Complexity of Layered Binary Search Trees with Relaxed Balance

Lars Jacobsen Kim S. Larsen *

University of Southern Denmark, Odense **

Abstract. When search trees are made relaxed, balance constraints are weakened such that updates can be made without immediate rebalancing. This can lead to a speed-up in some circumstances. However, the weakened balance constraints also make it more challenging to prove complexity results for relaxed structures.
In our opinion, one of the simplest and most intuitive presentations of balanced search trees has been given via layered trees. We show that relaxed layered trees are among the best of the relaxed structures. More precisely, rebalancing is worst-case logarithmic and amortized constant per update, and restructuring is worst-case constant per update.

Introduction

Usually, updating in a balanced search tree is carried out as follows: First, a search is carried out in order to determine the location of the update. Second, the update is performed. Third, local balance constraints are reconsidered. Since balance constraints are usually based on path lengths or subtree sizes, these constraints may have been violated, because most often, an insertion will add at least one node to the tree and a deletion will remove at least one node from the tree. If there is a balance problem, this is fixed completely if possible, and otherwise it is fixed at the cost of introducing a new problem closer to the root. This problem is then handled recursively until it disappears or is moved all the way to the root, where balance problems are normally easily fixed.

The three phases described above are referred to as searching, updating, and rebalancing. Informally, *relaxed balance* is a term used for the following. If a search tree has been equipped with relaxed balance, the searching and updating have been uncoupled from the rebalancing. Thus, it is now possible to search and make an update without performing any rebalancing. For this to be well-defined, the balance constraints must be weakened (relaxed) in such a way that the tree after an update is still in the now broader class of trees. Additionally, the standard tree, which is made relaxed, should belong to the class, and the overall goal of the (presumably generalized and/or expanded) collection of rebalancing

* Supported in part by the Danish Natural Sciences Research Council (SNF).

** Department of Mathematics and Computer Science, University of Southern Denmark, Main Campus: Odense University, Campusvej 55, DK–5230 Odense M, Denmark. Email: {eljay,kslarsen}@imada.sdu.dk. Fax: +45 65 93 26 91

operations is to bring the tree back to fulfilling the constraints of the standard balanced tree.

The benefit of the uncoupling depends on the environment. Discussions of this can be found in many of the papers on the subject, but here is a brief account. In a sequential system, bursts of requests, possibly from an external source, can be served faster if rebalancing is "turned off" during the period. After the burst, rebalancing should gradually bring the tree back in balance, while requests are served at the same time. In a parallel (shared-memory) system, a naïve implementation would lock the root of the tree so frequently that the degree of parallelism would be extremely low. In relaxed structures, it is generally possible to exclusively lock only nodes which will be involved in pointer changes, instead of all nodes which might be involved in pointer changes. This implies that most of the exclusive locking will take place close to the leaves.

The cost of the relaxation is that the guaranteed worst-case bound of logarithmic path lengths is temporarily lost. The options are to trust that this does not become a problem for these short periods of time (maybe the requests are known to be close to uniform), to monitor path lengths and rebalance when some limit is exceeded, to dedicate a fixed minimum amount of rebalancing time to each update (or group of updates), or something else along those lines. The best solution can only be found when the specifics of the concrete scenario are known.

However, to ensure that as much time as possible is dedicated to request processing, it is vital that rebalancing, when it is performed, is performed efficiently. The difficulty in proving the various possible efficiency bounds on the run-time complexity is of course that after the structure has been relaxed, much less is known about its appearance. For instance, if k updates are performed on a standard balanced search tree of size n, usually $(k \log n)$, or fewer, rebalancing operations can easily be shown to completely rebalance the tree. In a relaxed version, path lengths can approach $\log n + k$, so if k is more than a constant, will $(k \log n)$ operations still suffice?

To make relaxed proposals as usefull as possible in the sequential as well as in the parallel setting, it is always required that rebalancing is carried out in local independent steps. However, in the sequential setting, this may not be mandatory.

Finally, relaxed balance is also a topic of theoretical interest. Search trees are some of the most important data structures, and this line of work answers some very fundamental questions concerning whether or not the traditional tight coupling between updating and rebalancing is necessary for the efficient rebalancing results to follow.

We give a very brief summary of the developments; more details can be found in [16], for example. Some of the ideas were initiated in [10, 15]. AVL-trees [1, 23] were investigated in [17, 25, 28], red-black trees [3, 10, 31] in [5–8, 16, 26, 27], and (a, b)-trees [13, 22], B-trees [4], 2-3-trees [2, 12] in [18, 19, 25]. In [20], a general result for balanced trees was developed, and in [9, 11, 21, 24, 32], some variations

of the standard schemes were investigated. Locking in a parallel setting was discussed in [6, 27].

In this paper, we investigate *layered trees* [30]. A relaxed version of layered trees was given in [29]. The primary contribution of this paper is to establish the complexity results which hold for the structure. We give our own presentation of layered trees with and without relaxed balance; partly to make the paper self-contained, but also partly because greater precision in the formulation of rebalancing operations is required in order for a proof of amortized constant rebalancing to be established.

