Relaxed Multi-Way Trees with Group Updates[1]

Kim S. Larsen[2]

Department of Mathematics and Computer Science, University of Southern Denmark, Odense, Denmark

E-mail: kslarsen@imada.sdu.dk

Data structures with relaxed balance differ from standard structures in that rebalancing can be delayed and interspersed with updates. This gives extra flexibility in both sequential and parallel applications.

We study the version of multi-way trees called (a, b)-trees (which includes B-trees) with the operations insertion, deletion, and group insertion. The latter has applications in for instance document databases, WWW search engines, and differential indexing. We prove that we obtain the optimal asymptotic rebalancing complexities of amortized constant time for insertion and deletion and amortized logarithmic time in the size of the group for group insertion. These results hold even for the relaxed version.

This is an improvement over the existing results in the most interesting cases.

Key Words: search trees; multi-way trees; B-trees; relaxed balance; complexity; amortized analysis; group update; group insertion

1. INTRODUCTION

We focus on the type of multi-way trees usually referred to as (a, b)-trees [13, 23], and in particular, we adopt the *relaxed* (a, b)-trees [19]. In the context of search trees, "relaxed" is the term used when a structure is generalized in such a way that updating may be carried out independent of rebalancing which can be carried out later, possibly in small steps. In the context of B-trees [3], this approach was discussed first in [27], followed by complexity results in [19], and a study of variations with other properties in [14].

[1] A preliminary version of this paper appeared in the Proceedings of the Twentieth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, ACM Press, 2001, pp. 93–101.

[2] Supported in part by the Danish Natural Sciences Research Council (SNF) and in part by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

The paper [17] contains a fairly complete reference list to the work on relaxed structures in general. In brief, a relaxed version of AVL-trees [1] was introduced in [27, 28] with complexity results matching the standard results [24] in [18]. A relaxed version of red-black trees [9] (see also [2]) was introduced in [26] with complexity results gradually matching the standard results [30] in [6, 5, 17]. Another variant is described in [11]. Finally, a general result for creating relaxed structures can be found in [20], and performance results from experiments with relaxed structures can be found in [4, 10].

The disadvantage of relaxation is that the strict control on search path lengths is loosened (temporarily). The advantage of relaxed structures is flexibility. Since rebalancing can be delayed and carried out in small steps interspersed with updates, they give extra possibilities for control, both with regards to trade-off between time spent on updating and time spent on rebalancing in a single processor scenario, but also with regards to concurrency control. Note also that a relaxed structure can always be used as a standard structure simply by deciding to carry out all rebalancing operations due to an update immediately. Thus, an asymptotic complexity result carries over from the relaxed to the standard case.

In this paper, we consider *group* update operations, where a number of keys must or may be inserted or deleted at the same time. These operations, in particular group insertion, have renewed interest because of applications in WWW search engines or document databases using inverted index techniques [7, 8], in differential indexing when a very large disk-based index is supplemented by a main memory so-called differential index [16], in on-line index construction [25], or in other applications where a large number of keys must or can be brought into the main index at the same time, while allowing concurrent use of the system. Structures with relaxed balance are well suited for concurrent applications of this nature because newly inserted elements are available immediately after the actual update. Rebalancing can be done later, possibly by a background process, for instance when the search frequency drops.

For red-black trees [9] and height-valued trees [21] (an AVL-tree variant), relaxed variants have been studied in [12, 22] where an entire tree of new keys to be inserted can be brought into the tree as one update. For both structures, the upper bound derived on the number of rebalancing operations required to balance the tree again is $O(\log n + \log^2 m)$, where n is the size of the main index and m is the size of the tree which is inserted.

Group updates in B-trees have been considered in [29] based on [27]. The focus in [29] is on searching and the necessary concurrency control. There is no new bound on the number of operations, so the best bound one can give on the basis of [27] is $O(m \log_a n)$, where a is the degree of the nodes. However, if many updates go to the same leaves, the performance will be correspondingly better. As a remark regarding notation, since the base for the logarithm can be quite large for multi-way trees, we leave the constant in when stating the asymptotic performance, even though formally this does not signify anything since $O(\log_2 n) = O(\log_c n)$ for any constant c.

In this paper, we define a relaxed multi-way structure where the number of rebalancing operations carried out in response to the insertion of a tree of size m is the optimal $O_A(\log_a m)$ and insertion and deletion become $O_A(1)$. Note that we

use the notation $O_A(f(n))$ to mean *amortized* $O(f(n))$. These results also imply a relaxed binary structure with the same complexities (the logarithm now base 2).

