Alcom-FT Technical Report Series
ALCOMFT-TR-01-128

Search Trees with Relaxed Balance and Near-Optimal Height

Rolf Fagerberg [*†] Rune E. Jensen [‡] Kim S. Larsen [§†¶]

Abstract

We introduce the relaxed k-tree, a search tree with relaxed balance and a height bound, when in balance, of $(1+\varepsilon)\log_2 n + 1$, for any $\varepsilon > 0$. The rebalancing work is amortized $O(1/\varepsilon)$ per update. This is the first binary search tree with relaxed balance having a height bound better than $c \cdot \log_2 n$ for a fixed constant c. In all previous proposals, the constant is at least $1/\log_2 \phi > 1.44$, where ϕ is the golden ratio.

As a consequence, we can also define a standard (non-relaxed) k-tree with amortized constant rebalancing per update, which is an improvement over the original definition.

Search engines based on main-memory databases with strongly fluctuating workloads are possible applications for this line of work.

1 Introduction

The k-trees [7] differ from other binary search trees in that the height can be maintained arbitrarily close to the optimal $\lceil \log_2 n \rceil$ while the number of rebalancing operations carried out in response to an update remains $O(\log n)$. The price to be paid is that the size of each rebalancing operation (the number of nodes which must be inspected) grows as we approach $\lceil \log_2 n \rceil$. More precisely, one can show that to obtain height less than $(1+\varepsilon)\log_2 n + 1$, rebalancing operations have size $\Theta(1/\varepsilon)$ in [7].

Thus, using k-trees, a trade-off between search time and rebalancing time becomes an option; the more interesting direction being the scenario where updates are infrequent compared with searches. Under such circumstances, it

* BRICS, Department of Computer Science, University of Aarhus, Ny Munkegade, building 540, DK-8000 rhus C, Denmark. Email: rolf@brics.dk.

† Partially supported by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

‡ ALOC Bonnier A/S, Buchwaldsgade 35, DK-5000 Odense C, Denmark. Email: runej@aloc.dk.

§ Department of Mathematics and Computer Science, University of Southern Denmark, Main Campus: Odense University, Campusvej 55, DK-5230 Odense M, Denmark. Email: kslarsen@imada.sdu.dk.

¶ Supported in part by the Danish Natural Science Research Council (SNF).

may be beneficial to spend more time on the occasional updates in order to obtain shorter search paths.

Search engines pose a particular search/update problem. Searching is dominant, but when updates are made, they often come in bursts because keywords originate from large sites. Equipping search trees with relaxed balance has been proposed as a way of being able to adapt smoothly to this ever-changing scenario.

Relaxed balance is the term used for search trees where updating and rebalancing have been uncoupled, most often through a generalization of the basic structure. The uncoupling is achieved by allowing rebalancing after different updates to be postponed, broken down into small steps, and interleaved.

The challenge for the designers of these structures is to ensure and be able to prove efficient rebalancing in this more general structure. If that problem is overcome, then the benefit is the extra flexibility. During periods with heavy updating, rebalancing can be decreased or even turned completely off to allow a higher throughput of updates as well as searches. When the update burst is over, the structure can gradually be rebalanced again. Since a search engine is in constant use, it is important that this rebalancing is also carried out efficiently, i.e., using as few rebalancing operations as possible.

Besides search engines, the flexibility provided by relaxed balance may be an attractive option for any database application with strongly fluctuating work loads.

Relaxed balance has been studied in the context of AVL-trees [1] starting in [10] and with complexities matching the ones from the standard case in [6]. The height bound for AVL-trees in balance is $\frac{1}{\log_2 \phi} \log_2 n > 1.44 \log_2 n$. In the context of red-black trees [4], relaxed balance has been studied starting in [8, 9] with results gradually matching the standard case [11] in [3, 2, 5]. The height bound for red-black trees in balance is $2 \log_2 n$. A more thorough introduction to relaxed balance as well as a comprehensive list of references can be found in [5].

In this paper, we first develop an alternative definition of standard k-trees. The purpose of this is both to cut down on the number of special cases, and to pave the way for an improved complexity result. Based on this, a relaxed proposal is given, and complexity results are shown. The complexity results are in the form of upper bounds on the number of rebalancing operations which must be carried out in response to an update.

It is worth noting that the alternative definition of k-trees, which is the starting point for the relaxed definition, also gives rise to an improved complexity result for the standard case: in addition to the logarithmic worst-case bound, rebalancing can now be shown to use amortized constant time as well.

