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

# EFFICIENT REBALANCING OF B-TREES WITH RELAXED BALANCE

KIM S. LARSEN
*Department of Mathematics and Computer Science, Odense University*
*Campusvej 55, DK-5230 Odense M, Denmark*

and

ROLF FAGERBERG
*Department of Mathematics and Computer Science, Odense University*
*Campusvej 55, DK-5230 Odense M, Denmark*

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

ABSTRACT

B-trees with relaxed balance have been defined to facilitate fast updating on shared-memory asynchronous parallel architectures. To obtain this, rebalancing has been uncoupled from the updating so that extensive locking can be avoided in connection with updates.

We analyze B-trees with relaxed balance, and prove that each update gives rise to at most $\lfloor \log_a(N/2) \rfloor + 1$ rebalancing operations, where $a$ is the degree of the B-tree, and $N$ is the bound on its maximal size since it was last in balance. Assuming that the size of nodes is at least twice the degree, we prove that rebalancing can be performed in amortized constant time. So, in the long run, rebalancing is constant time on average, even if any particular update could give rise to a logarithmic number of rebalancing operations. We also prove that the amount of rebalancing done at any particular level decreases exponentially going from the leaves towards the root. This is important since the higher up in the tree a lock due to a rebalancing operation occurs, the larger a subtree which cannot be accessed by other processes for the duration of that lock.

All of these results are in fact obtained for the more general $(a, b)$-trees, so we have results for both of the common B-tree versions as well as 2-3 trees and 2-3-4 trees.

*Keywords:* search trees; parallel environment; relaxed B-trees; (a,b)-trees; rebalancing; amortized analysis.

## 1. Introduction

When B-trees[1] are used in an asynchronous parallel environment, locks must be applied when nodes are updated or when some rebalancing operations are carried out. If the strict balance conditions known from the sequential case are used, then these locks will be a major obstacle for the searching and updating processes.

This is the approach used by Samedi[2] in the first and probably most simple attempt at adapting B-trees to a parallel environment. Using semaphores,[3] all the nodes on the path from the root down to an update are locked. The obvious problem with this approach is that nodes close to the root are locked very frequently and for long periods of time, thereby preventing a high degree of parallelism. There are ways to improve on this without fundamentally changing the idea. In a proposal by Kwong and Wood,[4] locks are kept up to the first insertion or deletion safe node, where a node is insertion safe if it is not full and deletion safe if one deletion will not make it underfull. When searching from the root, nothing is locked until the deepest safe node is encountered. The number of locks can still be proportional to the height of the tree, though.

Lehman and Yao[5] introduce an implementation technique, where at any time only a constant number of locks have to be kept for any single update. Overflow is still taken care of by the inserting process. Sagiv[6] improves slightly on these results by using fewer locks and by allowing background processes to rebalance using compressions.

Finally, in a proposal by Nurmi, Soisalon-Soininen, and Wood,[7] rebalancing is separated entirely from the updating. A "tag bit" is used to register unbalance, and background processes deal with problems of unbalance in parallel with searches and updates. There are several advantages to this scheme. Instead of the standard locking of whole paths or stepwise locking down paths, the rebalancing processes will now do only $O(1)$ work at a time before they release locks and move on to another problem. This implies that, in principle, $O(n)$ processors can simultaneously access the tree, where $n$ is the size of the tree, since searching does not require exclusive locking. However, to obtain this, rebalancing must be kept down to a constant number of operations per update. Another advantage of the uncoupling is that it is also possible to postpone all or parts of the rebalancing until after peak working hours. The disadvantage, of course, is that the tree can totally degenerate if there are not enough background processes to do the rebalancing. The major problem, which this paper has in common with most of the papers mentioned above, is that none of them actually analyze the complexity of their proposals, or test their proposal in comparisons with proposals of others.

However, the proposal of Nurmi, Soisalon-Soininen, and Wood[7] is the most promising, but it turns out that, in addition to the lack of a proof of complexity, there are a few other problems. Their operations can lead to inconsistencies and some unfortunate design decisions have been made, which lead to a worse complexity than is necessary. We present a corrected set of operations, and, more importantly, we accompany our proposal by proofs of complexity, the importance of which is emphasized by the well known fact that minor changes in the restrictions on the parallel operations can lead to bounds on rebalancing that are order of magnitudes greater than necessary. In a specific example concerning search trees,[12] one small modification changed the complexity of carrying out $n$ operations from $\Omega(n^2)$ to $O(n \log n)$.

Uncoupling was first discussed in connection with red-black trees[8] by Guibas and Sedgewick,[8] and later in connection with AVL trees[9] by Kessel.[10] The first fully uncoupled proposal in connection with red-black trees (called chromatic trees) is by Nurmi and Soisalon-Soininen.[11] This was later improved upon by Boyar and Larsen,[12] and the new proposal was accompanied by a proof of complexity. A variation of that proposal has been implemented by Malmi[14] and tested on a large scale. Preliminary results imply that a high degree of parallelism can be obtained as a result of uncoupling updating and rebalancing. Further complexity results for chromatic trees have been obtained by Boyar, Fagerberg, and Larsen.[13] In connection with AVL trees, a fully uncoupled proposal was given by Nurmi, Soisalon-Soininen, and Wood.[7] This was later analyzed and improved.[15]

All of the results in this paper are obtained for $(a,b)$-trees, so we also have results for all of the special cases, including both of the common B-tree versions as well as 2-3 trees and 2-3-4 trees. We prove that each update gives rise to at most $\lfloor \log_a(N/2) \rfloor + 1$ rebalancing operations, where $N$ is the bound on the maximal size of the $(a,b)$-tree since it was last in balance. Assuming that $b \geq a$, we prove that rebalancing can be performed in amortized constant time. We also prove that the amount of rebalancing done at any particular level decreases exponentially going from the leaves towards the root.

## 2. B-Trees and $(\mathbf{a},\mathbf{b})$-Trees

In this paper, we consider a generalization of B-trees called $(a,b)$-trees or weak B-trees.[16] An $(a,b)$-tree is an ordered tree with minimal node size $a$ and maximal node size $b$ for integers $a$ and $b$ with $b \geq 2a-1$. The trees in this paper are *leaf-oriented.* This means that keys inserted by the user are stored in the leaves. The internal nodes only contain pointers to subtrees along with the necessary routing information, which consists of copies of user inserted keys. However, the keys in the internal nodes are not necessarily present in the leaves, since we do not want to update the routing information when keys are deleted. Assuming that an internal node (also called a block or a page) has $j$ children, the content of a node can be illustrated by the sequence $P_1K_1P_2K_2\cdots K_{j-1}P_j$, where the $K_i$'s are routers (keys) and the $P_i$'s are pointers. Routers are listed in increasing order, i.e., $K_1 < K_2 < \cdots < K_{j-1}$. Keys in the subtree pointed to by $P_i$ are less than or equal to $K_i$, whereas the keys in the subtree pointed to by $P_{i+1}$ are greater than $K_i$.