We now remark on the definition of what a group insertion algorithm is (the discussion for group deletion is similar). We study the core problem of moving m keys in between two neighbor keys in the main index. However, if one considers the problem of moving m arbitrary keys in, they first have to be divided up into groups (via a search procedure). This can be done for our structure exactly as it has been done in [12, 22, 29]. The difference between our approach and the earlier ones lie in the rebalancing after the insertion of a whole tree, which is the focus in the main part of our paper.

For group insertion of m arbitrary keys into the (non-relaxed) structures considered in [12, 22], a bound of $O(\log n + \sum_{i=1}^{p} \log^2 m_i)$ is stated, assuming that there are p locations where trees of sizes $m_1, \ldots, m_p$ are inserted. Our corresponding result for the same operation would be $O_A(\sum_{i=1}^{p} \log_a m_i)$.

2. RELAXED (A,B)-TREES

Partly for comparison and partly because a standard (a, b)-tree will be the ideal state for a relaxed (a, b)-tree, we give the definition here. Terminology which carries over to the relaxed case will not be repeated.

We consider leaf-oriented (a, b)-trees which means 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. The purpose of the routers is to guide the searches to the correct leaves. The term leaf-oriented, which is usually used when discussing relaxed data structures, corresponds to B^+-trees [3] versus internal B-trees.

Leaf-oriented trees are often the choice in large database-oriented applications. Thus, we assume that the leaves contain the keys and references to the actual data associated with the keys. For uniformity, these references are referred to as children just like the references from internal nodes.

2.1. Standard (a,b)-Trees

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

- 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.

Additionally, an (a, b)-tree must fulfill the following search tree invariant: Each internal node u with m children (pointers to subtrees) 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 all keys in the range $[k_i, k_{i+1})$, $0 \leq i \leq m - 1$, in the subtree of a node u are stored in the ith subtree of u.

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

2.2. Relaxed (a,b)-Trees

First we relax the invariants from the standard case such that updates can legally be performed without immediate subsequent rebalancing. Removing all requirements would of course accomplish this. However, as usual we are interested in being

able to rebalance efficiently at some later time, which means that some non-trivial invariant must be maintained. In order to express the new structural invariant, we introduce the following: Every node has a *tag*, which is a non-positive integer. This is also commonly referred to as the *weight* of the node. We define the *relaxed depth*, $rd(u)$, of a node u as follows:

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

where $t(u)$ denotes the tag of u, and $p(u)$ denotes the parent of u.

Only the structural invariant is altered:

- All nodes have at most b children.
- All leaves have the same relaxed depth.

Of course, by definition, an internal node must have at least one child.

Now, if a node in a relaxed (a, b)-tree has a property different from what it could have in a standard (a, b)-tree, we refer to this as a *conflict*: A tag value different from zero is referred to as a *weight* conflict. If the tag is zero, but the node as fewer than a children, this is an *underfull* conflict, and the node is called *underfull*. In the special case of the root, that node is underfull (and there is an underfull conflict) only if it has fewer than 2 children, i.e., one child.

To finish the degree terminology regarding nodes, a node with degree zero is called *empty* and a node with degree b is call *full*.

The intuition regarding tag values is that they measure the distance from where a node is located in the tree compared with where it ought to be. More concretely, if a node u has tag value $t < 0$, then all the descendants of u, the leaves in particular, are $|t|$ levels too far from the root compared with nodes which are not in the subtree of u. Thus, u should be moved $|t|$ levels closer to the root to fix the problem.

We proceed to the description of the operations on relaxed (a, b)-trees. In the following sections, we define the notation used in the illustrations of the operations, and we give additional explanation of any conditions which cannot be (or is not) given in these illustrations. Part of the purpose of the illustrations is to have these as easy visual reference in the proofs to follow, so we have made an attempt not to clutter them with obvious information which can be given once (in the sections below).

In general, the top-most node before an operation is carried out is physically the same node as the top-most node after the operation is carried out, such that the reference in its parent remains valid.

When we say that a number i of pointers are divided up as evenly as possible (into two nodes), we mean that one node receives $\lfloor i/2 \rfloor$ pointers and the other $\lceil i/2 \rceil$. When i is odd, it is not important for the results in this paper which node receives the most.