The paper [29] primarily focuses on the design ideas, and on the important issue (not least in a parallel setting) of limiting restructuring. The principal difference between changing a pointer and updating balance information is that searching can proceed simultaneous with the information updating. Thus, if fine-grained locking is an option, limiting restructuring operations is more important. With the set-up in [29], the authors can show that only a constant amount of restructuring is necessary per update.

Layered Trees

It is possible to give a quite general definition of a layered tree [30]. However, to present the ideas in a form as simple as possible, we first give one very specific definition. Later, we discuss the more general alternatives.

A layered tree is a binary search tree. It is leaf-oriented, meaning that all keys are kept in the leaves. Internal nodes contain routers, which are of the same type as the keys and often copies of some of these. However, the only purpose of the routers is to guide the searches to the correct leaves. In a leaf-oriented binary tree, internal nodes always have two children.

Leaf-oriented trees are often the choice in large database-oriented applications because keys often have significant amounts of information attached. It is generally more efficient not to have to encounter this extra information when searching down the tree and when changing internal nodes due to rebalancing.

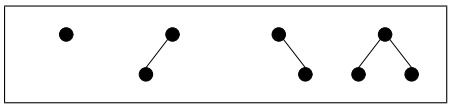


Fig. 1. The four basic configurations.

Additionally, when designing relaxed structures, there is no good way of carrying out deletions in a step-wise and local manner if the tree is not leaf-oriented. The problem is that if an internal node with two children should be deleted, the standard method for handling this is to switch keys with its internal

predecessor or successor and delete that node instead. However, that node can be located a non-constant distance away.

A leaf-oriented binary search tree is called layered if it can be constructed as described below from the configurations listed in Fig. 1:

1. Select one of the four basic configurations. The top node in the selected configuration will be the root of the whole tree.
2. Add a number of layers. One layer is added as follows: For each node u in the already constructed part of the tree which does not have a left (right) child, select one of the basic configurations and let the top node of the configuration be the left (right) child of u.
3. Construct a final layer of leaves, by adding a leaf everywhere a left or right child is missing.

We refer to the level of leaves as layer 0. The layer on top of that is layer 1 and so on. An edge connecting a node in some layer i with a node in the next layer $i+1$ is said to *cross the border* between the two layers. In the concrete implementation described in this paper, we assume that borders are explicitly stored in the structure. The most flexible way of doing this is by storing one bit in each node such that the bit is zero if it belongs to an even-numbered layer and one otherwise. The manipulation of this bit in connection with the operations to be discussed is easy, and we will not describe it explicitly. For easy future reference we define the following two subsets of basic configurations: the small configurations $\mathcal{C}_S = \{$  $\}$ and the large configurations $\mathcal{C}_L = \{$  $\}$.

Proposition 1. *The height of a layered tree with n leaves is bounded by $2\lfloor \log_2 n \rfloor$.*

Proof. We show by induction in the number of layers that a node in layer i has at least 2^i leaves in its subtree. This is trivial for the base case of a single leaf. For the induction step, we notice that any node u in the configurations from Fig. 1 at any level $i > 0$ has at least two descendants at level $i-1$. Since each of these, by the hypothesis, have at least 2^{i-1} leaves in their subtrees, u has 2^i leaves in its subtree. Thus, the layer of the root is at most $\lfloor \log_2 n \rfloor$, and so there are at most $\lfloor \log_2 n \rfloor + 1$ layers. Since the height of the highest basic configuration is two, the result follows.

Keys in the search tree come from a totally ordered domain. The keys in the leaves appear in strictly increasing order from left to right. A router in an internal node is greater than or equal to any key in its left subtree and less than any key in its right subtree.

In the light of this and Proposition 1, searching can obviously be performed in logarithmic time. The update operations, insert and delete, can also be performed in logarithmic time, and with at most a constant number of structural changes per update [30]. One way of describing this is as follows.

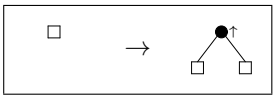
The general idea is to make the update, and register if there is a problem, i.e., if the tree is no longer constructed according to the layered tree definition.

Recursively, we remove the problem if possible, and otherwise move it to the next layer. At the root, any problem can be eliminated.

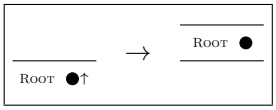
In the following, we describe the updating procedures. Proof of correctness follows later.

Insertions

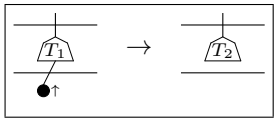
To insert a key, we search for the given key as usual in a search tree, and we end up at a leaf. If that leaf does not already contain the given key, a new leaf is created using operation *New leaf insertion*. The new key and the one already present in the existing leaf are arranged in order, and the key to the left is copied to the new internal node as its router.



New leaf insertion.



Up root.



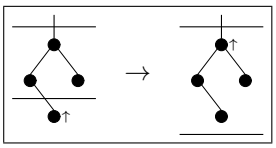
Up finish. $T_1 \in \mathcal{C}_S$. $T_2 \in \mathcal{C}_L$. $|T_2| = |T_1| + 1$.