If $u$ is a node, which is not the root, we let $\pi u$ denote the parent of $u$. The *level* of a node $u$ is now defined as follows:

$$l(u) = \begin{cases} 0, & \text{if } u \text{ is the root,} \\ l(\pi u)+1, & \text{otherwise.} \end{cases}$$

Additionally, we let $c(u)$ denote the number of keys in $u$, if $u$ is a leaf, and the number of pointers in $u$, otherwise.

For integers $a$ and $b$, where $b \geq 2a-1$, an $(a,b)$-tree must fulfill the constraints:

- for any pair of leaves $u_1$ and $u_2$, we have that $l(u_1) = l(u_2)$.
- if $u$ is the root[a], then $2 \leq c(u) \leq b$.
- if $u$ is not the root, then $a \leq c(u) \leq b$.

Obviously, the purpose of these criteria is to keep the tree balanced and reasonably dense such that access times of $\log_a(n)$ can be maintained, where $n$ is the size of the tree. For this purpose, two rebalancing operations are defined: *compress* operations have to be used after a deletion, which causes a violation of the density criteria, and *split* operations have to be used when a key has to be inserted into a full node. The underlying assumption, which is the reason for using an $(a, b)$-tree instead of a red-black tree or an AVL tree, is that it is not more expensive to rewrite a whole node than to rewrite a single key in the node. This is of course only true when a node is a sector on a disc, or something similar. B-trees are special cases of $(a, b)$-trees, where $b = 2a - 1$ (according to some authors, $b = 2a$). Further introduction to B-trees can be found elsewhere.[17]

## 3. Relaxed Balance

We now state the definition of $(a, b)$-trees with relaxed balance, also called a *relaxed* $(a, b)$-tree. This structure, along with a number of update operations, was originally proposed by Nurmi, Soisalon-Soininen, and Wood[7] (for B-trees).

The purpose of the relaxed balance conditions to be presented below is to allow updates to be performed without having to rebalance immediately. If that was the sole goal, then the conditions should be as weak as possible. However, we would also like to be able to rebalance fast, when we eventually decide to do it. Obviously, we would want as much information as possible available at this point. As an extreme, without any information at all, the whole tree would have to be built anew.

Nodes in an $(a, b)$-tree with relaxed balance are equipped with a *tag* value, which is either 0 or $-1$. The *relaxed* level of a node is defined by:

$$rl(u) = \begin{cases} tag(u), & \text{if } u \text{ is the root,} \\ rl(\pi u) + 1 + tag(u), & \text{otherwise.} \end{cases}$$

For integers $a$ and $b$, where $b \geq 2a - 1$, an $(a, b)$-tree with relaxed balance must fulfill the constraints:

- for any pair of leaves $u_1$ and $u_2$, we have that $rl(u_1) = rl(u_2)$.
- if $u$ is a leaf, then $0 \leq c(u) \leq b$.
- if $u$ is not a leaf, then $1 \leq c(u) \leq b$.
- if $u$ is a leaf, then $tag(u) = 0$.

[a] Actually, if the root $u$ is also a leaf, we must allow $c(u) = 1$, as this is the only way to represent a tree with one key. This will be of no concern in this paper, and it will not be mentioned further.

So, leaves are allowed to become completely empty, whereas internal nodes must contain at least one pointer.

Clearly, an ordinary $(a,b)$-tree is also a relaxed $(a,b)$-tree (assuming that each node $u$ in the $(a,b)$-tree has $tag(u) = 0$). When a relaxed $(a,b)$-tree fulfills the balance conditions for ordinary $(a,b)$-trees, we will refer to it as a *standard* $(a,b)$-tree.

The update operations, insert and delete, are described below.

Insert: First, the correct node $u$ is found by searching as usual. If there is room in the node, i.e., if $c(u) < b$, then the new key can be inserted directly into its correct place. Otherwise, $u$ is replaced by three nodes arranged as follows: the top node is given the tag value $-1$, and it has exactly two children. These two children both have tag values 0 and they share all the data which were in $u$ before the insertion along with the new key. So, one of these nodes receives $\lfloor \frac{b+1}{2} \rfloor$ keys; the other receives $\lceil \frac{b+1}{2} \rceil$. A copy of the rightmost key in the left child is inserted in the top node as a router. This is illustrated in figure 1 (for clarity, we only show pointers in the figures, i.e., no keys).

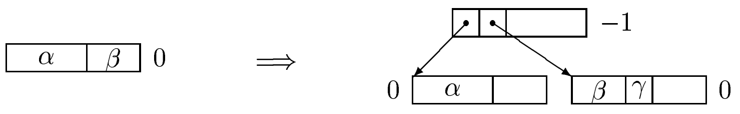


Figure 1: Insertion in case of overflow.

Delete: First, the correct node $u$ is found by searching as usual. Then the key, if present, is deleted.

Notice that the update operations do not cause a violation of the relaxed balance conditions, but an insertion might violate the balance criteria for standard $(a,b)$-trees, and a deletion might violate the density criteria for standard $(a,b)$-trees.

## 4. Rebalancing Operations

We now present the collection of rebalancing operations, which is a modification of the collection from Nurmi, Soisalon-Soininen, and Wood.[7] The purpose of these operations is to gradually transform an $(a,b)$-tree with relaxed balance into a standard $(a,b)$-tree. There are two types of problems to deal with: a node $u$ may be *negative*, i.e., $tag(u) = -1$, or a node $u$ may be *underfull*, i.e., $c(u) < a$ (in the case of the root, $c(u) < 2$). For technical reasons, if a node $u$ is negative, we do *not* designate it underfull, even if $c(u) < a$.

Root operations: If the root has tag value $-1$, then its tag value can be set to 0. If the root has only one child, then the root is deleted and the single child is made the new root. If both problems are present, then these will be dealt with in one operation.