2.2.1. Update Operations

Any update operation is preceded by a search for the correct location. The searching is facilitated by the search tree invariant and proceeds exactly as in all multi-way search trees. In the discussion of insertion and deletion below, we assume that we have already located the correct leaf.

In the illustrations, Greek letters are used to denote a (possibly empty) collection of pointers. If α is such a collection of pointers, we let $|\alpha|$ denote the number of pointers in the collection α. We do not explicitly show the keys or routers. The tags of the nodes are shown as superscripts to the right of the nodes. Single pointers are denoted by an x in the leaves and by a dot in the internal nodes.

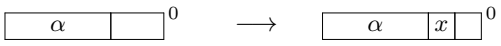

FIG. 1. Insertion of x: $|\alpha| < b$.

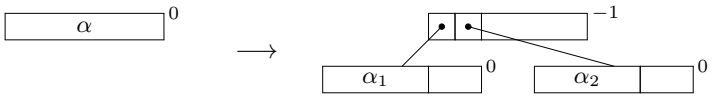

FIG. 2. Insertion of x: $|\alpha| = b$, $\alpha x = \alpha_1\alpha_2$.

Insertion: There are two possibilities. Either the correct leaf for the insertion is full or it is not. If the leaf is not full, the new key is just added to that leaf. Even though this is not apparent in the illustration, we assume that keys are kept in sorted order. If the leaf is full, all the existing keys together with the one to be inserted are divided as equally as possible into two groups α_1 and α_2. All the keys in α_1 are smaller than any key in α_2.

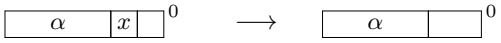

FIG. 3. Deletion of x.

Deletion: If present, the correct key (which could be in any location in the leaf) is found and removed.

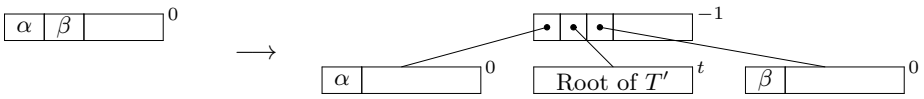

FIG. 4. Insertion of T': $t = -h(T')$.

Group Insertion: The node marked "Root of T'," is the root of an entire standard (a, b)-tree T'. It is a requirement that all the keys in T' lie between the largest key in α and the smallest key in β. The tree T' is build by the algorithm from the set of keys which is to be inserted at that location. Given a number of keys m, an (a, b)-tree containing these m keys can be build in many different ways. We use this freedom to build the tree with as few extreme nodes as possible, where an extreme node is a node with exactly a or exactly b children. This problem is investigated in great detail in [15]. Here, the following will suffice. Choose an integer d such that

$a < d < b$. This can always be done when $a \geq 2$ and $b \geq 2a$. Now create leaves with d keys in each until at most $a + b$ keys have not been placed in a leaf. There are then at least $a + b + 1 - d \geq a + 2$ and at most $a + b$ keys left which can be placed in one or two leaves containing at least a and at most b keys each. We now have a collection of nodes containing all keys. Recursively build layers of internal nodes on top of each other until one tree is formed. Each layer is constructed in the same manner such that all, except at most two nodes at each layer, have degree d. Since the tree will have height $O(\log_a m)$, the number of extreme nodes will also be $O(\log_a m)$.

2.2.2. Rebalancing Operations

Very informally, the collection of rebalancing operations is chosen in a problem-oriented manner as the smallest collection having the property that all problem types can be treated and eventually removed. We introduce each operation separately below.

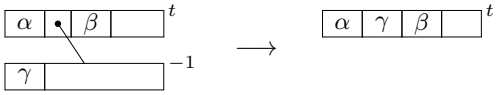

FIG. 5. Absorption: $|\alpha| + |\beta| + |\gamma| \leq b$.

Absorption: In order to prevent immediate propagation of pointers all the way to the root when nodes become overfull, a local extra layer, marked with a -1, may be introduced. When the pointers in such a node can fit in the parent node, possibly at a later time, the problem can be resolved by an absorption.

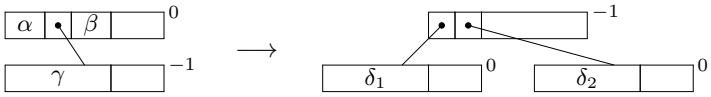

FIG. 6. Split: $|\alpha| + |\beta| + |\gamma| > b$, $\alpha\gamma\beta = \delta_1\delta_2$.