The new internal node is on layer 0, which is not allowed, and is therefore equipped with a *push-up request* (↑). This push-up request is dealt with recursively as follows. If it reaches the root, the problem is solved using operation *Up root*. Otherwise, if there is room at the next layer, i.e., its parent is part of a configuration consisting of at most two nodes, the problem is solved using operation *Up finish*.

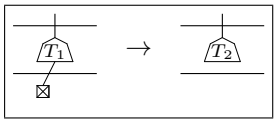
If the parent at the next layer is in a three-node configuration, the problem is moved up one layer using operation *Up push*.

Deletions

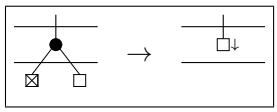
To delete a key, we search for the given key as usual in a search tree, and we end up at a leaf. If that leaf contains the given key, we proceed as follows (the leaf to be deleted is marked with two crossing lines in the figures). If the parent configuration has at least two nodes, using operation *Remove finish*, we can rearrange the nodes such that the leaf and its parent are deleted, while all configurations are still basic configurations.



Up push.



Remove finish. $T_1 \in \mathcal{C}_L$. $T_2 \in \mathcal{C}_S$. $|T_1| = |T_2| + 1$.



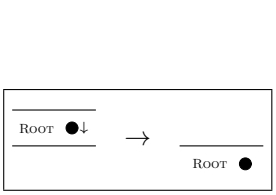
Remove continue.

If the next layer has a one-node configuration, we use operation *Remove continue*. This introduces a leaf at layer 1, which should be moved down to layer 0 before the tree can again be guaranteed to be a layered tree. We register this problem by marking the node with a *pull-down request* (↓).

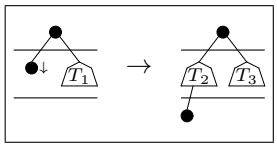
A pull-down request is handled recursively as follows. If it reaches the root, the problem is solved using operation *Down root*. Otherwise, if the sibling and parent configurations have at least three internal nodes together, then there are sufficiently many nodes locally such that the node can be moved down using either operation *Down finish 1* or *Down finish 2*, and at least one-node configurations can be created everywhere.

Finally, if the parent and sibling configurations contain only one node each, the problem is moved up one layer using operation *Down push*.

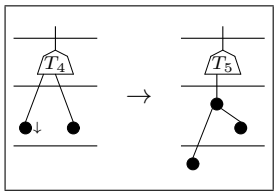
Observe that only operation *Remove continue* and *Down push* create pull-down requests. Since the only nodes which are marked are leaves or internal nodes with exactly one child on the next layer, such requests are created only if the marked node can be pulled down without violating the design criteria for layered trees.



Down root.



Down finish 1. $T_1 \in \mathcal{C}_L$. $T_2, T_3 \in \mathcal{C}_S$. $|T_1| = |T_2| + |T_3|$.



Down finish 2. $T_4 \in \mathcal{C}_L$. $T_5 \in \mathcal{C}_S$. $|T_4| = |T_5| + 1$.

Layered Trees with Relaxed Balance

To make the tree relaxed, we must allow that rebalancing can be interrupted at any time. In particular, its start can be delayed. In addition, the tree must be able to accommodate several updates for which the corresponding rebalancing has not been undertaken.

In addition to the basic configurations, several new configurations are allowed in the tree; any one or two node basic configuration where the bottom-most node is marked by a pull-down request, a zero-node configuration (a layer-crossing edge), and a four node configuration, where the top-most node is marked by a push-up request. The complete set of extra configurations (up to symmetric variants) are depicted in Fig. 2.

When an insertion is made, a leaf is replaced by an internal node with two leaves. If several insertions are made, large trees might be build this way without respecting the design criteria for relaxed layered trees. Such trees are always rooted at an internal node marked with a push-up request at layer 0. This part of the tree is called the *unstructured* part, while the part satisfying the design criteria for relaxed layered trees is called the *structured* part.

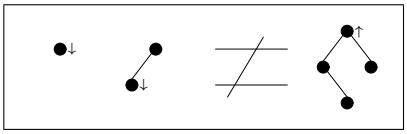


Fig. 2. Relaxed configurations.

Rebalancing is now carried out by moving a problem from the unstructured part into the structured part, and recursively towards the root, until the problem is removed.

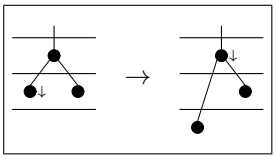
Since we cannot control when a deletion is actually carried out, the leaf to be deleted is marked physically for deletion by the operation *Delete mark*. Observe that the leaf might already be marked with a pull-down request, which is indicated by a parenthesized pull-down request ($(\Downarrow)$).