Split: If a node $u$, which is not the root, has $tag(u) = -1$, and its parent has tag value 0, then $u$ (and thus also the tag value) can either be deleted or the negative tag value can be moved closer to the root:

If $c(u) + c(\pi u) \leq b + 1$, then all the pointers from $u$ are moved into $\pi u$, and the routing information is updated (it is $b+1$, and not $b$, since the pointer to $u$

will no longer be necessary if the entire contents of $u$ are moved to $\pi u$ and $u$ is deleted). This case is illustrated in figure 2. If $c(u) + c(\pi u) > b + 1$, then $u$ and $\pi u$

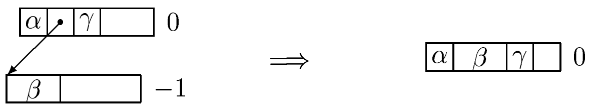


Figure 2: Split: tag adjustment when $|\alpha| + |\beta| + |\gamma| \le b$.

are replaced with three nodes: the top node is given the tag value $-1$, and it has exactly two children. These two children both have tag values 0 and they share all the pointers which were in $u$ and $\pi u$ before the operation. A copy of the rightmost key in the left child is inserted in the top node as a router. This case is illustrated in figure 3.

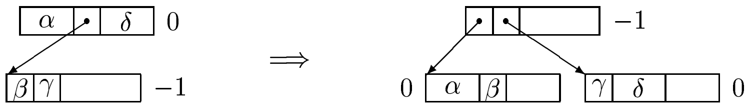


Figure 3: Split: tag adjustment when $|\alpha| + |\beta| + |\gamma| > b$.

Compress: Assume that $u$ is a node such that $c(u) < a$, and such that $u$ has a left or right sibling $v$ with $tag(v) = tag(u) = 0$. If $c(u)+c(v) \ge 2a$, then the pointers from $u$ and $v$ are distributed evenly among these two nodes. This case is illustrated in figure 4. Otherwise, all the pointers from $u$ are moved into $v$ and $u$ is deleted.

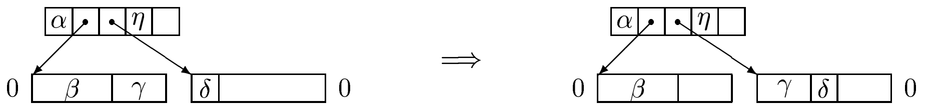


Figure 4: Compress: density adjustment when $|\beta| + |\gamma| + |\delta| \ge 2a$.

The parent is updated to reflect the movement of pointers. This case is illustrated in figure 5. The threshold is $2a$, and not $b + 1$, for example, to avoid removing $u$ whenever possible, as we do not want unnecessary propagation of underfull nodes upwards in the tree.

The split operations are also called *tag adjusting* rebalancing operations while compress operations are also referred to as *density adjusting* rebalancing operations. A root operation may be either.

When these update and rebalancing operations are applied in an asynchronous parallel environment, nodes must be locked to ensure that processes do not interfere with each other. All the operations used here are local, and only directly involved nodes need to be locked. A more elaborate discussion of this is given by Nurmi and Soisalon-Soininen.[11] Problems arising due to an update, or problems that are created as a side-effect of adjusting tag values or removing an underfull node problem, can be put into a problem queue by the process which creates the problem.

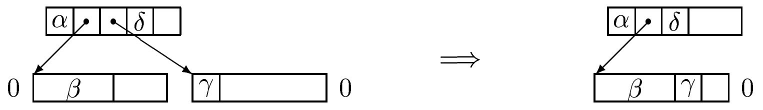


Figure 5: Compress: density adjustment when $|\beta| + |\gamma| + |\delta| < 2a$.

This, along with give-up techniques for rebalancing, is described in more detail by Goodman and Shasha.[18] Often, when working with more than one process, fairness is an issue. This is not the case here. If two rebalancing processes are trying to lock the same nodes, one will give up and instead fix another problem elsewhere in the tree.

In concluding this section on rebalancing operations, we discuss some of the problems in the original proposal by Nurmi, Soisalon-Soininen, and Wood.[7] To be fair, we are convinced that most of these problems are omissions rather than errors.

In that proposal, there are no requirements which must be fulfilled in order for a compress operation to be applied. If the two nodes in question, $u$ and its sibling $v$, do not have the same tag value, then a compress operation will create a tree where the relaxed balance conditions are violated.

It is not clear how one should fix the problem just described. One could possibly add the constraint that $tag(u) = tag(v)$. Then no violation can be introduced by the operation. It turns out, though, that allowing compressions when $tag(u) = tag(v) = -1$ increases the complexity of the rebalancing. Split operations will take care of nodes which have tag values $-1$, so compress operations should not be wasted on such problems. As is apparent in our proposal, the solution is to require $tag(u) = tag(v) = 0$.

There is another problem with the compress operation in their proposal.[7] They do not consider the possibility that an underfull node might not have a sibling, which would make a compress operation impossible. Additionally, they do not have any constraints on the split operation, but obviously, it must be required that $tag(\pi u) = 0$, where $u$ is the node with negative tag value to which a split operation should be applied. Without this requirement on the parent node, the relaxed balance conditions would be violated.

Given that there are a number of restrictions on when the various rebalance operations can be applied, it is necessary to prove that rebalancing is always possible, i.e., that the rebalancing operations listed are sufficient:

**Lemma 1** *If $T$ is a relaxed $(a, b)$-tree, but not a standard $(a, b)$-tree, then either a root operation, a split operation, or a compress operation can be applied.*

**Proof.** If $T$ is not a standard $(a, b)$-tree, then either there is a node with tag value $-1$ or there is an underfull node.

Assume that there is a node $u$ such that $tag(u) = -1$. If the root has tag value $-1$, then a root operation can be applied. Otherwise, the tag value of the root is 0. Let $v$ be the first node, which has tag value $-1$, on the path down from the root to $u$. Then $v$ cannot be the root, and $tag(\pi v) = 0$, so a split operation can be applied.

Now, assume that there are not any nodes with tag value $-1$. Then there must be an underfull node $u$. If the root has exactly one child, then a root operation can be applied. So, assume that the root has at least two children. Let $v$ be the first underfull node on the path from the root down to $u$. Then $v$ must have a sibling. If not, then $\pi v$ would have $v$ as its only child, and would also be underfull, which would imply that $v$ was not the first underfull node. Both $v$ and its sibling must have tag values 0, since we assumed that there were no $-1$'s. This means that a compress operation can be applied. □

## 5. Complexity

Having proved that on any relaxed $(a,b)$-tree, which is not a standard $(a,b)$-tree, at least one rebalancing operation can be applied, the question arises as to how many such operations can take place before we arrive at a standard $(a,b)$-tree.

