. 1) *Fuse*, 2) *Share*, 3) *Compress 1*, 4) *Compress 2*.

only an increase of 1 can be incurred. If $a > 2$, the increase in potential is always at most 2 when the number of pointers is reduced by one. Analogously, the cost of increasing the number of pointers in any node regardless of its tag and a is at most 2.

First observe that any update operation increases the potential by at most a constant. Hence, if we can prove that the application of any rebalancing operation decreases the potential by at least 1, the theorem follows from the completeness of the scheme. We consider each operation separately.

Both root operations *Root 1* and *Root 2* clearly decrease the potential since conflicts are removed and no new ones can emerge.

From Proposition 3, it follows that any negatively tagged node has at least one child. In *Minus One 2*, a potential of at most 2 is needed on the right hand side if the parent has a non-negative tag, but at least 3 is released if the negatively tagged node has at least two children. In the case where the negatively tagged node has only one child, the resulting node has the exact same number of children as the parent, and thus the same potential. If the parent has a negative tag, we have one pointer fewer in negatively tagged nodes on the right-hand side, but all stored in one node instead of two. Thus, the potential decreases by 1.

For the remaining *Minus One* operations, we have that at least a potential of 5 is released, since the negatively tagged node has at least 2 children (otherwise the parent would not be split), and in this case, the parent has b children. On the right hand side, we have in *Minus One 1* a potential of at most 2, since we create at

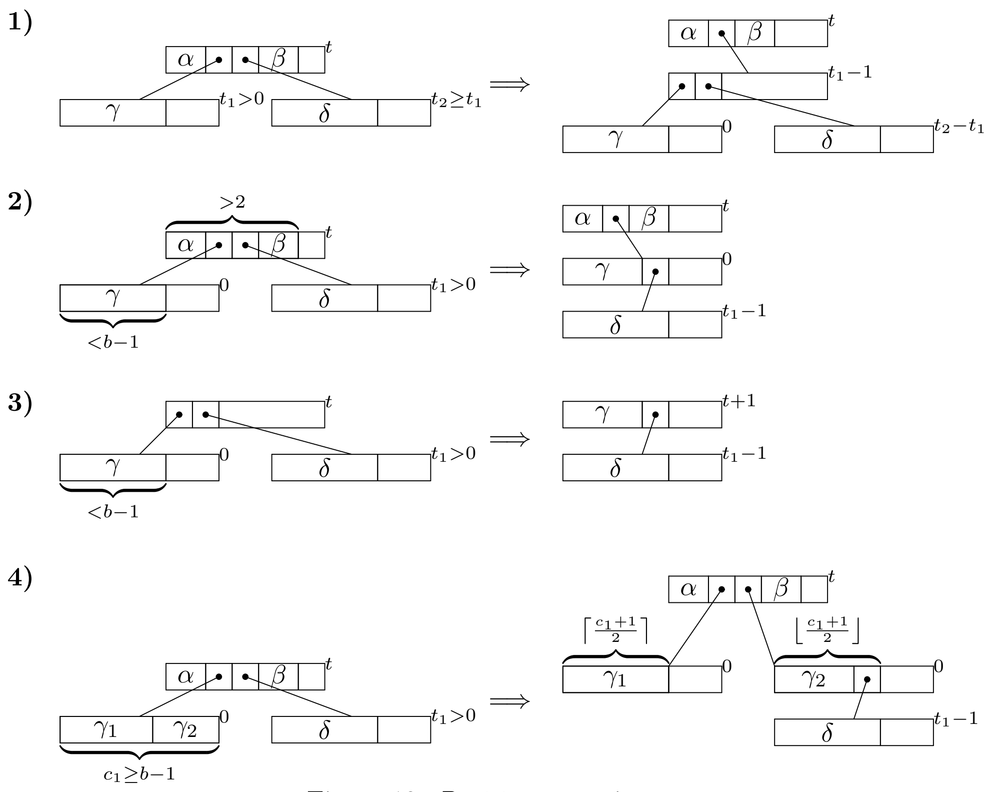

Figure 12: *Positive* operations.

most two minimally filled nodes. In *Minus One 3*, we may create a minimally filled node, and we add one pointer to a node summing to at most 3. Finally, in *Minus One 4*, we create a tag -1 node with two children, and at most one minimally filled node. Hence, we need a potential of at most 4.