A leaf-oriented relaxed layered search tree can be constructed in the following way:

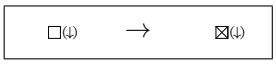
1. Select any configuration,except the layer-crossing edge. The top node of the selected configuration will be the root of the tree.
2. Add a number of layers: For each node u in the already constructed part of the tree, which does not have a left (right) child: if u is not marked with a pull-down request, and u is not on the layer above (a layer-crossing edge) add any of the node-containing configurations as the left (right) child of u. If u is marked with a pull-down request, add any configuration as the left (right) child of u, such that exactly one of the child configurations of a node marked by a pull-down request is a layer-crossing edge.
3. Construct the final layer by adding leaves, leaves marked for deletion, or unstructured trees to every node on the second to final layer that does not have a left (right) child, unless that node is marked by a pull-down request, in which case the node itself is made a leaf or a leaf marked for deletion.

Some operations involve the parent configuration, and some also the sibling configuration. An operation can generally not be carried out if the involved configurations are marked by requests. However, in some situations, we must allow that the sibling and parent configurations contain requests to avoid deadlocks.

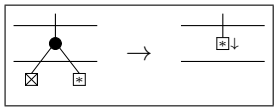
In the case of deletion, the sibling of the deleted leaf in operation *Remove continue*, might be marked for deletion, and is thus of course still marked after the application of the operation. This is indicated by an asterisk in the modified operation *Remove continue*. Analogously, two single-node siblings might both contain pull-down requests. Therefore, operation *Down finish 2* and *Down push* are modified to allow this.



Down push.

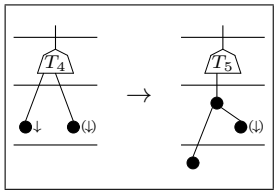


Delete mark.

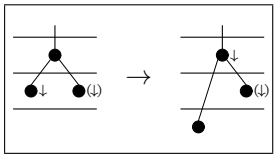


Modified *Remove continue.*

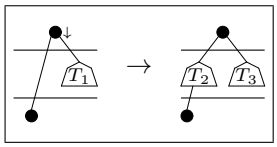
Furthermore, since we cannot control when updates are made, two new operations are needed to handle special cases of insertions. If a leaf is marked for deletion and an insertion is made at the very same leaf, the leaf is recycled as depicted in operation *Insert recycle*. If a leaf is marked with a pull-down request and an insertion is made at the very same leaf, the creation of the new internal node cancel out with the pull-down request; operation *Insert solve*.



Modified *Down finish 2*.
$T_4 \in \mathcal{C}_L$. $T_5 \in \mathcal{C}_S$.
$|T_4| = |T_5| + 1$.



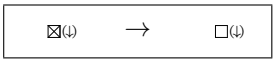
Modified *Down push.*



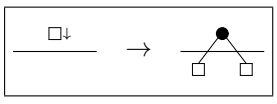
Down cancel. $|T_1| \geq 2$.
$|T_2|, |T_3| \geq 1$.
$|T_1| = |T_2| + |T_3|$.

Finally, pull-down requests are created if and only if both child configurations and the parent configuration are single nodes. However, when the request is to be resolved, this might not still be the case. One child is always a layer-crossing edge, while the other might be any other configuration. If the other child contains more than one node, these nodes can be rearranged such that it is no longer necessary to pull the marked node down. This is done by operation *Down cancel*. Observe that push-up requests among the rearranged nodes are analogously made obsolete, while pull-down requests must follow their respective layer-crossing child edges. It is an implicit precondition for applying any other operation involving pull-down requests that operation *Down cancel* cannot be applied.

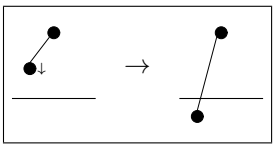
Analogously, the parent (the node marked by a pull-down request) might not be a single node anymore, in which case the node is just pulled down, using operation *Down finish 3*.



Insert recycle.



Insert solve.



Down finish 3.

Correctness and Complexity of Relaxed Balancing

By inspecting the individual operations, one can easily verify that the rebalancing operations satisfy the soundness property; applying any operation turns a relaxed layered tree into a relaxed layered tree.

Now we show that the collection of rebalancing operations is sufficient.

Theorem 1. *Completeness: Let T be a relaxed layered tree. While T contains at least one node marked by a request, some rebalancing operation can be applied.*

Proof. Let $\mathcal{R}$ denote the set of nodes marked by a request or marked for deletion on the top-most layer containing marked nodes.

If the root is in $\mathcal{R}$, then one of the *Root* operation can be applied. Assume that the root is not in $\mathcal{R}$. Assume that $\mathcal{R}$ contains some node u marked with a push-up request. Since u is top-most and non-root, the parent configuration is a basic configuration, and thus either operation *Up finish* or operation *Up push* can be applied. Observe that this is independent of whether or not u is located in the structured or the unstructured part of the tree.

Assume that $\mathcal{R}$ contains no nodes marked by a push-up request. Assume that $\mathcal{R}$ contains nodes marked by a pull-down request, and let u be such a node in a two node configuration, if any such exist. Consider the configurations below u. By the soundness property, one of these configurations is a layer-crossing edge. If the other configuration has at least 2 nodes, then operation *Down cancel* can be applied. Otherwise u can be moved down using operation *Down finish 3*.