For the complexity analysis in this section, we follow Nurmi, Soisalon-Soininen, and Wood[7] in assuming that initially the search tree is a standard $(a,b)$-tree, and then a series of search, insert, and delete operations occur. These operations may be interspersed with rebalancing operations. The rebalancing operations may also occur after all of the search and update operations have been completed; our results are independent of the order in which the operations occur. In any case, the $(a,b)$-tree is always an $(a,b)$-tree with relaxed balance, and after enough rebalancing operations, it will again be a standard $(a,b)$-tree.

If some of the operations are done in parallel, they must involve sets of nodes which are completely disjoint from each other. The effect will be exactly the same as if they were done sequentially, in any order. Thus throughout the proofs, we will assume that the operations are done sequentially. At time 0, there is a standard $(a,b)$-tree, at time 1 the first operation has just occurred, at time 2 the second operation has just occurred, etc.

In the following, we need the concepts of *height*, $h(u)$, and *relaxed height*, $rh(u)$, of a node $u$. The height of the root of an $(a,b)$-tree is defined to be the level of its leaves. The height $h(u)$ of a general node $u$ is the height that the subtree rooted at $u$ has if it is detached from the tree of which it is a part. The relaxed height of a node is defined similarly from the relaxed level of the leaves.

Clearly, if a node $u$ in an $(a,b)$-tree has a large height, then the subtree in which $u$ is the root will also contain many keys. In a relaxed $(a,b)$-tree, however, if a node has a large relaxed height, then this is only a sign of there having been many keys at some point; they may have been deleted since. It turns out to be useful to count those keys below $u$ which have been deleted. We do this, remembering every key that ever existed by associating them with nodes currently in the tree.

To begin with, every key is associated with the node it is currently in. When a deletion occurs, the deleted key is still associated with the node it was deleted from. When a node is deleted, all keys associated with that node are instead associated with the parent node immediately before the deletion. An *A-subtree* of a node $u$ is now all keys associated with nodes in the actual subtree of $u$.

**Lemma 2** *Assume that a number of updates and rebalancing operations are per-*

*formed on a relaxed $(a,b)$-tree, which was initially a standard $(a,b)$-tree. At any time the following holds: If $u$ is the root in the relaxed $(a,b)$-tree, then there are at least $2a^{rh(u)}$ keys in the A-subtree of $u$. If $u$ is any other node, then there are at least $a^{rh(u)+1}$ keys in the A-subtree of $u$.*

**Proof.** By induction on time. The base case is when no operations have been performed on the tree, which is then still a standard $(a,b)$-tree. In this case, the A-subtree of any node $u$ equals the subtree of $u$, and $rh(u) = h(u)$. Clearly, an $(a,b)$-tree of height $h$ with the minimum number of children, $a$, from any internal node has at least $a^{h+1}$ keys in the leaves, except that the root may have only two children, so its subtree can only be guaranteed to contain $2a^h$ keys.

For the induction step, we assume that the result holds at some time $t$, and prove that no matter which operation is carried out, then the result still holds at time $t+1$.

Insert: If there is room in the node in which we want to insert the new key, then the only change is that the A-subtree of this node and all ancestor nodes grow. If there is not room, then the number of keys in this node, including the new key to be inserted, must be $b+1$, which is larger than or equal to $2a$. Thus, the two new nodes with tag value 0 will get at least $a$ keys each, so their A-subtrees are large enough to match their relaxed height, which is also 0. The parent node has tag value $-1$, so its relaxed height is also 0. Thus, the A-subtree of that node contains almost twice as many keys as required. The only other change is that ancestors of these three nodes get larger A-subtrees while their relaxed heights remain unchanged.

Delete: The deleted key is still associated with the node in question, so there are no changes.

Root operations: If the root $u$ has only one child, then this child is made the new root, and the result follows since this child had a large enough A-subtree before the operation was carried out. Otherwise, if there are at least two children, then the tag value $-1$ is changed to zero. Since each child had an A-subtree of size at least $a^{rh(u)}$, the result follows.

Split: Recall that the problem node $u$ with tag value $-1$ cannot be a leaf. If there is room in the parent node, then the children of $u$ are moved into the parent node, and the problem node is deleted. As $u$ had tag value $-1$, this operation will not change the relaxed height of the parent node. Clearly, the sizes of its and its ancestors' A-subtrees remain unchanged. If there is not room in the parent node, then $c(u) + c(\pi u) - 1 \geq b + 1$. We replace $u$ and $\pi u$ with three nodes. Two nodes, $u_1$ and $u_2$, are given at least $a$ of these pointers each and their tag values are set to 0. By the induction hypothesis, their (at least) $a$ children all had A-subtrees which were large enough before this operation. So, obviously, $u_1$ and $u_2$ also have large enough A-subtrees. The third node has exactly two children, $u_1$ and $u_2$, and is given the tag value $-1$. Therefore, it has exactly the same relaxed height as $u_1$ and $u_2$, and the result follows since it contains the A-subtree of $u_1$ (and $u_2$ as well).

Compress: In the compress operation, the underfull node $u$ has a sibling $v$ such that $tag(u) = tag(v)$. If $c(u) + c(v) \geq 2a$, then the pointers from $u$ and $v$ are distributed evenly among $u$ and $v$. The only potential problem here, is that the

A-subtree of $v$ could become too small. However, if $v$ is a leaf, then $tag(v) = 0$, so $rh(v) = 0$ as well. This means that $a$ keys are sufficient. If $v$ is not a leaf, then it will have at least $a$ children after the operation. Using the induction hypothesis, these children all had large enough A-subtrees before this compress operation was applied. So, with at least $a$ children, $v$ must have a large enough A-subtree after the operation.

If $c(u) + c(v) < 2a$, then the contents of $u$ are moved to $v$ and $u$ is deleted. This will increase the size of the A-subtree of $v$ while its relaxed height remains unchanged. The ancestors of $u$ and $v$ will still have the same A-subtrees as before. No other nodes are affected (as $u$ is deleted from the tree, nothing has to hold for $u$). $\square$

If $T$ is the initial standard $(a, b)$-tree and $|T|$ is its size, then after a number of update operations, $i$ of which are insertions, $|T| + i$ is a bound on the size (number of keys) of the relaxed $(a, b)$-tree at any point during these updates. Let $N = |T| + i$.