Split: When the pointers in such an extra local node cannot fit in the parent, a split is performed, propagating the problem node towards the root. The pointers in $\alpha\gamma\beta$ are divided as evenly as possible into two groups δ_1 and δ_2.

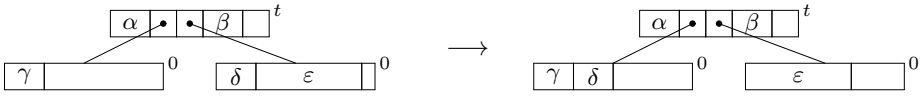

FIG. 7. Sharing: $|\gamma| < a$, $|\gamma| + |\delta| + |\varepsilon| \geq 2a$ (symmetric in children).

Sharing: When a node has fewer than a children, it may be possible to solve the problem by sharing with a sibling, provided that the total number of children among the two nodes is at least $2a$. There are two symmetric variants of this

operation: either the left or the right sibling is underfull. The pointers in the two nodes are divided up as evenly as possible.

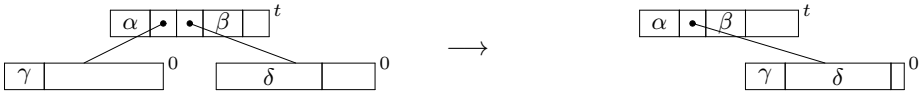

FIG. 8. Fusion: $|\gamma| < a$, $|\gamma| + |\delta| < 2a$ (symmetric in children).

Fusion: If a node has fewer than a children and neither of its at most two siblings have enough to share, then the pointers are moved to a sibling node, and the empty node and the reference to it in its parent are removed. There are two symmetric variants of this operation: either the left or the right sibling is underfull.

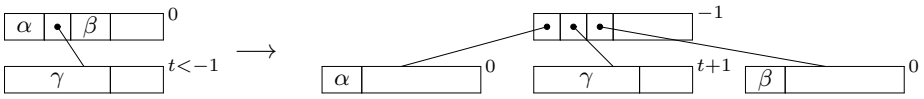

FIG. 9. Penetration.

Penetration: A node with tag value smaller than -1 is the root of a tree which has been inserted by a group insertion. In order to reestablish the standard (a, b)-tree invariants, the node should be moved closer to the root of the tree until its tag value becomes zero. This is done one layer at a time, possible creating other problems on the way.

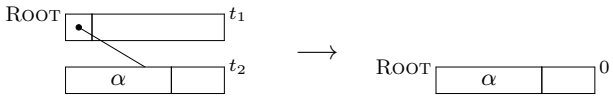

FIG. 10. Redundant Root Elimination.

Redundant Root Elimination: If the root of the tree has only one child, the root is redundant and can be removed just as in a standard (a, b)-tree. Since layers are measured relative to the root of the tree, the root cannot be placed at an incorrect layer. Thus, non-zero tag values at the root can simply be removed.

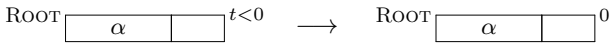

FIG. 11. Root Weight Elimination: $|\alpha| \geq 2$.

Root Weight Elimination: This operation simply removes non-zero tag value at the root in the case where redundant root elimination cannot be applied.

3. CORRECTNESS

There are some important points regarding correctness:

• If one of the operations is applied to a relaxed (a, b)-tree, then the resulting tree is again a relaxed (a, b)-tree.

• Leaves always have tag values zero.

Though it would be quite space consuming to go through every operation regarding these properties, they are very simple to check because they can be verified separately for each operation. We omit these details.

Of course, the first property must hold if the set-up should be meaningful at all. The second property, along with the definition of the update operations, ensures that updates can always be made.

The next important aspect is as to whether the collection of operations suffice to rebalance the tree. The following theorem establishes that the collection is complete.

THEOREM 3.1. *If there is a conflict in a relaxed (a, b)-tree, then one of the rebalancing operations can be applied.*

Proof. Assume first that the root has a conflict. Since there are no restrictions on the root operations, either *Redundant Root Elimination* or *Root Weight Elimination* can be applied.

Now assume that the root does not have a conflict. Let u be a top-most node with a conflict, i.e., a node closest to the root. We note that it has a parent, and that the parent does not have a conflict. Thus, we can assume that the parent has tag value zero.

If there is a top-most conflict which is a weight conflict, we choose such a conflict to deal with next. *Absorption* and *Split* cover all cases when the tag value is -1, and *Penetration* can be applied if the tag value is smaller.