Now assume that $\mathcal{R}$ contains nodes marked by a pull-down request, but that all these are single node configurations. Again, if a child which is not a layer-crossing edge contains at least two nodes, operation *Down cancel* can be applied. Otherwise we know that u's sibling configuration is either a single node (possibly marked by a pull-down request) or a basic configuration containing at least two nodes. In the first case, depending on whether the parent configuration of u has more than one node or not, either operation *Down push* or operation *Down finish 2* can be applied (recall that u was a top-most request, which means that u's parent configuration is a basic configuration). In the latter case, operation *Down finish 1* can be applied.

Finally, assume that $\mathcal{R}$ contains only leaves marked for deletion. By this assumption, the parent configuration of such a leaf contains no requests, so either operation *Remove finish* or *Remove continue* can be applied.

Amortized Constant Rebalancing

We use the standard potential function technique [33]. Any update operation creates exactly one problem in the unstructured part. Either a leaf marked for deletion or an internal node. This problem is either removed by a *finishing* rebalancing operation or moved into the structured part as a request which is then in turn moved a number of times using a *non-finishing* operation, until it is removed by a finishing operation.

Theorem 2. *Rebalancing is amortized constant.*

Proof. Assume that we remove every edge which connects two nodes in different layers. This splits the tree up into a collection of small trees with at most four nodes. We let $\mathcal{P}_i(T)$ for $i \in \{1, 2, 3, 4\}$ denote the number of pieces with i nodes resulting from splitting T.

We define the potential $\Phi(T)$ of the tree T as follows:

$$\Phi(T) = \mathcal{P}_1(T) + \mathcal{P}_2(T) + 3\mathcal{P}_3(T)$$

Any update operation, including the operation creating a request in the structured part, and any finishing operation may increase the potential, but it can do so by at most a constant. What remains is to show that every non-finishing operation decreases the potential by at least a constant to cover for its own application. The operations *Up push* and *Down push* are the only non-finishing rebalancing operations.

Operation *Up push* is applied only if the parent configuration of the node marked by the push-up request is a three node basic configuration. Recall that any node marked by a push-up request is the root of a four node configuration. Thus by the application, a three node configuration and a four node configuration is replaced by a four node, a two node, and a one node configuration, which decreases the potential by one.

Operation *Down push* is applied only if the parent and sibling configurations are single nodes. Furthermore, operation *Down push* is applied only if operation *Down cancel* cannot be applied. Thus, the children of the node marked by a pull-down request are a layer crossing edge and a single node, respectively. After the application, the node pulled down forms a two node configuration together with the single node child configuration at the child layer. Thus, four one node configurations are replaced by two one node configurations and a two node configuration, which decreases the potential by one.

Worst-Case Logarithmic Rebalancing

The previous theorem shows that rebalancing is amortized constant, if we start with an initially empty tree. However, if we start with a non-empty layered tree, we cannot use the theorem to guarantee a good complexity immediately. In the following, we show that even if we start with a layered tree, rebalancing is at most logarithmic in the worst-case.

Inspired by [16], we define a *count function* c as follows: If the tree is a standard layered tree, the count function is one on all leaves, and zero for all internal nodes. The *count sum* of a node u is the sum of the count function applied to all nodes in the subtree rooted at u, i.e., $\sum_{v \in T_u} c(v)$.

In a relaxed layered tree, the count function is maintained as follows: When an insertion is made, a leaf ℓ is replaced by an internal node with two leaves. The function value of the internal node is set to $c(\ell) - 1$, while the count function for both leaves is initialized to one.

When a leaf ℓ is actually deleted (not just marked), its parent u is deleted as well. The function value of the node v replacing the parent is then increased by $c(\ell) + c(u)$.

When nodes are rotated, some node is the root of the rotation. The function value of the root is assigned to the new root, while all the remaining function values are reassigned in-order to the remaining nodes involved in the rotation.

Since the count sum of the whole tree is incremented by insertions, but not decremented by deletions, the count sum of the root is always $n + i$ where n is the number of leaves in the tree the last time it was a standard layered tree, and i is the number of insertions.

Note that the values of the count function are always non-negative, and for leaves, they are positive.

We define the *relaxed layer* of a node u to be its layer in a layered tree unless u and u's parent are connected by a layer crossing edge. In this case, we define the relaxed layer to be one higher than its actual layer.

Lemma 1. *For any node u on relaxed layer j: $\sum_{v \in T_u} c(v) \geq 2^j$*

Proof. By induction on the number of operations on the tree since it was last a standard layered tree.

The base case follows by an argument similar to the proof of Proposition 1, since the count sum is exactly the number of leaves in any subtree.

It is easily verified that the result holds for any application of an update operation or an operation bringing a request into the structured part.

If nodes (in the structured part) are rearranged to form basic configurations, i.e., we also consider nodes marked by push-up request which are unmarked as a consequence of the rearrangement, the result follows immediately from the hypothesis since all such nodes have at least two descendants on the next layer.