**Corollary 1** *The relaxed height of any node in a relaxed $(a, b)$-tree is at most $\lfloor \log_a(N/2) \rfloor$.*

The intuition in the design of logarithmic rebalancing is that if a problem cannot be removed immediately, then it is removed at the cost of introducing another problem closer to the root (at the parent node). The potential problem in a parallel environment is that interference between different operations may create problems. One could easily imagine operations which would cancel each other, for instance. The theorem below shows that such negative interference does not occur with the operations defined in this paper.

**Theorem 1** *After $k$ updates in a relaxed $(a, b)$-tree, which was originally a standard $(a, b)$-tree $T$, at most $k(\lfloor \log_a(N/2) \rfloor + 1)$ rebalancing operations can be applied.*

**Proof.** First, we bound the number of operations that can be carried out due to negative nodes.

An insertion may create one node with tag value $-1$. Furthermore, a split operation in removing one node with tag value $-1$, may create another—we say that the problem (the tag value $-1$) has been *moved.* Thus, only insertion creates a problem with a negative tag value; a split just moves it. Notice that if the relaxed height of the node with tag value $-1$ before the operation was $h$, then the tag value $-1$ is moved to a node of relaxed height $h + 1$. Also, no other operation which involves a node with negative tag value will change the height of the node with this negative tag value. Thus, it will keep its relaxed height until it is either deleted or moved by another split operation.

When an insertion creates a negative tag, the node created will have relaxed height 0. Since the relaxed height of the tag increases every time it is moved, by corollary 1, it can be moved at most $\lfloor \log_a(N/2) \rfloor$ times. After that, since it cannot be moved again, it must disappear. Counting that operation as well, it takes at most $\lfloor \log_a(N/2) \rfloor + 1$ rebalancing operations to remove a negative tag value.

Next, we bound the number of operations that can be carried out due to underfull nodes.

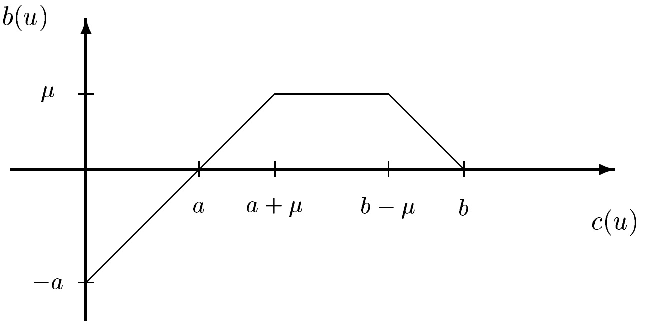
A deletion may create an underfull node. Furthermore, when a compress operation moves all the pointers from one node $u$ into another $v$, and then deletes $u$, the parent node will have one less child. As an effect, it may become underfull—but only if the parent has tag value 0 (since, by definition, negative nodes are not called underfull). Again, we will say that an underfull problem has been moved from $u$ to $\pi u$. Since the parent node has tag value 0, $rh(\pi u) = rh(u) + 1$.

As no operation decreases the relaxed height of any node, and as underfull nodes are created with relaxed height at least 0 (since leaves have tag values 0), it follows from corollary 1 that an underfull node can be moved by a compress operation at most $\lfloor \log_a(N/2) \rfloor$ times. After that, since it cannot be moved again, it must disappear. Counting that operation as well, it takes at most $\lfloor \log_a(N/2) \rfloor + 1$ rebalancing operations to fix an underfull problem. $\square$

## 6. Amortized Complexity Results

In this section, we prove that in the amortized sense, $O(1)$ rebalancing operations per update are enough to keep a relaxed $(a, b)$-tree balanced, if $b \geq 2a$. For ordinary $(a, b)$-trees, this was proven by Huddleston and Mehlhorn,[16,19] and the proof given here follows the same lines, with the necessary modifications due to the relaxation of the balance rules and the associated new rebalancing operations.

Following the exposition by Mehlhorn[19], we define the *balance* $b(u)$ of a node $u$ (different from the root) to be the function of $c(u)$, the graph of which is depicted below.



If $u$ is the root, then 2 is used instead of $a$. The slopes of the three line segments of the graph are, from left to right, 1, 0, and $-1$. The constant $\mu$ depends on $a$ and $b$, and is defined by

$$\mu = \begin{cases} a-1, & \text{if } 2a-1 \leq \lfloor \frac{b+1}{2} \rfloor, \\ \lceil \frac{b+1}{2} \rceil - a, & \text{otherwise.} \end{cases}$$

This definition of $\mu$ is equivalent to the one used by Mehlhorn.[19] The important properties of $\mu$ are the following:

**Lemma 3** *For $b \geq 2a$ and $a \geq 2$,*

- $1 \leq \mu \leq a - 1$.
- $a + \mu \leq 2a - 1 \leq b - \mu$.

- *If* $c(u) = \lfloor \frac{b+1}{2} \rfloor$ *and* $c(v) = \lceil \frac{b+1}{2} \rceil$, *then* $2\mu - 1 \le b(u) + b(v) \le 2\mu$.

**Proof.** The proof consists of elementary manipulations and is omitted (for verification of the third property, note that the double inequality is equivalent to the statement that at least one of $u$ and $v$ has maximal balance). □

We define the *total balance* $B(T)$ of an $(a, b)$-tree $T$ to be

$$B(T) = \sum_{u \in \mathcal{N}^0(T)} b(u) \; + \sum_{u \in \mathcal{N}(T)} tag(u),$$

where $\mathcal{N}(T)$ denotes the set of nodes of $T$ and $\mathcal{N}^0(T)$ denotes the set of nodes of $T$ having tag value 0. Using $B$ as our potential function, we can prove the following theorem:

**Theorem 2** *Assume* $b \ge 2a$ *and* $a \ge 2$. *If* $i$ *insertions and* $d$ *deletions are performed on a relaxed* $(a, b)$*-tree, which was originally a standard* $(a, b)$*-tree* $T$ *containing* $n$ *elements, then at most*

$$(2 + \frac{1}{a})i + 2d + \frac{n}{a} + 1$$

*rebalancing operations can occur.*

**Proof.** Insertions, deletions, and rebalancing operations can change $B(T)$. Denoting this change by $\Delta B$, we claim that

- For an *insertion* or a *deletion*, $\Delta B \ge -1$.
- For a *tag adjusting operation which creates a new node*, $\Delta B \ge 2\mu - 1$.
- For a *density adjusting operation which removes a node*, $\Delta B \ge \mu$.
- For any *other operation*, $\Delta B \ge 0$.

