Relaxed Balancing Made Simple

Th. Ottmann* E. Soisalon-Soininen†

November 27, 1995

Abstract

Relaxed balancing means that, in a dictionary stored as a balanced tree, the necessary rebalancing after updates may be delayed. This is in contrast to *strict* balancing meaning that rebalancing is performed immediately after the update. Relaxed balancing is important for efficiency in highly dynamic applications where updates can occur in bursts. The rebalancing tasks can be performed gradually after all urgent updates, allowing the concurrent use of the dictionary even though the underlying tree structure is not completely in balance. The contribution of the present paper is that we introduce a new scheme for relaxed balancing, which is obtained by a simple generalization of strict balancing. Our approach implies a simple proof of the fact that the number of the needed rebalancing operations (to put the tree in balance) for relaxed balancing is the same as for strict balancing.

1 Introduction

A *dictionary* is a scheme for storing a set of data such that the operations *search*, *insert*, and *delete* can be carried out efficiently. Standard implementations of dictionaries using balanced search trees like AVL-trees, red-black-trees, half-balanced trees, and others presuppose that each update operation

*Institut für Informatik, Universität Freiburg, Am Flughafen 17, D-79110 Freiburg i.Br. E-mail: ottmann@informatik.uni-freiburg.de

†Laboratory of Information Processing Science, Helsinki University of Technology, Otakaari 1 A, FIN-02150 Espoo, Finland. E-mail: ess@cs.hut.fi

is followed by a sequence of rebalancing steps which restore the respective balance condition. Maintaining the balance conditions assures that the trees can not degenerate into linear lists and search and update operations can be performed in a number of steps which is always logarithmic in the number of keys stored in a tree.

In a concurrent environment, however, uncoupling the updating (insertion and deletion) from the rebalancing transformations may increase the possible amount of concurrency and speed up updates considerably. This leads to the notion of *relaxed balance*. Instead of requiring that the balance condition is restored immediately after each update operation the actual rebalancing transformations can be delayed arbitrarily and interleaved freely with search and update operations.

The literature contains a number of proposals for relaxed balancing in binary and multiway search trees [9], [10], [11], [12], [13].

The previous solutions have different drawbacks, however, which we will avoid by the solution proposed in this paper. Except for the recent paper [18], all previous solutions are not based on the standard balancing transformations but require a large number of different new balancing transformations. Soisalon-Soininen and Widmayer [18] show that by maintaining the actual heights of subtrees it is possible to assure that to perform one of the standard balancing transformations for height-balanced trees (rotation or double rotation) is locally beneficial. On the other hand, even their solution does not guarantee that the number of structural changes in order to restore the height-balance condition after k updates is of order $O(k)$.

In this paper we propose a new solution to the relaxed balancing problem which considerably simplifies previous solutions, but has all the nice desirable features one might expect from a relaxed balancing scheme. We restrict ourselves to the case of binary trees in order to clarify the ideas underlying our solution as much as possible. The interested reader should be able to extend the solution to multiway trees also.

The main aim of our proposal is to extend the constant-linkage-cost update algorithm for binary trees implicitely contained in [15] in such a way that updates and local structural changes are uncoupled and may be arbitrarily delayed and interleaved. However, as in [18] we require that a sequence of delayed update requests is settled bottom up. Another key idea in our solution is the assumption that the deletion of a key in a tree leads to a *removal request* only; the actual removal of a leaf is considered to be a part

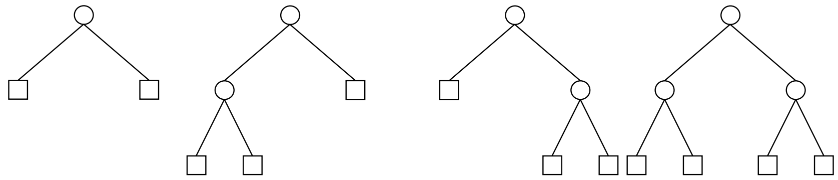

Figure 1: Set Z of stratum trees

of the structural change to restore the balance condition. This has the effect that sequences of deletions can be accumulated very similar to sequences of insertions.

2 Layered binary search trees

The trees in this paper are leaf-oriented binary search trees, which are full binary trees (each node has either two or no children). The nullary nodes of a tree are called the external nodes or leaves while the other nodes are said to be internal nodes. We assume that the keys (chosen from a totally ordered universe) are stored in the leaves of the binary tree. The internal nodes contain routers, which guide the search from the root to a leaf.

For simplicity, we do not distinguish between search trees with stored items (in left-to-right order at the leaves) and their underlying graph structure.

A binary search tree is *balanced* if its height is bounded by $O(logN)$, where N is the numer of keys stored in the tree.

We will consider search trees which are *stratified* or *layered*, i.e. they consist of a number of strata or layers. As the balance condition it is required that each leaf has the same stratum height, i.e. the same distance to the root when counting the numer of layers (but not the numer of edges) of a root-to-leaf path.

In order to be more specific, from now on we assume that Z is the set (shown in Figure 1) of 4 binary trees of heights 1 and 2.

We consider the class of all *Z-stratified trees*, which is the smallest class of trees obtained as follows:

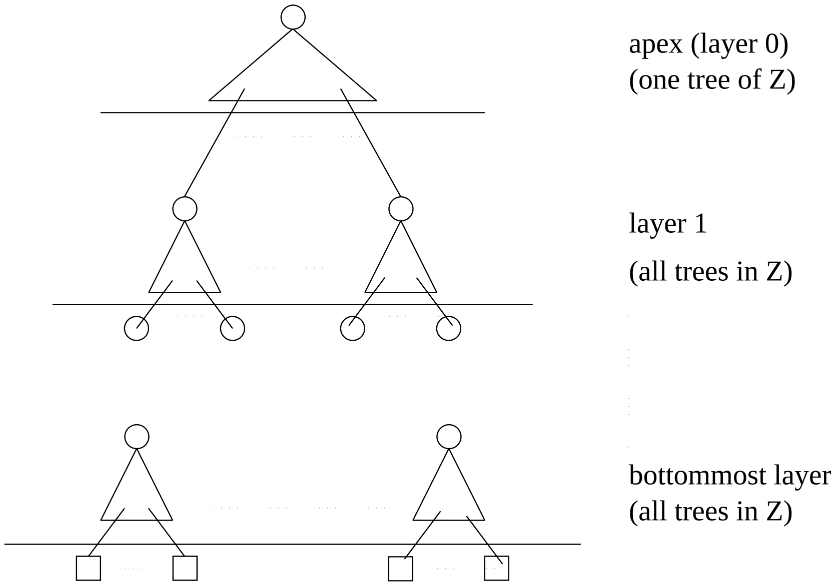

Figure 2: Structure of a Z-stratified tree

1. Each tree in Z is a Z-stratified tree.
2. Given a Z-stratified tree, append to each of its leaves one of the trees in Z. Then the appended trees form a new stratum (layer) and the resulting tree is again a Z-stratified tree.

Z-stratified trees can, therefore, be considered as being decomposed into a number of different layers as shown schematically in Figure 2. Note that the decomposition of a given binary tree into strata (showing that the tree is Z-stratified) may be not unique. Therefore we consider trees with different decompositions to be different and always assume that the stratum borders are explicitely given.

A possible way to ensure this is to color the nodes immediately above and below a stratum border by different colors.

Note that the class of Z-stratified trees as defined above coincides with the class of symmetric binary B-trees [2], the class of half-balanced trees [14], and the class of red-black-trees [7] when ignoring their respective update algorithms.

Furthermore, it should be clear that the height (maximal number of edges on a root-to-leaf path) of a Z-stratified tree with N leaves is $O(logN)$.

In the next sections we will describe the update (insertion and deletion) algorithms for Z-stratified trees. The idea behind the rebalancing transformations to be performed after the insertion or deletion of a key is that either a structural change in the local vicinity of a node completely settles the problem of restoring the balance condition (being a Z-stratified tree) or (exclusively!) this problem is recursively shifted to the next higher layer without any structural change on the current layer at all. By "structural change" we always mean some pointer changes. Color changes, i.e. any local move of a stratum border is not counted. This is justified by the fact that it is not necessary to lock nodes involved in color changes in a concurrent environment. For, search operations may very well be performed concurrently with color changes. A search can be misled only if it collides with a structural (pointer) change. We will first explain the algorithms to handle single insertions and deletions only, inclusive their rebalancing transformations. We will then explain how sequences of updates can be accumulated and the rebalancing transformations can be uncoupled from the updates.

3 Insertions

In order to insert a new key x into a Z-stratified search tree we locate first its position among the leaves and replace the leaf by an internal node with two leaves. The two leaves now store the old key (where the search ended) and the new key x.

Note that the resulting tree is no longer a Z-stratified search tree any more since there occurs now an internal node immediately below the bottom most stratum border. In order to correct this and to restore the balance condition we deposit a *push-up* request "$\uparrow$", cf. Figure 3:

The task to be performed for a node with a push-up request is to push it over the stratum border below which it occurs. However, this may violate the Z-stratified search tree structure. Our primary goal is to settle a push-up request with only a constant number of structural changes in the tree. Therefore, we proceed in such a way that a push-up request leads

1. either to a structural change involving a few nodes occuring on one

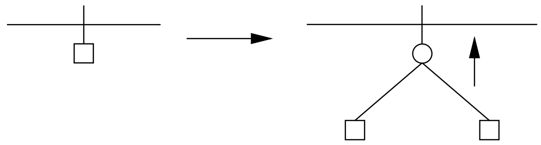

Figure 3: Insertion of a new item and depositing a push-up request

layer

2. or (exclusively) to a push-up request for a node immediately below the stratum border of the next higher layer and no structural change at all.

We distinguish between two cases:

Case 1 [There is "enough room" on the next higher layer]

In this case the node which we want to push up over the stratum border is opened to a stratum tree from Z which has not the maximal number 4 of leaves hanging over the stratum border. Then we can replace the stratum tree by a new one with one more leaf, append all subtrees in the same order, and have restored the balance condition. Figure 4 shows the structural changes to be carried out in this case, where we omitted all symmetric cases.

Case 2 [There is not "enough room" on the next higher layer]

In this case the node which we want to push up over the stratum border is appended to one of the 4 leaves of a full binary tree t of height 2 occuring on the next higher layer. Now we cannot settle the push-up request by a local structural change on the next layer. Therefore, we simply move the node with the push-up request over the stratum border and recursively transfer the problem of pushing a node up over a stratum border to the root of the stratum tree t. Figure 5 shows one of the 4 possibilities where the node with the push-up request may occur. Note that no structural change (pointer change) occurs in *Case 2*!

We tacitly assume that a new layer and a new apex are introduced as soon as the push-up request has reached the root of the original Z-stratum tree. That means, Z-stratum trees grow at the root by splitting off a new apex from an overfull topmost layer.

As we have seen, a single insertion may lead to a push-up request for a node hanging down immediately below the bottommost stratum border.