If a node (marked by a pull-down request) is pulled down, we have two cases: It is either pulled down using operation *Down push*, in which case the relaxed layer is unchanged, or by a finishing operation, in which case the relaxed layer is decreased. In either case, the count sum is unchanged, and the result follows again immediately from the hypothesis.

Observe that any node marked by a push-up request has both its children on the same layer as itself. Thus, such a node is the root of a subtree with twice the count sum it needs, and the result follows from the hypothesis—even when the node is pushed to the next layer.

What remains is to verify that the result holds after the application of operation *Down cancel* when nodes marked by pull-down requests are rearranged. However, this follows from the way relaxed layers are maintained. Since both children of nodes marked by pull-down requests have the same relaxed layer—that of nodes on the next layer—the result follows from the hypothesis.

Theorem 3. *Rebalancing is worst-case logarithmic.*

Proof. Rebalancing after any update involves bringing the problem into the structured part of the tree, applying a number of non-finishing operations, and

applying one finishing operation. Hence, if the number of non-finishing operations can be bounded by a logarithmic term, the theorem follows. However, the application of a non-finishing operation moves a problem to a node on a higher relaxed layer, and as, by Lemma 1, the size of a subtree is exponential in the relaxed layer, there can be at most a logarithmic number of such layers.

More precisely, if i insertions (and possibly some deletions) are applied to a tree of size n, we get the bound $n+i = \sum_{v \in T_{\text{ROOT}}} c(v) \geq 2^{j_{\text{ROOT}}}$, where j_{ROOT} is the relaxed layer of the root.

Since at the root, the number of the layer and the relaxed layer must coincide, the root is in layer at most $\lfloor \log_2(n+i) \rfloor$. Including initial, finishing, and non-finishing operations, at most $\lfloor \log_2(n+i) \rfloor + 2$ operations can be applied per update.

Worst-Case Constant Restructuring

The following result is from [29].

Theorem 4. *Restructuring is worst-case constant.*

Proof. As was observed earlier, every finishing rebalancing operation removes at least one request. Hence, at most one finishing rebalancing operation can be applied per update. Since neither of the non-finishing operations make any structural changes, the theorem follows.

Concluding Remarks

The objective of this presentation of relaxed layered trees was twofold. We wanted to give a presentation precise enough that correctness and complexity proofs could be based on it. At the same time, we wanted to keep the presentation simple, in the spirit of the presentation of the standard version. The first objective has been obtained, but, admittedly, some of the simplicity is lost in the transition to a relaxed version. The problem is that the extra configurations, which are allowed in the relaxed setting, multiplies the total number of cases. With the level of precision which is required to establish all the complexity results, there does not seem to be any way to treat the operations at a higher level of abstraction to cut down on the number of cases.

On the positive side, we have shown that relaxed layered trees are among the best relaxed binary search trees. In particular, all the asymptotic complexities of [16] are matched: No update gives rise to more than a logarithmic number of rebalancing operations, of which at most one is restructuring. Additionally, rebalancing is amortized constant per update. It should also be noted that the potential function used in the proof for amortized constant rebalancing can be modified to satisfy the requirements for Theorem 1 in [14]. Thus, rebalancing in relaxed layered trees is exponentially decreasing with respect to the height.

As it is also pointed out in [29], there are many ways of tuning the operations to improve performance. For instance, several rebalancing operations can be

redefined or extended such that push-up requests and pull-down requests would cancel out when possible.

There is also a trade-off in the number of legal configurations and the number of rebalancing operations (and their complexity). For example, one could define relaxed layered trees without the two node configuration with the bottom-most node marked by a pull-down request. However, then the set of operations is increased and some operations must be made larger.