2 K-Trees

The k-trees of [7] are search trees where all external nodes have the same depth and all internal nodes are either unary or binary. The trees are leaf-oriented, meaning that the external nodes contain the keys of the elements stored, and the internal nodes contain *routers*, which are keys guiding the search. Binary internal nodes contain one key, unary internal nodes contain no keys. To avoid arbitrarily deep trees, restrictions are imposed on the number of unary nodes: on any level of the tree, the first k nodes to the right of a unary node should (if present) be binary. As this does not preclude a string of unary nodes starting at the root, it is also a requirement in [7] that the rightmost node on each level is binary.

It is intuitively clear that a larger value of k gives a lower density of unary nodes, which implies a smaller height for a given number n of stored keys. The price to be paid is an increased amount of rebalancing work per update—more precisely, one can show that with the proposal of [7] a height bound of $(1 + \Theta(1/k)) \log_2 n + 1$ can be maintained with $O(k \log n)$ work per update. Thus, k-trees offer a tradeoff between the height bound and the rebalancing time, and furthermore allow for height bounds of the form $c \cdot \log_2 n + 1$, where the constant c can be arbitrarily close to one.

While the possibility of such a tradeoff is a very interesting property, the k-trees of [7] also have some disadvantages. One is that, unlike red-black trees, for example, they do not have an amortized constant bound on the amount of rebalancing work per update. As a counterexample, consider a series of alternating deletions and insertions of the smallest key in a complete binary tree of height h, having $2^h - 1$ binary nodes. Since the tree does not contain any unary nodes, it is a valid k-tree for any k, and it is easy to verify that the rebalancing operations described in [7] will propagate all the way to the root after each update. Another disadvantage is the lack of left-right symmetry in the definition of k-trees in [7], forcing operations at the rightmost path in the tree to be special cases. This approximately doubles the number of operations compared with the number of essentially different operations.

We therefore propose an alternative definition of k-trees. This definition will allow us to add relaxed balance using a relatively simple set of rebalancing operations, for which we can prove an amortized complexity of $O(1)$ per update. Additionally, this enables us to define a new non-relaxed k-tree with the same complexity, simply by deciding to rebalance completely after each update. This is an improvement of the result from [7].

Our basic change is in the way the density of unary nodes is kept low. On each level in the tree, except the topmost, we divide the nodes into *groups* of $\Theta(k)$ neighboring nodes. Thus, a group is simply a contiguous segment of a given level. The groups are implemented by marking the leftmost node in each group, using one bit. Furthermore, in each group, we allow *two* unary nodes, contrary to the original proposal [7] which considers unary nodes one by one. Intuitively, this is what gives the amortized constant rebalancing per update. The top of the tree is managed differently, as the levels are too small to contain a group.

Definition 1 For any integer $k \geq 2$, a *symmetric k-tree* is a tree containing unary and binary nodes, where all external nodes have the same depth. The topmost $1+\lceil\log k\rceil$ levels consist of binary nodes only. In level number $2+\lceil\log k\rceil$ from the top, there is at least one binary node. On the rest of the levels in the tree, the internal nodes are divided into groups of neighboring nodes. Each group contains at least $2k$ nodes and at most $4k$ nodes. In each group, at most two of the nodes are unary. □

We call level number $2+\lceil\log k\rceil$ the *buffer level.* For the number S of nodes in the buffer level, we have $2k \leq S = 2^{\lceil\log k\rceil+1} < 4k$.

The tree is turned into a search tree by storing elements in the external nodes and routers in the binary internal nodes in accordance with the usual in-order ordering. Searching proceeds as in any binary search tree, except that unary nodes (which contain no keys) are just passed through.

To add relaxed balance, we must allow insertions and deletions to proceed without immediate rebalancing, and therefore must relax the structural constraints of Definition 1. To achieve this, we allow nodes of degree larger than two, and allow an arbitrary number of unary nodes in a group. To keep the actual trees binary, we use the standard method [4] of representing i-ary nodes by binary subtrees with $i-1$ nodes, indicating the root of each subtree by one bit of information, traditionally termed a red/black color.

We define the *black depth* of a node as the number of black nodes (including itself, if black) on the path to the root. The *black level* number i consists of all black nodes having the black depth i. Note that for any node, the number of black nodes below it on a path to an external node is the same for all such paths. We call this number the *black height* of the node.

Definition 2 For any integer $k \geq 2$, a *relaxed k-tree* is a tree containing unary and binary nodes, where nodes are colored either black or red. The root, the unary nodes and the external nodes are always black. All external nodes in the tree have the same black depth. In the topmost $1+\lceil\log k\rceil$ black levels there are no unary nodes, and no node has a red child. In black level number $2+\lceil\log k\rceil$, there is at least one binary node. On the rest of the black levels in the tree, the internal nodes are divided into groups of neighboring nodes, with each group containing at least $2k$ nodes and at most $4k$ nodes. □

A relaxed k-tree is a standard (symmetric) k-tree if all nodes are black, and no group contains more than two unary nodes. It turns out that in our relaxed search trees, we also need to allow *empty* external nodes, i.e., external nodes with no elements. Later in this paper, we give a set of rebalancing operations which can turn a relaxed k-tree with empty external nodes into a standard k-tree without empty external nodes, and we give bounds on the number of operations needed for this.