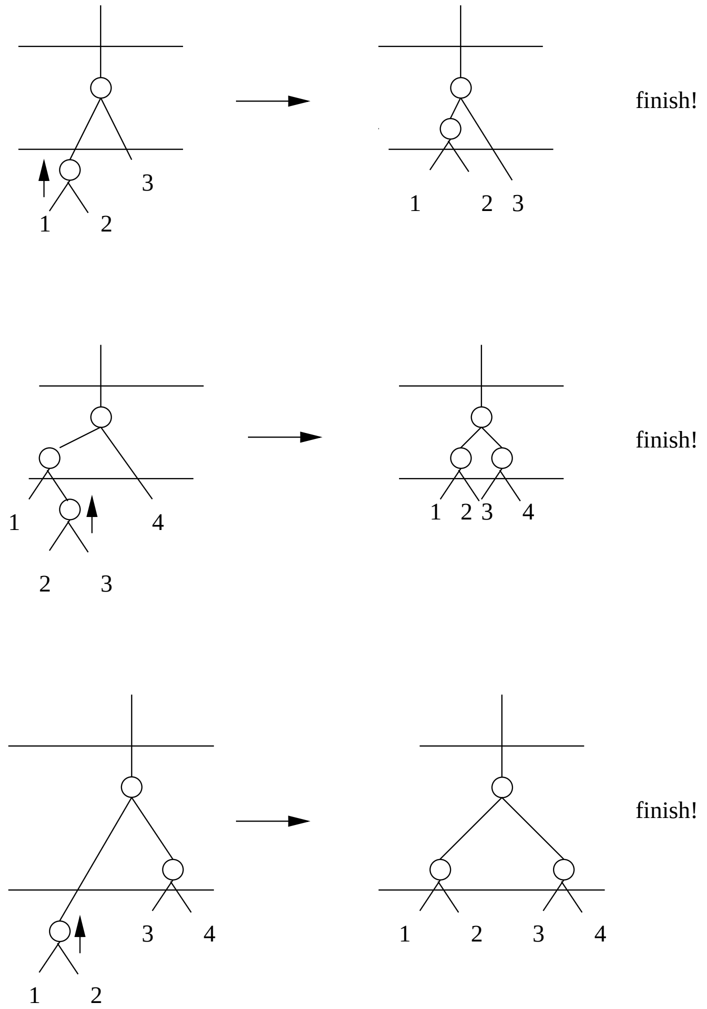

Figure 4: Local structural changes settling a push-up request

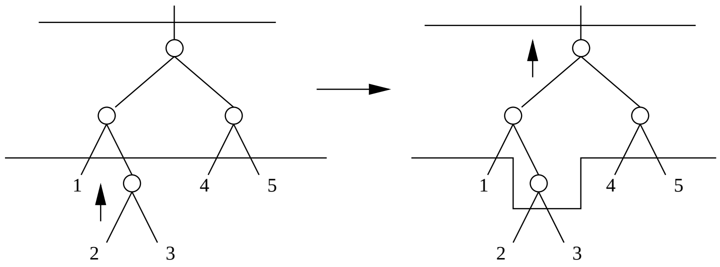

Figure 5: Recursive shifting of a push-up request to next higher layer

Settling the push-up request may either lead to a series of push-up requests for nodes on the search path below the border of higher layers and no restructuring at all or to a local structural change (replacement of one stratum tree by another one) and stop. Therefore, it should be obvious that one push-up request can be settled by a constant number of pointer changes.

We will now explain how a *sequence of insertions* can be handled in such a way that it is not necessary to restructure the tree after each single insertion. First, we observe that push-up requests can be accumulated and bubble up in the tree concurrently as long as it is assured that no two of them interfere at the same stratum trees. That is, if there are several push-up requests for nodes hanging down from the same stratum tree at a certain layer, we treat them one after the other in an arbitrary order in accordance with *Case 1* and *Case 2* described above. As soon as one push-up request has disappeared (by a structural change or by a recursive shift to the next higher layer) we can already continue to settle the next push-up request.

This solves already the problem of how to accumulate a sequence of insertions which all fall into different leaves of the originally given Z-stratified tree. We simply deposit a push-up request for each newly created internal node occurring immediately below the bottommost stratum border.

Now observe that we can do the same also in the case when an (accumulated) insertion falls into a leaf which is not a leaf of the originally given Z-stratum tree but which has been created by a previous insertion. This means, that we can accumulate push-up requests for nodes hanging down from the bottommost stratum border and settle them as described above. We just settle them in such a way that we always take the nodes immediately below the bottommost stratum border of the originally given Z-stratified tree first.

In this way, a sequence of insertions may lead to a growth of the originally given Z-stratum tree below the bottommost stratum layer as in the case of natural (unbalanced) binary search trees. Each of the newly created nodes has a push-up request attached to it. The push-up requests are settled as described above in top-down manner but otherwise in an arbitrary order. If all push-up requests have been settled (and disappeared), the resulting tree is again a Z-stratified search tree. Figure 6 schematically shows an intermediate stage of insertions and structural changes for a Z-stratified tree.

We summarize the result obtained so far by the following theorem.

Theorem 1 *Let T be a Z-stratified search tree and assume that insertions and rebalancing transformations (possibly interspersed) are applied to T. Then the number of structural changes (pointer changes) needed to restore the balance condition (of being Z-stratified) is at most $O(i)$ where i is the number of insertions performed.*

4 Deletions

In order to delete a key from a Z-stratified search tree, we first locate the leaf where the key is stored and deposit a *removal request* "$\times$" at that leaf.

A removal request may either be settled immediately at the bottommost layer by a small structural change or may lead to a *pull-down* request "$\downarrow$" for a node occurring immediately above a stratum border. (Initially, this is always the bottommost stratum border.)

If a removal request applies to a leaf of a tree from Z with 3 or 4 leaves it can easily be settled by replacing the tree with another tree from Z which has one leaf less. A removal request applied to a leaf of the tree in Z with only two leaves leads to a pull-down request. This is depicted in Figure 7 where we again have omitted all symmetric cases.