We may now assume that the top-most conflict is an underfull node and that none of its siblings has a weight conflict. Thus, the conflict node and its siblings have tag values zero. Clearly then, *Sharing* and *Fusion* cover all cases. ■

Of course, this is merely one aspect of the question as to whether the collection of operations suffice to rebalance the tree.

Theorem 3.1 leaves the question unanswered as to whether the rebalancing process will ever terminate (if the updating terminates). This question is answered affirmatively in the next section.

4. COMPLEXITY

As already mentioned, searching and the actual updating is carried out as in [12, 22], for instance; with regards to searching, also as in [29]. We focus on the subsequent rebalancing. We derive the amortized rebalancing complexity of the update operations using the potential function technique [31].

Our aim is to show that rebalancing due to a single update (insertion or deletion) is $O_A(1)$ and that rebalancing due to an insertion of a tree of size m is $O_A(\log m)$. By definition, this means that if we start with an empty structure and carry out k single updates and p group insertions of trees of sizes $m_1, \ldots, m_p$, then the total worst-case rebalancing complexity is $O(k + \sum_{i=1}^{p} \log m_i)$.

We define a so-called potential function Φ from our structures into the integers, and show that a simple update will increase Φ with at most $O(1)$, that a group insertion of a tree of size m will increase Φ with at most $O(\log m)$, and that any rebalancing operation will decrease Φ. More formally, increasing Φ with at most $O(f)$ means that if an update changes a structure T to T', then $\max(\Phi(T') - \Phi(T), 1) \in O(f)$, whereas decreasing Φ means that $\Phi(T') < \Phi(T)$. Since Φ will be defined such that it is always non-negative and initially zero, this will give the desired bound on the rebalancing.

First we define the potential $\Phi(u)$ of a node u. The potential of a tree is then merely the sum of the potentials of all the nodes in the tree. We use the notation $c(u)$ to denote the number of children of u.

$$\Phi(u) = \begin{cases} 3, & t(u) = 0,\ c(u) < a \\ 1, & t(u) = 0,\ c(u) = a \\ 2, & t(u) = 0,\ c(u) = b \\ 3, & t(u) = -1,\ c(u) \leq 2 \\ 5, & t(u) = -1,\ c(u) > 2 \\ 12(|t(u)| - 1) + 5, & t(u) < -1 \\ 0, & \text{otherwise} \end{cases}$$

The "otherwise" case covers nodes u where $t(u) = 0$ and $a < c(u) < b$.

If u is the root, $\Phi(u)$ is defined similarly, except that we substitute the constant 2 for a.

There is no guaranteed recipe for defining a potential function. It is an integrated part of the proof and strongly linked to the exact definition of updating and rebalancing operations. However, there are guidelines and standard techniques which often work, at least partially.

Rebalancing after one deletion, for instance, may take time $\Omega(\log_a n)$, if all nodes on the search path to the update location, and their siblings, have degree a. Thus, in order to show that the update gives rise to only $O_A(1)$ rebalancing, there must be $\Omega(\log_a m)$ potential in the tree which is released as a consequence of the rebalancing operations. This is the motivation for giving nodes u with $t(u) = 0$ and $c(u) = a$ a non-zero potential.

If an extra pointer must be inserted into a node u with $c(u) = b$, then the node must be split, and if $b = 2a$, this gives rise to one node with $c(u) = a$. This is the reason why the potential of a node with $c(u) = b$ must be larger than the potential of a node with $c(u) = a$.

The remaining cases are for real problems of imbalance. When these are resolved, some of them may create nodes with $c(u) = a$ or $c(u) = b$ or even other real problems of imbalance, and their larger potentials reflect this. Thus, to some extent, the potential of the nodes correspond to the relative difficulty of solving the different problems of imbalance.

It is well-known that even though (a, b)-trees can be defined if just $b \geq 2a - 1$, the best complexities are only obtained if $b \geq 2a$ [13]. Naturally, since we might just use plain insertion and deletion, this property carries over.

THEOREM 4.1. *If $b \geq 2a$, then starting from an empty (a,b)-tree, the number of rebalancing operations is $O_A(1)$ in response to an insertion or deletion and $O_A(\log_a m)$ in response to a group insertion of another (a,b)-tree of size $m \geq 2$.*

Proof. In this proof, we make many references to the illustrations in the appendix. Some terminology shortens the proof significantly: We use P for parent to refer to the top node of an operation. Similarly, for children, we use L, C, and R for left, right, and center, respectively. A subscript of "1" refers to a node before the operation is carried out and a subscript of "2" refers to a node after the operation is carried out.