These claims are proven as follows:

Insertions and deletions: For deletions and for insertions without overflow, only one node $u$ is changed. As $c(u)$ either increases or decreases by one, the claim is obvious from the graph of $b(u)$. For insertions with overflow, two new nodes $u$ and $v$ are created with $c(u) = \lfloor \frac{b+1}{2} \rfloor$ and $c(v) = \lceil \frac{b+1}{2} \rceil$. The overflowing node had a balance of zero before the operation. Thus, by lemma 3, $\Delta B \ge (2\mu - 1) - 1 \ge 0$.

Tag adjusting operations which create a new node: Note that negative nodes never have more than two children: they are created in this way, and no operation can increase their number of children, as can be seen by inspection of the operations (in particular, this holds for insertions because leaves always have tag value zero). Thus, for these tag adjusting operations, the lower two nodes $u$ and $v$ must have $c(u) = \lfloor \frac{b+1}{2} \rfloor$ and $c(v) = \lceil \frac{b+1}{2} \rceil$ after the operation, and the upper node in the operation must have balance zero before the operation. By lemma 3, $\Delta B \ge 2\mu - 1$.

Density adjusting operations which remove a node: Denote the removed node by $u$ and its sibling by $v$. For this operation, we have $c(u) \le a - 1$ and $c(u) + c(v) \le 2a - 1$. The removal of $u$ increases $B(T)$ by an amount $\Delta b(u) = a - c(u)$. Define

$x$ by $c(v) = a + \mu - x$. If $c(u) \geq x$, the balance of $v$ changes by an amount $\Delta b(v) \geq x$ (this also holds for $x$ negative, since then $\Delta b(v) = 0$ by lemma 3). As $c(u) + c(v) \leq 2a - 1$ is equivalent to $(a - \Delta b(u)) + (a + \mu - x) \leq 2a - 1$, we obtain that $\mu + 1 \leq \Delta b(u) + x \leq \Delta b(u) + \Delta b(v)$. If $c(u) < x$, then $\Delta b(v) = c(u)$, so in this case $\mu + 1 \leq a = (a - c(u)) + c(u) = \Delta b(u) + \Delta b(v)$ by lemma 3. The balance of the upper node in the operation can decrease by at most one. Thus, $\Delta B \geq \mu$.

Other operations: For tag adjusting operations which remove a node, the lower node is negative and thus has at most two children, as noted above. Hence, the number of children of the upper node increases by at most one. Since the negative node is removed, $B(T)$ cannot decrease. For a density adjusting operation which does not remove a node, assume that $k$ children are transferred from the node $v$ to the node $u$. As $u$ was underfull, its balance increases by $\min\{k, \mu + 1\}$ (after the operation, $c(u) \leq b - \mu$, since the children are shared equally between $u$ and $v$). As $v$ is not underfull after the operation, its balance decreases by at most $\min\{k, \mu\}$. Hence, $B(T)$ cannot decrease. Obviously, root operations can only increase $B(T)$.

Thus the claims are proven. Denote by $\tau$ the number of tag adjusting operations which create a new node, and by $\delta$ the number of density adjusting operations which remove a node. If $T_1$ is the initial tree and $T_2$ the final tree, the claims above imply

$$B(T_1) - (i + d) + (2\mu - 1)\tau + \mu\delta \leq B(T_2).$$

As $T_1$ is a standard $(a, b)$-tree, $B(T_1) \geq 0$. By lemma 1 and theorem 1, enough rebalancing operations will eventually remove all balance problems, so we may without loss of generality assume that $T_2$ is a standard $(a, b)$-tree. For such trees, all tags are 0, making the potential function $B$ used here equal to the one used by Mehlhorn.[19] Thus, his upper bound on the potential function applies here, yielding $B(T_2) \leq \mu + \mu\frac{n+i-d-2}{a+\mu-1}$. Since $\mu \geq 1$, we have $2\mu - 1 \geq \mu$. Hence,

$$\delta + \tau \leq 1 + \frac{n + i - d - 2}{a + \mu - 1} + \frac{i + d}{\mu} \leq 1 + (1 + \frac{1}{a})i + d + \frac{n}{a}$$

To bound the rest of the rebalancing operations, we note the following. For an underfull node $u$, call $a - c(u)$ (in the case of the root, $2 - c(u)$) its *amount of underfullness*. The total amount of underfullness in the tree can only increase due to deletions—by at most one for each deletion (recall that, by definition, negative nodes are not called underfull). The total number of negative nodes can only increase due to insertions—by at most one for each insertion. All the remaining rebalancing operations (root operations and the remaining cases of tag adjusting and density adjusting operations) either reduce the total amount of underfullness in the tree or the number of negative nodes by at least one. Thus, the remaining rebalancing operations are bounded by $i + d$. This proves the theorem. $\square$

The theorem above can be paraphrased by saying that when the number of updates is $\Omega(n)$, then there are only $O(1)$ rebalancing operations per update. This applies in particular to the situation where the initial structure is empty, as it is customary to assume when discussing amortized complexity results.

Huddleston and Mehlhorn,[16,19] also prove that in ordinary $(a, b)$-trees, the number of updates that require rebalancing to occur $h$ levels up in the tree is an exponentially decreasing function of $h$. This kind of result is of particular importance in a parallel environment, since the higher up in the tree a lock due to a rebalancing operation occurs, the larger the subtree which cannot be accessed by other processes for the duration of that lock.

Like in theorem 2, the proof of this result can also be adapted to relaxed $(a, b)$-trees. To do this, we need a definition of *relaxed height* slightly different from that of section 5. With the definition in section 5, a negative node has the same relaxed height as its first descendant having tag value 0. In this section, we will need a negative node to have the same relaxed height as its first ancestor having tag value 0. This can be accomplished by defining the *variant relaxed height* $rh'(u)$ of a node $u$ by $rh'(u) = rl(l) - rl(u)$, where $rl$ is the *relaxed level* defined in section 2, and $l$ is any leaf (recall that all leaves have the same relaxed level). For the rest of this section, the term relaxed height will refer to $rh'$.