A pull-down request always applies to a node occurring immediately above a stratum border. In order to settle a pull down request we have to move the node below the stratum border and we have to assure that we restore the layered tree structure by only a constant number of structural changes in the tree. Therefore, a pull-down request consists of a movement of the node below the stratum border above which it occurs and

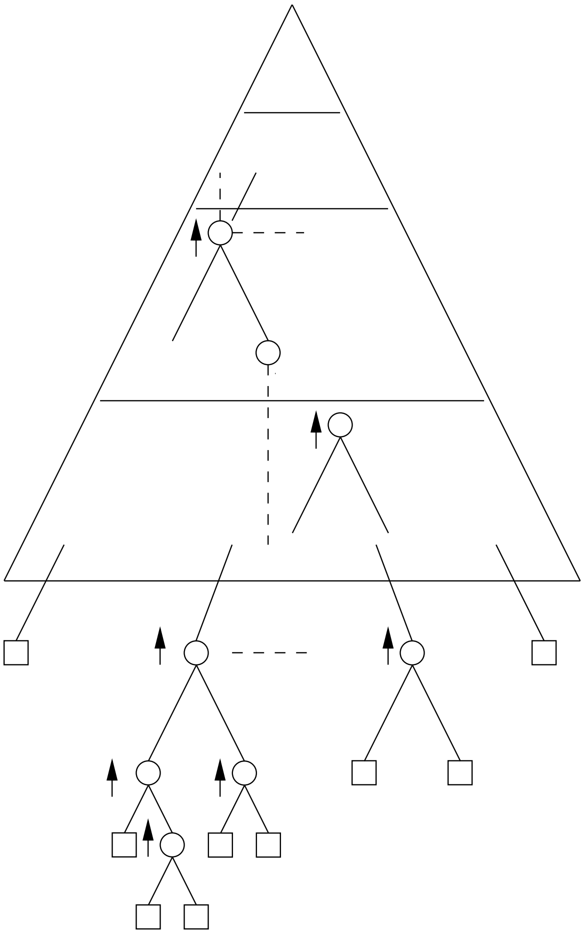

Figure 6: Z-stratified tree after a sequence of insertions with unsettled push-up requests

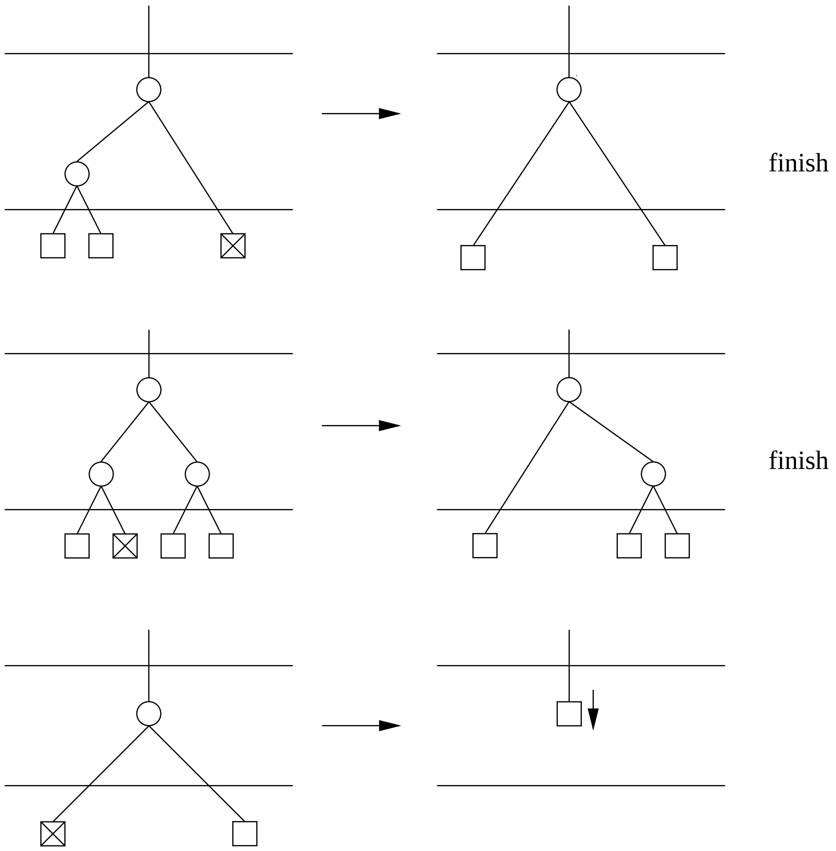

Figure 7: Handling of a removal request

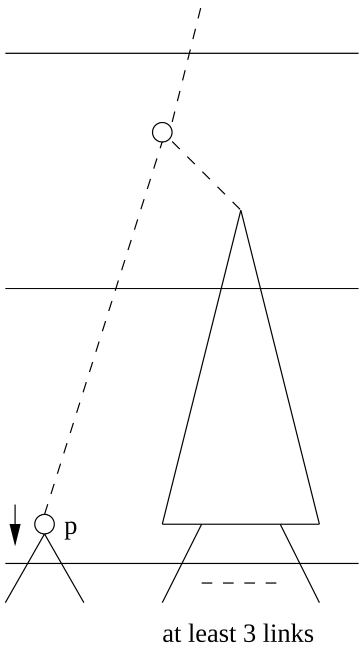

Figure 8: Node with pull-down request has enough nodes in its vicinity

1. either a local structural change of the Z-stratum tree at the layer where the node with the pull-down request occurs

2. or (exclusively) in a recursive call of a pull-down request for a node immediately above the stratum border of the next higher layer and no structural change in the tree at all.

We again distinguish two cases depending on the number of nodes in the immediate vicinity of the node with the pull-down request.

Case 1 [There are “enough nodes” in the vicinity of node p with a pull-down request]

In this case we assume that the predecessors of p on the same and on the next higher layer have so many successors despite of p that at least 3 pointers cross the stratum border above which p lies, cf. Figure 8 for a graphical explanation of this situation.