Below, we prove that every rebalancing operation decreases the potential. Thus, the number of rebalancing operations which can be carried out is bounded by the increase in potential due to the update operations.

Insertion and *Deletion* alter a constant number of nodes and do not introduce tag values smaller than -1. Thus, by definition of the potential function, these operations increase the potential by at most a constant.

Group Insertion adds an unbounded number of nodes m to the tree. However, since the tree T', which is added, is a standard (a,b)-tree constructed such that there are at most $O(\log_a m)$ extreme nodes, the potential for all other nodes in T' is zero. In addition, there is a potential increase due to the nodes directly involved in the operation, i.e., the nodes shown in Fig. 4. The node C_2 gives rise to a potential increase proportional to the height of the tree T', which is $O(\log_a m)$, whereas the other nodes only give rise to a constant potential increase. Thus, the total is $O(\log_a m)$.

The operations and the potential function were designed to accomplish what we prove now, namely that any rebalancing operation decreases the potential:

Absorption: By removing the -1 node, the potential drops at least 3. Now, assume first that $t = 0$. If P_2 is underfull, then P_1 was also underfull, since a -1 node has degree at least one. Thus, the maximal increase will occur if $|\alpha| + |\beta| + |\gamma| = b$. This increase is 2, so there is a total drop of at least 1. If instead $t = -1$, then the potential for P_2 can increase compared with P_1 if the degree of P_1 was at most 2. Again, this increase is at most 2. If $t < -1$, then the potential of P_2 equals that of P_1. Thus, in all cases, there is a total decrease in potential of at least 1.

Split: C_1 must have degree at least 2; otherwise we cannot have $|\alpha|+|\beta|+|\gamma| > b$. Assume that the degree of C_1 is 2. Then the degree of P_1 is b. Thus, the potential before the operation is $3 + 2 = 5$. Since $|\alpha| + |\beta| + |\gamma| > b \geq 2a$, at most one of L_2 and R_2 can have degree as small as a. Thus, the potential after the operation is at most $3 + 1 = 4$. Now assume that the degree of C_1 is at least 3. Thus, its potential is 5. The same argument as before applies to the situation after the operation, so we get a total drop in potential of at least 1.

Sharing: This operation has two symmetric variants, which can be treated simultaneously. One of L_1 and R_1 is underfull and has a potential of 3. Afterwards, L_2 and R_2 both have degree at least a and less than b, so the potential of each node is at most 1. Thus, the potential decreases with at least 1.

Fusion: This operation has two symmetric variants, which can be treated simultaneously. The potential of P_1 can increase with at most 2. This happens in the case where $t = 0$ and the degree of P_1 is a. Thus, we must show that the potential at the level below decreases by at least 3. Now, an underfull node is removed which decreases the potential by 3. The remaining child of P_2 has possibly had its degree increased. However, since its degree is less than $2a \leq b$, the potential of that node cannot increase.

Penetration: The potential of C_2 is 12 smaller than C_1; also if $t + 1 = -1$. The potential of P_2 is 5 and the potential of L_2 and R_2 is at most 3 for each of them. This is a total of 11. Thus, the potential decreases.

Redundant Root Elimination: Clearly, the potential of C_1 cannot increase, and since P_1 is underfull, removing this nodes decreases the potential.

Root Weight Elimination: The number of pointers in the node is at least two. By inspecting the potential function (recalling that for the root, a is replaced by the value 2 in the definition of the potential), it follows that with at least 2 pointers, a node with a negative tag always has a strictly larger potential than a node with tag zero. Thus, the potential decreases. ■

This result is asymptotically optimal. Clearly, no operation can take time less than a constant, so the asymptotic complexity of *insertion* and *deletion* cannot be improved.

Regarding the complexity of *group insertion*, choose a path which from every node follows a pointer located roughly in the middle of the node. If we insert a tree of height h at the resulting leaf, at least $h - 1$ nodes in the original tree must be split. This imposes a lower bound of $\Omega(\log_a m)$ on the operation.

5. ADDITIONAL OPERATIONS

Note that in Theorem 4.1, we could have taken *insertion* to mean "any modification of a leaf such that the number of elements increase while the search tree invariant is preserved." Clearly, as it appears from the proof of that theorem, this more general operation would also be $O_A(1)$. Similarly, *deletion* could be taken to mean "any modification of a leaf such that the number of elements decrease while the search tree invariant is preserved," and this operation would also be $O_A(1)$.