We define the *relaxed height of an operation* to be the relaxed height, before the operation occurs, of the lower nodes in the operation, except for root operations, where we define it to be the relaxed height of the root. Note that the relaxed height of tag adjustment operations and root operations is at least one, and that the relaxed height of the other operations is at least zero. We can prove the following:

**Theorem 3** *Assume $b \geq 2a$ and $a \geq 2$. If $i$ insertions and $d$ deletions are performed on a relaxed $(a, b)$-tree, which was originally a standard $(a, b)$-tree $T$ containing $n$ elements, then at most*

$$\frac{2i + n}{(\mu + 1)^h}$$

*rebalancing operations can occur at relaxed height $h$ for $h \geq 1$. For $h = 0$, the bound is $d$.*

**Proof.** For an $(a, b)$-tree $T$, we define $B_h(T)$, the *balance at relaxed height $h$*, to be

$$B_h(T) = \sum_{u \in \mathcal{N}_h^0(T)} b(u) + \sum_{u \in \mathcal{N}_h(T)} tag(u),$$

where $\mathcal{N}_h(T)$ is the set of nodes in $T$ of relaxed height $h$, and $\mathcal{N}_h^0(T)$ is the set of nodes in $T$ of relaxed height $h$ having tag value 0. An operation of a given relaxed height can change some of the $B_h$'s. We claim that

- For a *deletion* or an *insertion without overflow*, $\Delta B_0 \geq -1$.
- For an *insertion with overflow*, $\Delta B_0 \geq 2\mu - 1$ and $\Delta B_1 = -1$.
- For a *tag adjusting operation of relaxed height $h \geq 1$ which creates a new node*, $\Delta B_h \geq 2\mu$ and $\Delta B_{h+1} \geq -1$.
- For a *density adjusting operation of relaxed height $h \geq 0$ which removes a node*, $\Delta B_h \geq \mu + 1$ and $\Delta B_{h+1} \geq -1$.
- For all cases not mentioned above, $\Delta B_h \geq 0$ for all $h$.

These claims are simply more detailed statements of the claims in theorem 2, and their verification follows the same lines. We will not repeat the arguments here.

Denote by $\tau_h$ the number of tag adjusting operations of relaxed height $h$ which create a new node, and by $\delta_h$ the number of density adjusting operations of relaxed height $h$ which remove a node. Also, denote by $i_1$ the number of insertions without overflow, and by $i_2$ the number of insertions with overflow. If $T_1$ is the initial tree and $T_2$ the final tree, the claims above imply

$$\begin{array}{rcl} B_0(T_2) & \geq & B_0(T_1) - i_1 - d + (2\mu - 1)i_2 + (\mu+1)\delta_0, \\ B_1(T_2) & \geq & B_1(T_1) - i_2 - \delta_0 + 2\mu\tau_1 + (\mu+1)\delta_1, \\ B_h(T_2) & \geq & B_h(T_1) - \tau_{h-1} - \delta_{h-1} + 2\mu\tau_h + (\mu+1)\delta_h, \end{array}$$

when $h \geq 2$. As $T_1$ is a standard $(a,b)$-tree, $B_h(T_1) \geq 0$ for all $h$. Also, $i = i_1 + i_2$. For notational convenience, we define $\tau_0 = i_2$ ($\tau_0$ would otherwise be undefined). Then the inequalities can be rewritten as

$$\begin{array}{rcl} B_0(T_2) + i + d & \geq & 2\mu\tau_0 + (\mu+1)\delta_0, \\ B_h(T_2) + \tau_{h-1} + \delta_{h-1} & \geq & 2\mu\tau_h + (\mu+1)\delta_h, \end{array}$$

for $h \geq 1$. By lemma 3, $\mu \geq 1$, so we have $2\mu \geq \mu + 1$. Thus,

$$\begin{array}{rcl} \tau_0 + \delta_0 & \leq & \dfrac{B_0(T_2) + i + d}{\mu + 1}, \\ \tau_h + \delta_h & \leq & \dfrac{B_h(T_2) + \tau_{h-1} + \delta_{h-1}}{\mu + 1}, \end{array}$$

for $h \geq 1$. Using the second inequality repeatedly, we obtain

$$\tau_h + \delta_h \leq \frac{i+d}{(\mu+1)^{h+1}} + \sum_{j=0}^{h} B_j(T_2)\frac{(\mu+1)^j}{(\mu+1)^{h+1}},$$

for $h \geq 0$. Without loss of generality, we may assume that $T_2$ is a standard $(a,b)$-tree. For such trees, all tags are 0, making the $B_h$'s used here equal to the ones used by Mehlhorn,[19] except that our indices are off by one compared to his definitions. Thus, with minor modifications Mehlhorn's bound on the $B_h$'s applies here, yielding $\sum_{j=0}^{h} B_j(T_2)(\mu+1)^{j+1} \leq (\mu+1)(n+i-d)$ for all $h \geq 0$. Hence,

$$\tau_h + \delta_h \leq \frac{i+d}{(\mu+1)^{h+1}} + \frac{(\mu+1)(n+i-d)}{(\mu+1)^{h+2}} = \frac{2i+n}{(\mu+1)^{h+1}}, \qquad (1)$$

for $h \geq 0$.

To bound the rest of the rebalancing operations, we do as follows. For an underfull node (recall that a negative node is, by definition, not called underfull), define its amount of underfullness as in the proof of theorem 2. For a relaxed $(a,b)$-tree $T$, let $C_h(T)$ denote the total amount of underfullness of nodes having relaxed height $h$, plus the total number of negative nodes having relaxed height $h$. It is easy to verify that

- For an *insertion with overflow*, $\Delta C_1 = 1$.
- For a *deletion*, $\Delta C_0 \leq 1$.
- For a *tag adjusting operation of relaxed height $h \geq 1$ which removes a node*, $\Delta C_h \leq -1$.
- For a *tag adjusting operation of relaxed height $h \geq 1$ which creates a new node*, $\Delta C_h = -1$ and $\Delta C_{h+1} = 1$.
- For a *density adjusting operation of relaxed height $h \geq 0$ which does not remove a node*, $\Delta C_h \leq -1$.
- For a *density adjusting operation of relaxed height $h \geq 0$ which removes a node*, $\Delta C_h \leq -1$, and $\Delta C_{h+1} \leq 1$.
- For a *root operation of relaxed height $h \geq 1$*, $\Delta C_h \leq -1$.
- For all cases not mentioned above, $\Delta C_h \leq 0$.