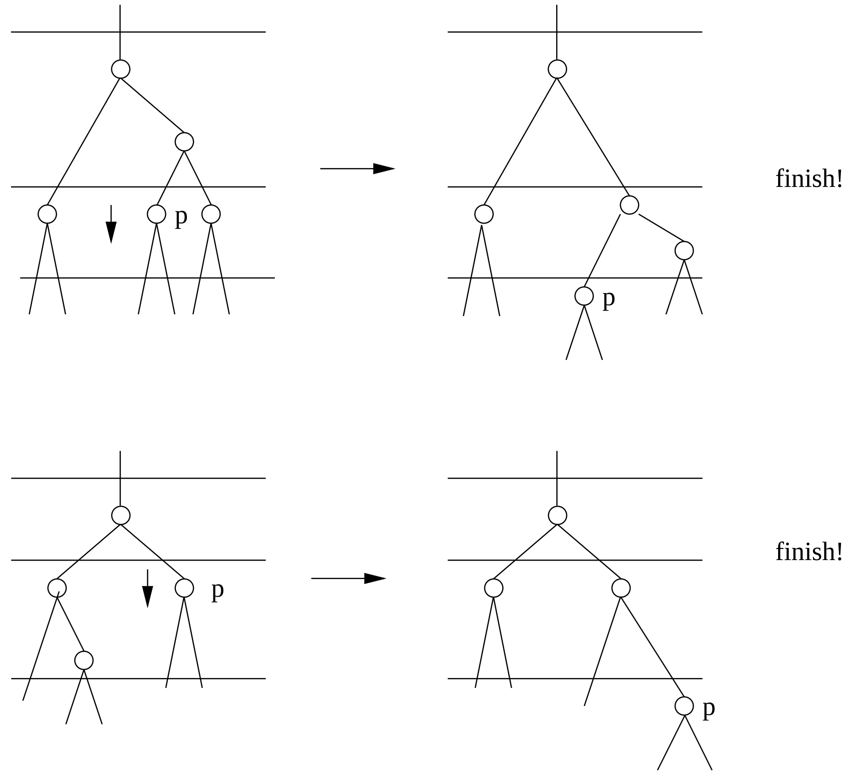

Figure 9: Instances of structural transformations settling a pull-down request

In this case we can obviously move node p below the stratum border and rearrange the tree locally by choosing only stratum trees from the set Z such that the Z-stratified structure of the tree is restored.

Our description of the generic structural transformation settling a pull-down request has a considerable amount of freedom left and allows to fine tune the algorithm, if desired. In order to be more specific we show in Figure 9 a few possible transformations which can be subsumed under the generic *Case 1*.

Case 2 [There are "not enough" nodes in the vicinity of node p with a pull-down request]

Now we assume that the node p with the pull-down request has the min-

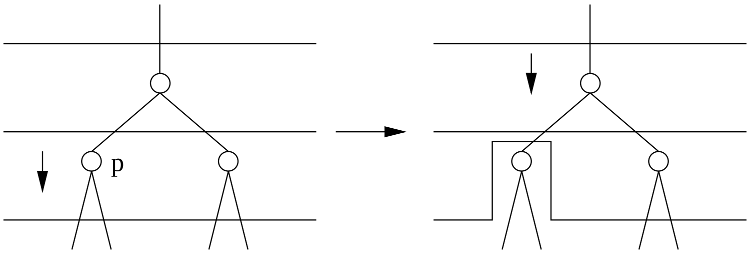

Figure 10: Recursive shift of a pull-down request to next higher layer

imal number of relatives of its own and its preceeding layer. In this case we cannot settle the pull-down request by a structural transformation in the vicinity of p. Therefore, we move p below the stratum border and recursively shift the pull-down request to p's father on the next higher level, see Figure 10 for a graphical explanation.

Note that in this case no structural (pointer) change is carried out.

We also tacitly assume that one layer at the top of a Z-stratum tree disappears when a pull-down request has reached the root of the tree. That is, the transformation in *Case 2* shown by Figure 10 also covers the case where p has no father any more.

As we have seen, a single deletion from a Z-stratified tree leads to a removal request for a leaf of the tree. This removal request can either be settled immediately by a small structural change at the bottommost layer or triggers a pull-down request. Pull-down requests may bubble up in the tree by recursive shifts to higher layers until they can be finally settled by a structural transformation involving a (small) constant number of nodes and pointers.

We will now explain how a *sequence of deletions* can be handled in such a way that it is not necessary to restructure the tree immediately after each single deletion. First, we observe that deletions can be accumulated by depositing a removal request at a leaf for each single deletion. Several removal requests can be settled concurrently in an arbitrary order as long as it is assured that no two removal requests interfere at the same stratum tree. One has to settle them one after the other (in an arbitrary order). This, in particular, implies that a removal request for a node which has attached

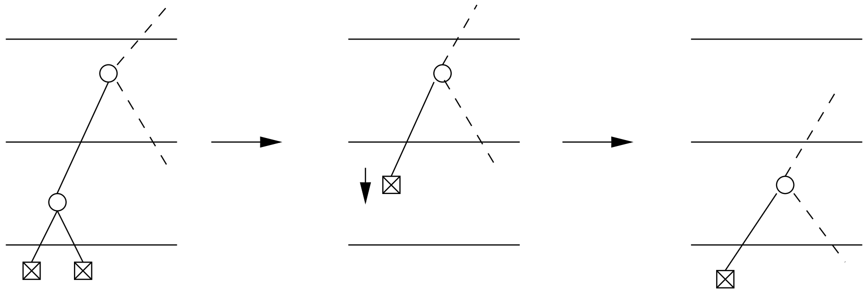

Figure 11: Handling removal requests for two brother-leaves

a pull-down request (as the result of a previously settled removal request) cannot be handled before the pull-down request has been settled or bubbled up in the tree. This assumption assures that two removal requests appended to both leaves of the same node of the Z-stratum tree with 2 leaves on the bottommost layer are always handled correctly: Before the second removal request is settled a pull-down request has moved the involved leaf back below the bottommost stratum border, cf. Figure 11 for a graphical explanation.

In this way it is assured that every sequence of (accumulated) deletions and their respective rebalancing transformations can be interleaved; they are correctly settled just as if they were performed serially.

Figure 12 schematically shows an intermediate stage of deletions and structural changes for a Z-stratified search tree.

Figure 13 shows a sequence of transformations which settle removal and pull-down requests in a Z-stratified stratum tree with 3 strata.

We summarize the result of this section by the following theorem.