A standard operation in dictionary implementations is the *join* operations (also sometimes referred to as *merge*, *meld*, or *union*) which takes two dictionaries as arguments and combines them into one. It is always assumed that all keys in one of the dictionaries are smaller than all keys in the other. Assume that the dictionaries have sizes n_1 and n_2. Then by performing a *group insertion* of the smaller into the larger, in the left-most or right-most leaf, as appropriate to preserve the search tree invariant, we obtain a relaxed *join* operation with complexity $O_A(\log \min\{n_1, n_2\})$.

Another standard operation is the *split* operation (on dictionaries; not on (a, b)-tree nodes) which given a key value produces two dictionaries; one with all the keys smaller than or equal to the given key and another with the rest. By first searching in T for the given key value and using *group insertion* to insert a "fake" root with a small enough tag value t ($|t|$ should be larger than the height of T), then this fake root will eventually make its way up to become a child of the root at which point the left-most and right-most pointers in the root can be used to form the desired

trees. A special mark must be put on the fake root and the operations modified such that no other operation than *split* can be applied to such a marked child of the root.

For *group deletion*, if all deletions regarding any particular leaf are carried out at the same time, the time to rebalance after the deletion of m elements located in $p \leq m$ leaves become $O_A(p)$ instead of $O_A(m)$.

The proofs of the sections above is the place to start if one considers altering the collection of rebalancing operations. After a radical change, it is of course necessary to verify all properties again. However, if operations are only generalized, the collection is of course still sufficient. One example of a possible generalization is the following: In the *fusion* operations, allow L_1, R_1, and C_2 to have non-zero, but identical, tag values. With this generalization, a slightly different potential function can be used to obtain the same asymptotic results. However, it is important to note that a generalization is not necessarily an improvement and can lead to worse performance. In fact, in the worst scenario, a generalization could lead to infinite loops. For the safe generalizations, where the same asymptotic results can obtained, experiments can be used to decide on the exact collection.

6. CONCLUDING REMARKS

Using colors or techniques as in [2], $(2, 4)$-trees can be represented as binary trees, where small parts of the tree of height zero or one represent nodes of degree up to four. By interpreting all the rebalancing operations in that representation, we immediately obtain that a relaxed binary tree with all the same properties exists, i.e., rebalancing after *insertion* and *deletion* is $O_A(1)$ and rebalancing after *group insertion* is $O_A(\log m)$, where m is the size of the tree which is inserted.

This improves on previous results for binary search trees which were obtained for AVL-trees in [22] and red-black trees in [12], both claiming a rebalancing complexity in this case of $O(\log n + \log^2 m)$, where n is the size of the tree in which the update is carried out. Of course, the result from [12, 22] is a worst-case result, whereas ours is amortized time. This means that if one considers any one operation in isolation, we cannot claim a good worst-case bound. We can of course claim $O(n)$, since no more than $O(n)$ potential can be accumulated in the tree. However, in practice, it is the complexity of carrying out (long) sequences of operations which is interesting.

Additionally, our amortization proof is given in the traditional manner, assuming that the starting state is an empty tree. However, it is of course also interesting to discuss which results hold when starting with an initially nonempty structure. One interesting point is that as soon as $\Omega(n)$ operations have been carried out, then these operations can "pay" for the potential which should be present in the tree. Thus, after that point, all the asymptotic amortized results are valid. In principle, this holds whenever $\Omega(n)$ operations have been performed, independent of what the actual constant is, so it could be $\frac{1}{1000}n$ operations, for example. In practice, this constant should probably be closer to one before reasonable run-time constants can be guaranteed.

This discussion assumes that the initial nonempty tree has been built completely independent from the intended use. By inspection of the potential function used in this paper, one observes that if all nodes have a number of children strictly between the extreme values of a and b, then the potential of the whole tree is zero. Thus, all

the amortized results hold immediately. In fact, it is also unproblematic to allow for instance a constant number of extreme nodes. Trees with such properties can be constructed for all except very small values of a and b [15].

When choosing the values for a and b, b is naturally defined to match the block size. Choosing a subsequently such that $2a = b$ is natural, but any smaller a (larger than 2) is allowed and will result in a structure with the properties established in this paper. Clearly, choosing a large results in the best space utilization guarantee and the best maximum height guarantee, whereas choosing a smaller will decrease the number of rebalancing operations which must be carried out. Thus, small a's should be considered only in fairly dynamic environments.