References

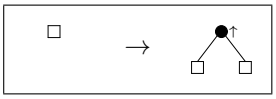
1. G. M. Adel'son-Vel'skiĭ and E. M. Landis. An Algorithm for the Organisation of Information. *Doklady Akadamii Nauk SSSR*, 146:263–266, 1962. In Russian. English translation in *Soviet Math. Doklady*, 3:1259-1263, 1962.
2. Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.
3. R. Bayer. Symmetric Binary B-Trees: Data Structure and Maintenance Algorithms. *Acta Informatica*, 1:290–306, 1972.
4. R. Bayer and E. McCreight. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica*, 1(3):97–137, 1972.
5. Joan Boyar, Rolf Fagerberg, and Kim S. Larsen. Amortization Results for Chromatic Search Trees, with an Application to Priority Queues. In *Fourth International Workshop on Algorithms and Data Structures*, volume 955 of *Lecture Notes in Computer Science*, pages 270–281. Springer-Verlag, 1995.
6. Joan Boyar, Rolf Fagerberg, and Kim S. Larsen. Amortization Results for Chromatic Search Trees, with an Application to Priority Queues. *Journal of Computer and System Sciences*, 55(3):504–521, 1997.
7. Joan F. Boyar and Kim S. Larsen. Efficient Rebalancing of Chromatic Search Trees. In *Proceedings of the Third Scandinavian Workshop on Algorithm Theory*, volume 621 of *Lecture Notes in Computer Science*, pages 151–164. Springer-Verlag, 1992.
8. Joan F. Boyar and Kim S. Larsen. Efficient Rebalancing of Chromatic Search Trees. *Journal of Computer and System Sciences*, 49(3):667–682, 1994.
9. Joaquim Gabarró, Xavier Messeguer, and Daniel Riu. Concurrent Rebalancing on HyperRed-Black Trees. In *Proceedings of the 17th International Conference of the Chilean Computer Science Society*, pages 93–104. IEEE Computer Society Press, 1997.
10. Leo J. Guibas and Robert Sedgewick. A Dichromatic Framework for Balanced Trees. In *Proceedings of the 19th Annual IEEE Symposium on the Foundations of Computer Science*, pages 8–21, 1978.
11. S. Hanke, Th. Ottmann, and E. Soisalon-Soininen. Relaxed Balanced Red-Black Trees. In *Proc. 3rd Italian Conference on Algorithms and Complexity*, volume 1203 of *Lecture Notes in Computer Science*, pages 193–204. Springer-Verlag, 1997.
12. J. E. Hopcroft. Title unknown. Unpublished work on 2-3 trees, 1970.
13. S. Huddleston and K. Mehlhorn. A New Data Structure for Representing Sorted Lists. *Acta Informatica*, 17:157–184, 1982.
14. Lars Jacobsen and Kim S. Larsen. Exponentially Decreasing Number of Operations in Balanced Trees. In *Seventh Italian Conference on Theoretical Computer Science*, 2001. This volume.

15. J. L. W. Kessels. On-the-Fly Optimization of Data Structures. *Communications of the ACM*, 26:895–901, 1983.
16. Kim S. Larsen. Amortized Constant Relaxed Rebalancing using Standard Rotations. *Acta Informatica*, 35(10):859–874, 1998.
17. Kim S. Larsen. AVL Trees with Relaxed Balance. *Journal of Computer and System Sciences*, 61(3):508–522, 2000.
18. Kim S. Larsen and Rolf Fagerberg. B-Trees with Relaxed Balance. In *Proceedings of the 9th International Parallel Processing Symposium*, pages 196–202. IEEE Computer Society Press, 1995.
19. Kim S. Larsen and Rolf Fagerberg. Efficient Rebalancing of B-Trees with Relaxed Balance. *International Journal of Foundations of Computer Science*, 7(2):169–186, 1996.
20. Kim S. Larsen, Thomas Ottmann, and Eljas Soisalon-Soininen. Relaxed Balance for Search Trees with Local Rebalancing. In *Fifth Annual European Symposium on Algorithms*, volume 1284 of *Lecture Notes in Computer Science*, pages 350–363. Springer-Verlag, 1997. To appear in *Acta Informatica*.
21. Kim S. Larsen, Eljas Soisalon-Soininen, and Peter Widmayer. Relaxed Balance through Standard Rotations. In *Fifth International Workshop on Algorithms and Data Structures*, volume 1272 of *Lecture Notes in Computer Science*, pages 450–461. Springer-Verlag, 1997. To appear in *Algorithmica*.
22. Kurt Mehlhorn. *Sorting and Searching*, volume 1 of *Data Structures and Algorithms*. Springer-Verlag, 1984.
23. Kurt Mehlhorn and Athanasios Tsakalidis. An Amortized Analysis of Insertions into AVL-Trees. *SIAM Journal on Computing*, 15(1), 1986.
24. Xavier Messeguer and Borja Valles. HyperChromatic trees: a fine-grained approach to distributed algorithms on RedBlack trees. Tech. Report LSI-98-13-R, Departament de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya, 1998.
25. 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.
26. Otto Nurmi and Eljas Soisalon-Soininen. Uncoupling Updating and Rebalancing in Chromatic Binary Search Trees. In *Proceedings of the Tenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 192–198, 1991.
27. Otto Nurmi and Eljas Soisalon-Soininen. Chromatic Binary Search Trees—A Structure for Concurrent Rebalancing. *Acta Informatica*, 33(6):547–557, 1996.
28. Otto Nurmi, Eljas Soisalon-Soininen, and Derick Wood. Relaxed AVL Trees, Main-Memory Databases and Concurrency. *International Journal of Computer Mathematics*, 62:23–44, 1996.
29. Th. Ottmann and E. Soisalon-Soininen. Relaxed Balancing Made Simple. Technical Report 71, Institut für Informatik, Universität Freiburg, 1995.
30. Thomas Ottmann and Derick Wood. Updating Binary Trees with Constant Linkage Cost. *International Journal of Foundations of Computer Science*, 3(4):479–501, 1992.
31. Neil Sarnak and Robert E. Tarjan. Planar Point Location Using Persistent Search Trees. *Communications of the ACM*, 29:669–679, 1986.
32. Eljas Soisalon-Soininen and Peter Widmayer. Relaxed Balancing in Search Trees. In *Advances in Algorithms, Languages, and Complexity*, pages 267–283. Kluwer Academic Publishers, 1997.