Theorem 2 *Let T be a Z-stratified search tree and assume that deletions and rebalancing transformations (possibly interspersed) are applied to T. Then the number of structural changes (pointer changes) needed to restore the balance condition (of being Z-stratified) is at most $O(d)$ where d is the number of deletions.*

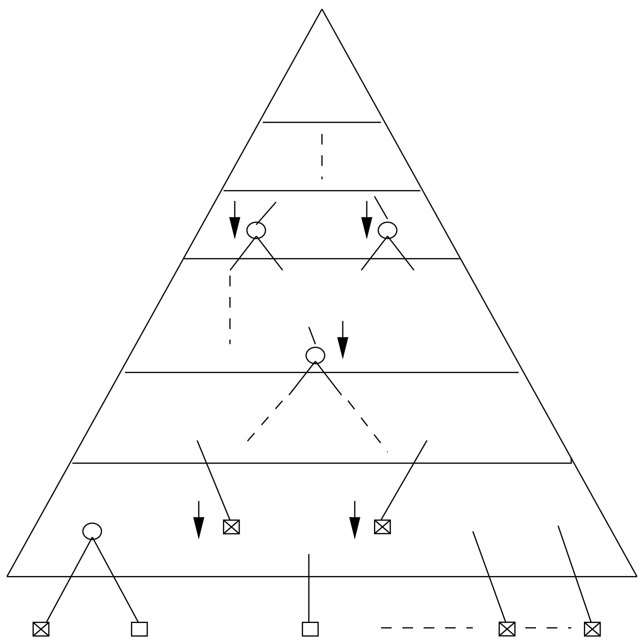

Figure 12: Z-stratified tree after a sequence of deletions with unsettled removal and pull-down requests

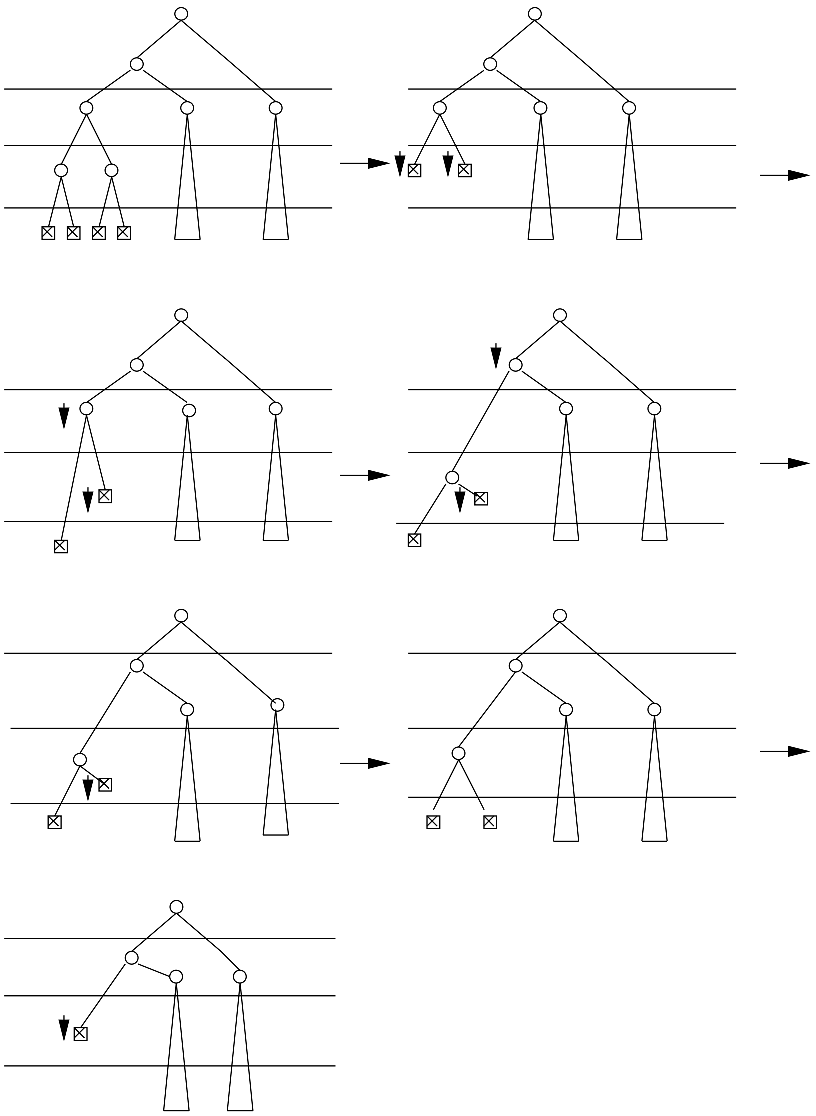

Figure 13: Sequence of removal and pull-down requests

5 Interleaving updates and rebalancing transformations freely

As we have explained in the previous sections Z-stratified search trees grow and shrink at the top when multiple insertions and deletions are applied to them and the structural changes which we have described above are performed in order to restore the Z-stratified tree structure. New keys enter the tree from below at the bottommost stratum border; keys are deleted by extracting them again from below.

We will now explain how arbitrary interleaved sequences of insertions, deletions and restructurings can be handled.

If an insertion or a deletion falls into a leaf which is a leaf of the originally given stratum tree, nothing new happens except for the possibility that now an insertion may fall into a leaf with a removal request. It should be obvious how to proceed in this case: Abandon the removal request and (re-)insert the key at that leaf, see Figure 14.

If, on the other hand, a deletion falls into a leaf which has been created by a previous insertion but has not yet moved upwards to the border of the bottommost layer, one can delete the leaf (and an internal node) and abandon one pull-up request. Figure 15 shows an example for this event.

Apart from these minor additions nothing new is necessary in order to allow that insertions, deletions and rebalancing transformations (settling push-up, removal, and pull-down requests) can be carried out concurrently and interspersed. This leads to the main theorem of this paper:

Theorem 3 *Let T be a Z-stratified search tree and assume that insertions, deletions and rebalancing transformations (possibly interspersed) are applied to T. Then the number of structural changes (pointer changes) needed to restore the balance condition (of being Z-stratified) is at most $O(i+d)$ where i is the number of insertions and d the number of deletions.*

We finally state the following obvious but important property for Z-stratified search trees and their update scheme.