ACKNOWLEDGMENT

The author would like to thank the anonymous referees for suggesting several improvements.

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*, 1:173–189, 1972.
4. L. Bougé, J. Gabarró, X. Messeguer, and N. Schabanel. Concurrent Rebalancing of AVL Trees: A Fine-Grained Approach. In *Proceedings of the Third Annual European Conference on Parallel Processing*, volume 1300 of *Lecture Notes in Computer Science*, pages 421–429. Springer-Verlag, 1997.
5. J. Boyar, R. Fagerberg, and K. 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. J. F. Boyar and K. S. Larsen. Efficient Rebalancing of Chromatic Search Trees. *Journal of Computer and System Sciences*, 49(3):667–682, 1994.
7. A. F. Cardenas. Analysis and Performance of Inverted Data Base Structures. *Communications of the ACM*, 18(5):253–263, 1975.
8. C. Faloutsos and H. V. Jagadish. Hybrid Index Organizations for Text Databases. In *Third International Conference on Extending Database Technology*, volume 580 of *Lecture Notes in Computer Science*, pages 310–327, 1992.
9. L. J. Guibas and R. 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. S. 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, T. 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. Hanke and E. Soisalon-Soininen. Group Updates for Red-Black Trees. In *Proceedings of the 4th Italian Conference on Algorithms and Complexity*, volume 1767 of *Lecture Notes in Computer Science*, pages 253–262. Springer-Verlag, 2000.
13. S. Huddleston and K. Mehlhorn. A New Data Structure for Representing Sorted Lists. *Acta Informatica*, 17:157–184, 1982.
14. L. Jacobsen and K. S. Larsen. Variants of (a, b)-Trees with Relaxed Balance. *International Journal of Foundations of Computer Science*, 12(4):455–478, 2001.

15. L. Jacobsen, K. S. Larsen, and M. N. Nielsen. On the Existence and Construction of Non-Extreme (a,b)-Trees. Tech. report 11, Department of Mathematics and Computer Science, University of Southern Denmark, Odense, 2001. Submitted to *Information Processing Letters*.

16. S.-D. Lang, J. R. Driscoll, and J. H. Jou. Batch Insertion for Tree Structured File Organizations—Improving Differential Database Representation. *Information Systems*, 11(2):167–175, 1986.

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

18. K. S. Larsen. AVL Trees with Relaxed Balance. *Journal of Computer and System Sciences*, 61(3):508–522, 2000.

19. K. S. Larsen and R. Fagerberg. Efficient Rebalancing of B-Trees with Relaxed Balance. *International Journal of Foundations of Computer Science*, 7(2):169–186, 1996.

20. K. S. Larsen, T. Ottmann, and E. Soisalon-Soininen. Relaxed Balance for Search Trees with Local Rebalancing. *Acta Informatica*, 37(10):743–763, 2001.

21. K. S. Larsen, E. Soisalon-Soininen, and P. Widmayer. Relaxed Balance through Standard Rotations. *Algorithmica*, 31(4):501–512, 2001.

22. L. Malmi and E. Soisalon-Soininen. Group Updates for Relaxed Height-Balanced Trees. In *Proceedings of the Eighteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 358–367. ACM Press, 1999.

23. K. Mehlhorn. *Sorting and Searching*, volume 1 of *Data Structures and Algorithms*. Springer-Verlag, 1986.

24. K. Mehlhorn and A. Tsakalidis. An Amortized Analysis of Insertions into AVL-Trees. *SIAM Journal on Computing*, 15(1):22–33, 1986.

25. C. Mohan and I. Narang. Algorithms for Creating Indexes for Very Large Tables Without Quiescing Updates. In *Proceedings of the Eleventh ACM SIGMOD International Conference on Management of Data*, pages 361–370. ACM Press, 1992.

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

27. 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.

28. O. Nurmi, E. Soisalon-Soininen, and D. Wood. Relaxed AVL Trees, Main-Memory Databases and Concurrency. *International Journal of Computer Mathematics*, 62:23–44, 1996.

29. K. Pollari-Malmi, E. Soisalon-Soininen, and T. Ylönen. Concurrency Control in B-Trees with Batch Updates. *IEEE Transactions on Knowledge and Data Engineering*, 8(6):975–984, 1996.

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

31. R. E. Tarjan. Amortized Computational Complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985.