In *Positive 1*, a potential of at least 6 is released from reducing the positive tags of two nodes. However, the number of pointers in one node is reduced by one, and a degree two node (with non-negative tag) is created. This increases the potential (regardless of a) by no more than 5. In *Positive 2*, the number of pointers in one node is reduced by one, but to no less than 2, while the added pointer does not increase the potential. Totally, the potential is decreased by 1. In *Positive 3*, the increase and decrease of positive tags cancel out, but a degree 2 node is replaced by a non-full node, which has had its number of pointers increased by one. Thus, totally a potential of at least 1 is released. Finally, in *Positive 4*, a positive tag is decremented at the cost of making at most two minimally filled nodes.

Since the resulting node after a *Fusion* has potential equal to or less than that of the larger of the fused nodes, the potential decreases by at least one since the potential of an underfull node is larger than the cost of decreasing the number of pointers in a node by one—regardless of a. In *Share*, an underfull node is replaced by at most two minimally filled nodes. Since the potential of a unary node is larger than the increase in potential when a positive tag is incremented, the two *Compress* operations are easily seen to decrease the potential. □

5. Experimental Results

In the previous sections, we have shown that the three proposals for (a, b)-trees with relaxed balance, the one from [20] and the chain-based and height-based from this present paper, have identical asymptotic amortized complexities.

In this section, we investigate the constants, focusing of course on search times (path lengths), but also on space utilization (creation of nodes) and parallel efficiency with number of nodes to be locked exclusively during rebalancing as a key figure.

The design of the tests reflects that the interesting applications for relaxed search trees are the ones which involve storing highly dynamic sets. We examine to which extend the structures are capable of handling a large number of updates to an initially balanced tree without increasing the lengths of search paths excessively, even if no rebalancing is carried out (simulating a period where rebalancing is temporarily "turned off"). In addition, once rebalancing is undertaken, we investigate how much work is required to make the tree balanced again, and how much of the tree is made temporarily inaccessible due to locks (in case of a parallel application). Though searching in the parallel setting also requires locks, these may be shared, whereas rebalancing involving pointer changes requires exclusive locking. These are referred to as x-locks in the tables.

Though we also register information relevant to parallel applications, our experiments are carried out in a sequential setting only.

The sizes of the trees are kept roughly constant such that measurements of the average length of search paths are comparable. All conflicts are kept in a FIFO queue, and if a dequeued conflict cannot be handled, it is added to the rear of the queue. A dequeued conflict which no longer violates the constraints is just discarded, and such an operation is not counted as a rebalancing operation.

We consider the case where keys are drawn from a uniform distribution, and another case where keys come in groups, i.e., a whole sequence of keys are inserted in the direction of a single leaf.

5.1. Uniform Updates

A tree of size 50,000 is build using insertions and the tree is rebalanced completely after each insertion. A measure for the average response time (number of nodes visited) is measured. In this experiment, as in all the following, 50,000 uniformly distributed searches are made to estimate this number.

Then 100,000 insertions of uniformly chosen elements are made, followed by 100,000 deletions of uniformly chosen elements. This is repeated five times for a total of 1,000,000 updates. Then the average response time is measured again. Finally, the tree is completely rebalanced.

Table 1 shows the results obtained. The number of rebalancing operations as well as the number of exclusive locks required in a parallel setting are counted. To compare the space usage, the final size and the total number of node allocations during the process are counted. Finally, the average path length for the balanced and unbalance tree is measured. Recall that for a balanced (a, b)-tree, by definition, the path length is the same to all leaves.

We refer to our present proposals, in order of presentation, as Chain and Height, and to the proposal from [20] as LF.

Method	Rebalancing		Nodes created		Response time	
	Operations	x-locks	Final size	Total	Balanced	Unbalanced
$a, b = 2, 4$:						
Chain	103183	243627	34937	80754	10.00	11.91
LF	152228	446898	37216	156752	10.00	12.28
Height	102852	321653	37261	105865	10.00	11.10
$a, b = 3, 6$:						
Chain	63918	148534	18491	48707	8.00	9.69
LF	90731	250396	18806	95590	8.00	10.11
Height	59460	178974	18837	62734	8.00	8.93
$a, b = 4, 8$:						
Chain	46639	107959	12520	34857	7.00	8.61
LF	64164	171898	12578	68531	7.00	9.03
Height	41740	122672	12483	45221	7.00	7.91

Table 1: Experimental comparison for uniform applications.