Theorem 4 *Let T be a Z-stratified search tree and assume that insertions, deletions but no rebalancing transformations are applied to T. Assume further that the resulting tree T' has no internal nodes hanging below the bot-*

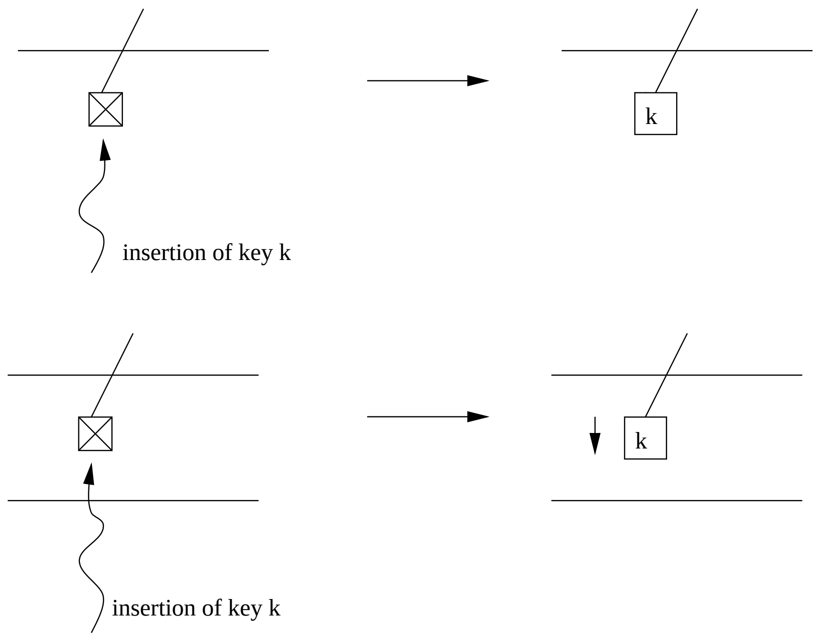

Figure 14: (Re-)insertion of a key in a leaf with a removal request

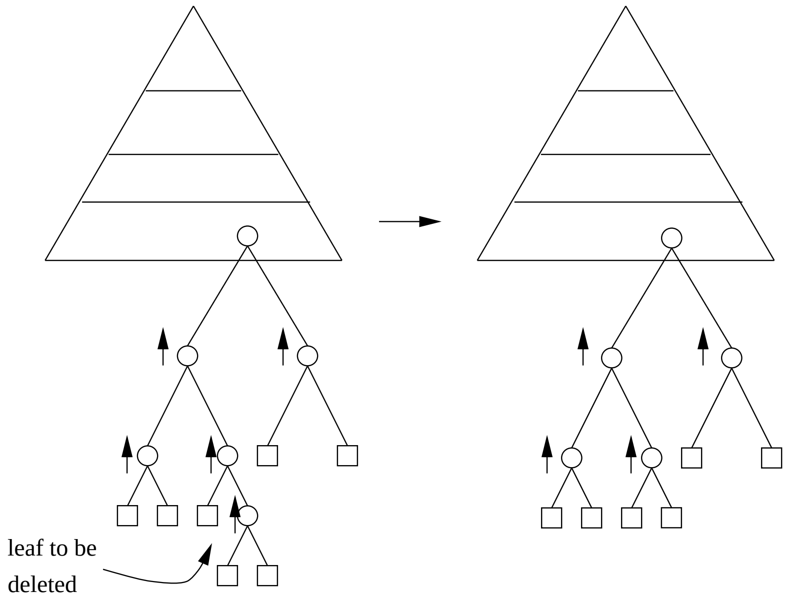

Figure 15: Deletion of a leaf not immediately below the bottommost stratum border

tommost stratum border of T and no leaves with (unsettled) removal requests. Then T' does not contain any unsettled pull-down or push-up requests.

Theorem 4 roughly states that no structural changes in a Z-stratified search tree are necessary if a sequence of updates has been carried out which has not changed the shape of the original tree. A possible sequence of those updates is a sequence of insertions of keys $k_1, \ldots, k_r$ which is followed by a sequence of deletions of $k_1, \ldots, k_r$ (or vice versa). Note, that the relaxed balancing scheme proposed in [11] generates even for such a sequence of updates a tree which is full of balance conflicts.

On the other hand, if a sequence of updates happens to add a full layer of stratum trees from Z below the bottommost layer of the originally given Z-stratified tree, our relaxed balancing scheme is not able to notice this and triggers push-up requests for all of the internal nodes hanging down below the bottommost stratum border of the original Z-stratified tree. This could be avoided only if some global information is accumulated during updates. Reference [18] shows how to do this for the case of height balanced trees. We conjecture that a similar idea is also applicable for Z-stratified trees. (Keep track of the stratum heights on all insertion paths!)

6 Conclusions

Our aim was to keep the relaxed balancing scheme conceptually as simple as possible. However, it should be clear that there is a lot of room for tuning the update algorithms. We give an example:

Push-up and pull-down requests both bubble up in a Z-stratified tree. In order to avoid conflicts, we have required that they are never applied simultaneously to the same stratum tree. On the other hand, in a case as the one shown by Figure 16 the combined transformation may save a lot of restructuring work.

As another example we propose an alternative way of handling combined removal and pull-down requests. Above we have allowed that individual nodes may obtain both (simultaneously) a pull-down and a removal request. All what we have required in order to assure a correct performance was that in such a case the pull-down request has to be handled first.