Denote by $\tau'_h$ the number of tag adjusting operations of relaxed height $h$ which remove a node, and by $\delta'_h$ the number of density adjusting operations of relaxed height $h$ which do not remove a node. Also, denote by $r_h$ the number of root operations of relaxed height $h$. The facts above imply

$$\begin{array}{rcl} C_0(T_2) & \leq & C_0(T_1) + d - \delta_0 - \delta'_0, \\ C_1(T_2) & \leq & C_1(T_1) + i_2 + \delta_0 - \tau_1 - \tau'_1 - \delta_1 - \delta'_1 - r_1, \\ C_h(T_2) & \leq & C_h(T_1) + \tau_{h-1} + \delta_{h-1} - \tau_h - \tau'_h - \delta_h - \delta'_h - r_h, \end{array}$$

when $h \geq 2$. Without loss of generality, we may assume that $T_2$ is a standard $(a, b)$-tree, and thus $C_h(T_1) = C_h(T_2) = 0$ for all $h$. Denote by $t_h$ the total number of rebalancing operations of relaxed height $h$. Since root operations and tag adjusting operations cannot have relaxed height 0, the above inequalities can (again defining $\tau_0 = i_2$) be rewritten as

$$\begin{array}{rcl} t_0 & \leq & d, \\ t_h & \leq & \tau_{h-1} + \delta_{h-1}, \end{array}$$

for $h \geq 1$. By inequality (1), the theorem follows. □

Note that the results in this section do not hold when $b = 2a - 1$ (Huddleston and Mehlhorn[16] give a simple counterexample which also applies here).

## 7. Conclusion

In this paper, we have generalized the proposal of Nurmi, Soisalon-Soininen, and Wood[7] for relaxed B-trees to relaxed $(a, b)$-trees, and we have introduced a more carefully designed set of rebalancing operations. With this new set of rebalancing operations, the uncoupling of the rebalancing from the updating, allowing a high degree of parallelism, is obtained at practically no cost, since we have proved that the complexity of rebalancing relaxed $(a, b)$-trees is essentially the same as the complexity of rebalancing ordinary $(a, b)$-trees.

Since B-trees, 2-3 trees, and 2-3-4 trees are special cases of $(a, b)$-trees, we also have relaxed versions for these, including the logarithmic rebalancing results. Notice that, as in the sequential case, the amortized results from section 6 only hold when $b \geq 2a$, so only 2-3-4 trees and the variant of B-trees with $b = 2a$ have these properties.

Though we have not focused on storage utilization, one remark seems appropriate here. It is clear that underfull nodes waste space, and if many deletions but no rebalancing operations are done for an extended period of time, the amount of wasted space can be arbitrarily large. However, if the necessary rebalancing operations are carried out in parallel with the updates, the problem should not be significantly larger than for ordinary B-trees or $(a, b)$-trees, as the tree should always be close to balanced (a precise quantification is difficult, since it will depend on the actual order in which the rebalancing processes attend to the problems of unbalance, as well as on the updates performed). Note that by our amortization results, "necessary" means $O(1)$ per update on average (when $b \geq 2a$).

One important observation (made in the proof of theorem 2) is that nodes with tag value $-1$ can never have more than two children, so while the tree is used, such nodes could be kept in main memory (in records with room for only two values) so that space is not wasted.

## References

1. R. Bayer and E. McCreight, "Organization and maintenance of large ordered indexes", *Acta Inform.* **1** (1972) 173–189.
2. B. Samadi, "B-trees in a system with multiple users", *Inform. Process. Lett.* **5 (4)** (1976) 107–112.
3. E. W. Dijkstra, "Co-operating sequential processes", in *Programming Languages*, ed. F. Genuys (Academic Press, 1968) 43–112.
4. Y.-S. Kwong and D. Wood, "A new method for concurrency in *B*-trees", *IEEE Trans. Software Eng.* **8 (3)** (1982) 211–222.
5. P. L. Lehman and S. B. Yao, "Efficient locking for concurrent operations on B-trees", *ACM Trans. Database Systems* **6 (4)** (1981) 650–670.
6. Y. Sagiv, "Concurrent operations on $B^*$-trees with overtaking", *J. Comp. System Sci.* **33** (1986) 275–296.
7. O. Nurmi, E. Soisalon-Soininen and D. Wood, "Concurrency control in database structures with relaxed balance", *ACM Proc. of the sixth ACM Symposium on Principles of Database Systems* (1987) 170–176.
8. L. J. Guibas and R. Sedgewick, "A dichromatic framework for balanced trees", *19th IEEE Foundations of Computer Science* (1978) 8–21.
9. G. M. Adel'son-Vel'skiĭ and E. M. Landis, "An algorithm for the organisation of information", *Dokl. Akad. Nauk SSSR* **146** (1962) 263–266. (In Russian. English translation in *Soviet Math. Dokl.* **3** (1962) 1259–1263.)
10. J. L. W. Kessels, "On-the-fly optimization of data structures", *Comm. ACM* **26** (1983) 895–901.

11. O. Nurmi and E. Soisalon-Soininen, “Uncoupling updating and rebalancing in chromatic binary search trees”, *Proc. of the tenth ACM Symposium on Principles of Database Systems* (1991) 192–198.
12. J. F. Boyar and K. S. Larsen, “Efficient Rebalancing of Chromatic Search Trees”, *J. Comp. System Sci.*, 49 (3), 667-682, 1994. Also in LNCS 621, 151–164, Springer-Verlag, 1992.
13. J. Boyar, R. Fagerberg and K. S. Larsen, “Amortization Results for Chromatic Search Trees, with an Application to Priority Queues”, *Proc. of the 4th Intl. Workshop, WADS '95* (1995) 270–281.
14. L. Malmi, “An efficient algorithm for balancing binary search trees”, TKO-B84, Dept. of Comp. Sci., Helsinki University of Technology”, 1992.
15. K. S. Larsen, “AVL trees with relaxed balance”, *Proc. 8th Intl. Parallel Processing Symposium* (IEEE Computer Society Press, 1994) 888–893.
16. S. Huddleston and K. Mehlhorn, “A new data structure for representing sorted lists”, *Acta Inform.* **17** (1982) 157–184.
17. J. D. Ullman, “Principles of Database and Knowledge-Base Systems, Vol. 1”, (Computer Science Press, 1988).
18. N. Goodman and D. Shasha, “Semantically-based concurrency control for search structures”, *Proc. of the fourth ACM Symposium on Principles of Database Systems* (1985) 8–19.
19. K. Mehlhorn, “Data Structures and Algorithms, Vol. 1: Sorting and Searching”, (Springer-Verlag, 1984).