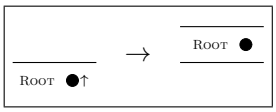
33. Robert Endre Tarjan. Amortized Computational Complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985.

Overview: All the Operations from within the Paper

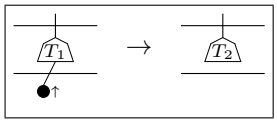
The Sequential Structure



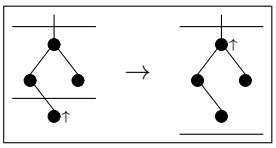
New leaf insertion.



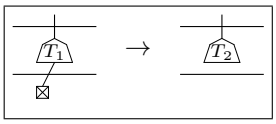
Up root.



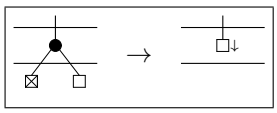
Up finish. $T_1 \in \mathcal{C}_S$. $T_2 \in \mathcal{C}_L$. $|T_2| = |T_1| + 1$.



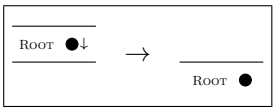
Up push.



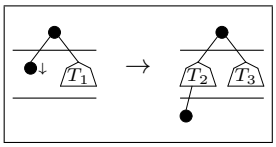
Remove finish. $T_1 \in \mathcal{C}_L$. $T_2 \in \mathcal{C}_S$. $|T_1| = |T_2| + 1$.



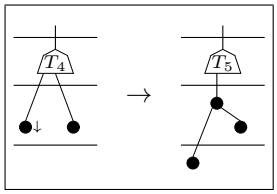
Remove continue.



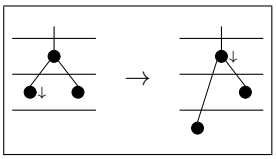
Down root.



Down finish 1. $T_1 \in \mathcal{C}_L$. $T_2, T_3 \in \mathcal{C}_S$. $|T_1| = |T_2| + |T_3|$.

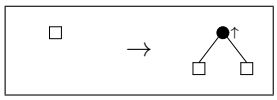


Down finish 2. $T_4 \in \mathcal{C}_L$. $T_5 \in \mathcal{C}_S$. $|T_4| = |T_5| + 1$.

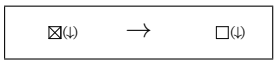


Down push.

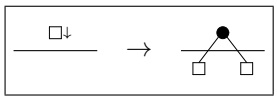
The Relaxed Structure



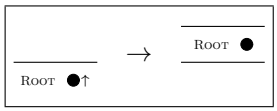
New leaf insertion.



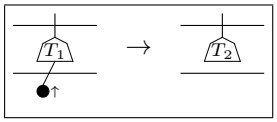
Insert recycle.



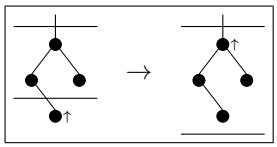
Insert solve.



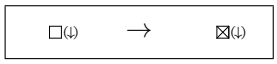
Up root.



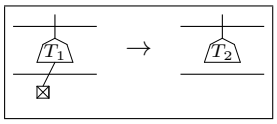
Up finish. $T_1 \in \mathcal{C}_S$. $T_2 \in \mathcal{C}_L$. $|T_2| = |T_1| + 1$.



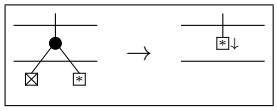
Up push.



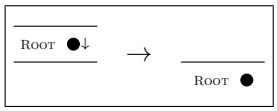
Delete mark.



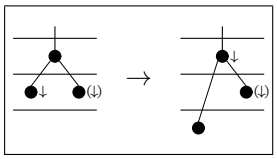
Remove finish. $T_1 \in \mathcal{C}_L$. $T_2 \in \mathcal{C}_S$. $|T_1| = |T_2| + 1$.



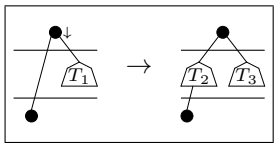
The modified *Remove continue*.



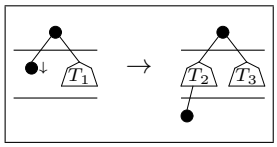
Down root.



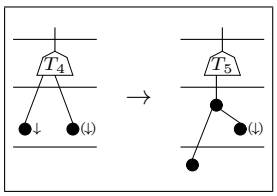
The modified *Down push*.



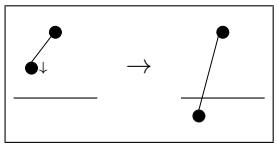
Down cancel. $|T_1| \geq 2$. $|T_2|, |T_3| \geq 1$. $|T_1| = |T_2| + |T_3|$.



Down finish 1. $T_1 \in \mathcal{C}_L$. $T_2, T_3 \in \mathcal{C}_S$. $|T_1| = |T_2| + |T_3|$.



The modified *Down finish 2*. $T_4 \in \mathcal{C}_L$. $T_5 \in \mathcal{C}_S$. $|T_4| = |T_5| + 1$.



Down finish 3.