Clearly, we could have imposed a more restrictive requirement on how to settle a removal request: If a removal request applies to one of the leaves of

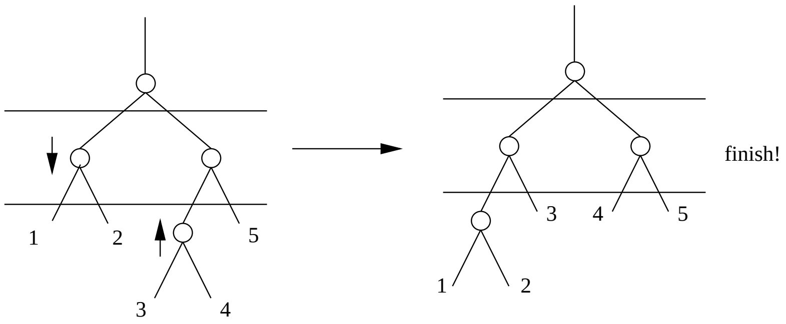

Figure 16: Combined settlement of a push-up and pull-down request

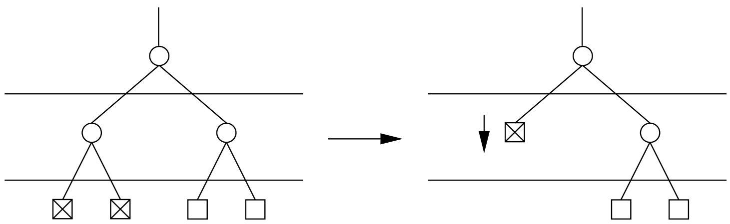

Figure 17: Handling accumulated deletions

the Z-stratum tree with only two leaves then we can first check whether the father of the leaf has an internal node as its brother and allow the removal of the leaf and the triggering of a pull- down request only if this is true. This would imply that a transformation as shown in Figure 17 is allowed whereas the one shown in Figure 18 is not.

We mention once again that our restriction to the class of Z-stratified trees with the very specific set Z of 4 stratum trees was motivated only by our aim to keep the presentation simple. The approach of uncoupling updates and rebalancing transformations can easily be extended to arbitrary classes of (A, Z)-stratified trees, [15], which are structurally closed under insertions and deletions.

Finally, we note that the basic idea of our relaxed balancing scheme (accumulation of insertions and deletions and settling them in an arbitrary order,

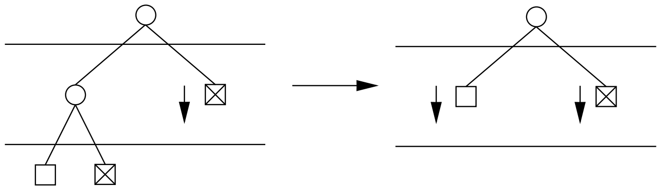

Figure 18: Illegal settlement of a removal request

using the same rebalancing operations as for standard balanced search trees) is applicable to other classes of search trees as well, even for those (like height balanced trees) which are not stratified. The idea is simply to let the tree grow below the original leaves as a random binary search tree and to mark an original leaf, if it has to be deleted. Then this set of unsettled updates can be handled just as a sequence of standard updates which are followed immediately by rebalancing transformations. All what is necessary is to assure that the postponed updates are settled top-down and no two restructurings interfere at the same nodes.

References

[1] G.M.Adel'son-Vels'kii and E.M.Landis, An algorithm for the organisation of information. *Soviet Math. Dokl.* **3** (1962), 1259–1262.

[2] R.Bayer, Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica* **1** (1972), 8–21.

[3] J.Boyar, R.Fagerberg and K.Larsen, Chromatic priority queues. Preprint 15, Department of Mathematics and Computer Science, Odense University, 1994.

[4] J.Boyar and K.Larsen, Efficient rebalancing of chromatic search trees. *Journal of Computer and System Sciences* **49** (1994), 667–682.

[5] C.S.Ellis, Concurrent search in AVL-trees. *IEEE Transactions on Computers* **C-29** (1980), 811–817.

[6] C.S.Ellis, Concurrent search and insertions in 2–3 trees. *Acta Informatica* **14** (1980), 63–86.

[7] L.J.Guibas and R.Sedgewick, A dichromatic framework for balanced trees. In: *Proc. 19th IEEE Symposium on Foundations of Computer Science*, 1978, pp. 8–21.

[8] A.Keller and G.Wiederhold, Concurrent use of B-trees with variable-length entries. *SIGMOD Record* **17**:2 (June 1988), 89–90.

[9] J.L.W.Kessels, On-the-fly optimization of data structures. *Comm. ACM* **26** (1983), 895–901.

[10] K.Larsen, AVL trees with relaxed balance. In: *Proc. 8th International Parallel Processing Symposium*, IEEE Computer Society Press, 1994, pp. 888–893.

[11] K.Larsen and R.Fagerberg, B-trees with relaxed balance. In: *Proc. 9th International Parallel Processing Symposium*, IEEE Computer Society Press, 1995, pp. 196–202.

[12] O.Nurmi and E.Soisalon-Soininen, Uncoupling updating and rebalancing in chromatic binary search trees. In: *Proc. 10th ACM Symposium on Principles of Database Systems*, 1991, pp. 192–198.

[13] O.Nurmi, E.Soisalon-Soininen and D.Wood, Concurrency control in database structures with relaxed balance. In: *Proc. 6th ACM Symposium on Principles of Database Systems*, 1987, pp. 170–176.

[14] H.J.Olivié, A new class of balanced search trees: half-balanced binary search trees. *R.A.I.R.O. Theoretical Informatics* **16**:1 (1982), 51–71.

[15] Th.Ottmann and D.Wood, Updating binary trees with constant linkage cost. In: *International Journal of Foundations of Computer Science*, **3**, 1992, pp. 479–501.

[16] Y.Sagiv, Concurrent operations on B*-trees with overtaking, *Journal of Computer and System Sciences* **33**:2 (1986), 275–296.

[17] D.Shasha and N.Goodman, Concurrent search structure algorithms. *ACM Transaction on Database Systems* **13** (March 1988), 53–90.

[18] E.Soisalon-Soininen and P.Widmayer, Relaxed balancing in binary search trees, Preprint ETH Zürich, 1995.