As can be seen from the table, the number of rebalancing operations used by Chain and Height are comparable, and roughly $\frac{2}{3}$ of the number used by LF. However, Height locks approximately 3.05 nodes per rebalancing operation on average as compared to Chain's 2.3 and LF's 2.75. For LF and Height, this number tends to decrease as a, b increase due to fewer expensive non-leaf operations. In contrast to B^{link}-trees, where only one node needs to be locked at a time [14], for all relaxed proposals, it is generally the case that all the nodes involved in an operation must be locked at the same time.

The final sizes of all the trees are comparable. However, Chain only makes 50% of the node allocations made by LF and 75% of the allocations made by Height. The average search time after 1,000,000 updates is comparable for all three algorithms, although Height seems to perform a little better than the others.

5.2. Non-Uniform Updates

We now modify the above experiment such that the presence of conflicts in the trees become more evident. Under a uniform distribution, they have a great tendency to cancel out.

Insertions are now made as follows. One element (a leaf) is chosen and a sequence of keys all from a small interval around the chosen element (and not already present in the tree) are inserted in random order. The length of the sequence is chosen to be approximately $2b^2$ such that the number of elements is significantly larger than what will fit in one leaf. In this way, a tree of size 50,000 is build. After each insertion, we apply (at most) two rebalancing operations (using the FIFO queue). This keeps the tree almost balanced.

Then 400,000 updates are made, while the average path length is measured for every 20,000 updates. The first 200,000 updates consist of alternating 10,000 insertions in sequences and 10,000 uniform deletions, with no rebalancing. Then rebalancing is resumed (again two rebalancing operations per update), while 10,000 uniform insertions alternating with 10,000 uniform deletions are made, until the 400,000 updates are reached.

This models the situation that after a burst of updates, where rebalancing has temporarily been turned off, we start rebalancing again while servicing update requests at a slower pace. The results are shown in Table 2.

Method	Rebalancing		Nodes created	
	Operations	x-locks	Final size	Total
$a, b = 2, 4$:				
Chain	142975	361433	30148	82982
LF	170534	504291	30825	153282
Height	114893	357484	30633	96007
$a, b = 3, 6$:				
Chain	93003	234566	16745	49469
LF	104876	290954	16712	93304
Height	68236	206586	16758	54918
$a, b = 4, 8$:				
Chain	68979	174053	11533	34929
LF	76536	206469	11578	67101
Height	49024	147022	11554	38178

Table 2: Experimental comparison for insertions in sequences.

The number of rebalancing operations used by Height are now clearly lower than for both Chain and LF. Despite the fact that Height locks more nodes per operation, it totally locks fewer than any of the two others. As can be observed from Figure 13, which shows the average path length at different times during the 400,000 updates, Height and Chain regain balance faster than LF, once rebalancing is undertaken. However, the average path length for Height is always the shorter.

The reason for this is that the extra nodes allocated by Height are used more efficiently since insertions arrive in bursts, and fewer sparse tag -1 nodes are created. However, Chain still allocates the smallest number of nodes.

6. Concluding Remarks

As demonstrated by the experiments, we have reached our goal of improving upon the drawbacks of the original proposal for relaxed (a, b)-trees, while preserving the property that rebalancing is amortized constant.

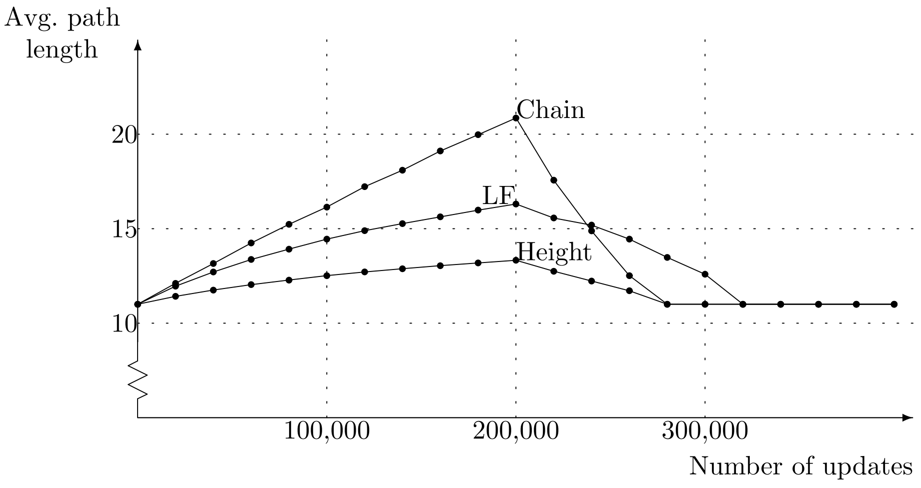

Figure 13: Average path length as function of number of updates, $a, b = 2, 4$.

In the experiments described in the test section, we have used the traditional definition of requiring that all nodes (except the root) have at least a children before we consider the tree balanced. Experiments show, however, that often in practice, it is better to ignore this requirement and only remove a node if it becomes empty [13]. We have run all tests with this strategy as well, and the strategy improves upon all the proposals with roughly the same percentage. Thus, the relative ordering remains the same and the same conclusions are reached.

For teaching and verification purposes, it is worth noting that the proof of complexity for both of the proposals in this paper, but primarily the former, are much simpler than the one given originally.

In addition, the second proposal of this paper represents a step in the direction of making rebalancing fully parallel, in the sense that any conflict can be addressed at any time without sacrificing the worst-case complexity. Attempts in that direction has been made for red-black trees. In [8, 25], a rebalancing operation may be applied to any conflict, but it has not been possible to prove a reasonable complexity result at the same time. It ought to be a topic for further research to obtain this for (a, b)-trees or in fact for any balanced tree scheme.

Finally, in [20], logarithmic results are proven which complement the amortized constant results. More precisely, it is proven that one update cannot give rise to more than roughly $\log_a N$ rebalancing operations, where N is the maximal size of the tree during the period. However, in practice, the amortized constant results, stating that rebalancing after k updates can be handled in $O(k)$ rebalancing operations, are by far the more important guarantee. Therefore, we have omitted the rather lengthy logarithmic proofs here. We have, however, verified that the proof technique from [20] can also be used here, so our two new proposals also have that property.

Acknowledgements

The second author was supported in part by the Danish Natural Sciences Research Council (SNF) and in part by the ESPRIT Long Term Research Programme of the EU under project number 20244 (ALCOM-IT).

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. R. Bayer. Symmetric Binary B-Trees: Data Structure and Maintenance Algorithms. *Acta Informatica*, 1:290–306, 1972.
3. R. Bayer and E. McCreight. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica*, 1(3):97–137, 1972.
4. Joan Boyar, Rolf Fagerberg, and Kim S. Larsen. Amortization Results for Chromatic Search Trees, with an Application to Priority Queues. In *Proceedings of the Fourth International Workshop on Algorithms and Data Structures*, volume 955 of *Lecture Notes in Computer Science*, pages 270–281. Springer-Verlag, 1995.
5. Joan Boyar, Rolf Fagerberg, and Kim S. Larsen. Amortization Results for Chromatic Search Trees, with an Application to Priority Queues. *Journal of Computer and System Sciences*, 55(3):504–521, 1997.
6. Joan F. Boyar and Kim S. Larsen. Efficient Rebalancing of Chromatic Search Trees. In *Proceedings of the Third Scandinavian Workshop on Algorithm Theory*, volume 621 of *Lecture Notes in Computer Science*, pages 151–164. Springer-Verlag, 1992.
7. Joan F. Boyar and Kim S. Larsen. Efficient Rebalancing of Chromatic Search Trees. *Journal of Computer and System Sciences*, 49(3):667–682, 1994.
8. Joaquim Gabarró, Xavier Messeguer, and Daniel Riu. Concurrent Rebalancing on HyperRed-Black Trees. In *Proceedings of the 17th International Conference of the Chilean Computer Science Society*, pages 93–104. IEEE Computer Society Press, 1997.
9. 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.
10. Sabina Hanke. The Performance of Concurrent Red-Black Tree Algorithms. In *Proceedings of the 3rd International Workshop on Algorithm Engineering*, volume 1668 of *Lecture Notes in Computer Science*, pages 286–300. Springer-Verlag, 1999.
11. S. Hanke, Th. Ottmann, and E. Soisalon-Soininen. Relaxed Balanced Red-Black Trees. In *Proceedings of the 3rd Italian Conference on Algorithms and Complexity*, volume 1203 of *Lecture Notes in Computer Science*, pages 193–204. Springer-Verlag, 1997.
12. S. Huddleston and K. Mehlhorn. A New Data Structure for Representing Sorted Lists. *Acta Informatica*, 17:157–184, 1982.
13. Theodore Johnson and Dennis Shasha. *B*-Trees with Inserts and Deletes: Why Free-at-Empty is Better Than Merge-at-Half. *Journal of Computer and System Sciences*, 47:45–76, 1993.
14. Theodore Johnson and Dennis Shasha. The Performance of Current B-Tree Algorithms. *ACM Transactions on Database Systems*, 18(1):51–101, 1993.
15. J. L. W. Kessels. On-the-Fly Optimization of Data Structures. *Communications of the ACM*, 26:895–901, 1983.

16. Vladimir Lanin and Dennis Shasha. A Symmetic Concurrent B-Tree Algorithm In *Proceedings of the Fall Joint Computer Conference*, pages 380–389. IEEE Computer Society Press, 1986.

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. To appear in *Journal of Computer and System Sciences.*

18. Kim S. Larsen. Amortized Constant Relaxed Rebalancing using Standard Rotations. *Acta Informatica*, 35(10):859–874, 1998.

19. Kim S. Larsen and Rolf Fagerberg. B-Trees with Relaxed Balance. In *Proceedings of the 9th International Parallel Processing Symposium*, pages 196–202. IEEE Computer Society Press, 1995.

20. 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.

21. Kim S. Larsen, Thomas Ottmann, and Eljas Soisalon-Soininen. Relaxed Balance for Search Trees with Local Rebalancing. In *Proceedings of the Fifth Annual European Symposium on Algorithms*, volume 1284 of *Lecture Notes in Computer Science*, pages 350–363. Springer-Verlag, 1997.

22. Kim S. Larsen, Eljas Soisalon-Soininen, and Peter Widmayer. Relaxed Balance through Standard Rotations. In *Proceedings of the Fifth International Workshop on Algorithms and Data Structures*, volume 1272 of *Lecture Notes in Computer Science*, pages 450–461. Springer-Verlag, 1997. To appear in *Algorithmica*.

23. Philip L. Lehman and S. Bing Yao. Efficient Locking for Concurrent Operations on B-Trees. *ACM Transactions on Database Systems*, 6(4):650–670, 1981.

24. Kurt Mehlhorn. *Sorting and Searching*, volume 1 of *Data Structures and Algorithms.* Springer-Verlag, 1984.

25. Xavier Messeguer and Borja Valles. HyperChromatic trees: a fine-grained approach to distributed algorithms on RedBlack trees. Tech. Report LSI-98-13-R, Departament de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya, 1998.

26. 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.

27. 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.

28. Otto Nurmi and Eljas Soisalon-Soininen. Chromatic Binary Search Trees—A Structure for Concurrent Rebalancing. *Acta Informatica*, 33(6):547–557, 1996.

29. Otto Nurmi, Eljas Soisalon-Soininen, and Derick Wood. Relaxed AVL Trees, Main-Memory Databases and Concurrency. *International Journal of Computer Mathematics*, 62:23–44, 1996.

30. Yehoshua Sagiv. Concurrent Operations on B^*-Trees with Overtaking. *Journal of Computer and System Sciences*, 33:275–296, 1986.

31. Neil Sarnak and Robert E. Tarjan. Planar Point Location Using Persistent Search Trees. *Communications of the ACM*, 29:669–679, 1986.

32. Robert Endre Tarjan. Amortized Computational Complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985.