3 Height Bound

By the *height* of a tree we mean the maximal number of edges on any path from the root to an external node. We now show that the height of symmetric k-trees is just as good as that of the original version in [7].

Theorem 1 The height of a symmetric k-tree with n external nodes is bounded by $\log_\alpha n + 1$, where $\alpha = 2 - 1/k$.

Proof On any level, except the buffer level, at most two out of each $2k$ nodes are unary. It follows that the number of nodes for each new level, except the buffer level, increases at least by a factor of $2(1 - 1/k) + 1/k = 2 - 1/k = \alpha$. For the buffer level, the number of nodes does not decrease. Hence, a tree of height h contains at least α^{h-1} external nodes. $\square$

Using the identity $\log_\alpha(x) = \log_2(x)/\log_2(\alpha)$, this height bound may be stated in a more standard way as $c \cdot \log_2 n + 1$. Examples of values of c attainable by varying k are shown in Table 1.

k	2	3	4	5	6	7	8	9	10	20	50	100
c	1.71	1.36	1.24	1.18	1.14	1.12	1.10	1.09	1.08	1.038	1.015	1.007

Table 1: Corresponding values of k and c.

The asymptotic relationship between k and c is as follows:

Corollary 1 In symmetric k-trees, the height is bounded by

$$(1 + \Theta(1/k)) \log_2 n + 1.$$

Proof This follows from Theorem 1 by the identity $\log_\alpha(x) = \log_2(x)/\log_2(\alpha)$ and the first order approximations

$$\begin{aligned} \log_2(1+\varepsilon) &= 0 + \varepsilon/\ln 2 + O(\varepsilon^2), \\ 1/(1-\varepsilon) &= 1 + \varepsilon + O(\varepsilon^2). \end{aligned}$$

$\square$

4 Operations

As mentioned above, a *search* operation proceeds as in any binary search tree, except that unary nodes are just passed through.

An *insert* operation starts by a search which ends in an external node v. If v is empty, the new element is placed there. Otherwise, a new external node containing the new element is made. In that case, if the parent of v is unary, it is made binary, and the new external node becomes a child of the binary node. The key of the new element is inserted as router in the binary node. If the parent of v is binary, a new red binary node is inserted below it, having v and the new external node as children and the key of the new element as router.

A *delete* operation first searches for the external node v, containing the element to be deleted. If the parent of v is unary, the leaf becomes empty. Otherwise, v is removed, and the binary parent is made unary (discarding the router in it), in case it is black, and is removed completely, in case it is red.

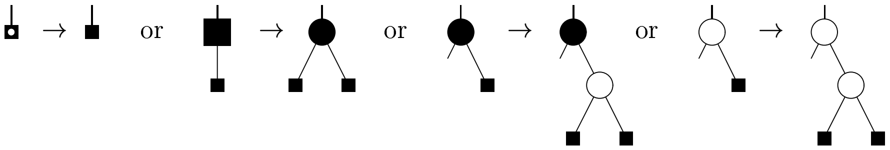

Figure 1: The insert operation.

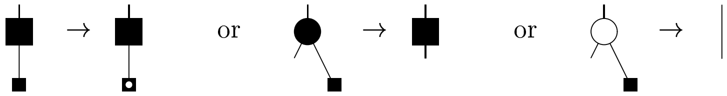

Figure 2: The delete operation.

Fundamental to the *rebalancing operations* on k-trees is the observation [7] that the position of unary nodes may be moved horizontally in the tree by shifting subtrees. In Fig. 3, the position of the unary node on the left is moved six nodes to the right. Letters denote subtrees.

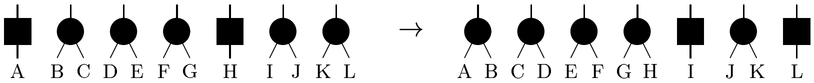

Figure 3: The slide operation.

We call this operation a *slide operation*, and we use it to move the positions of unary nodes horizontally among the black nodes of the same black depth.

Note that for a slide involving i neighboring nodes, it is necessary to redistribute some keys to keep the in-ordering of the keys in the tree. The keys in question are contained in the binary nodes among the nodes involved in the slide, as well as in the least common ancestors of each of the consecutive pair of nodes involved in the slide. This is at most $2i - 1$ keys in total. Excluding the time for locating these least common ancestors, the slide can be performed in $O(i)$ time. We address the question of the time for locating these common ancestors later. In [7], this question is not considered at all.

A relaxed k-tree may contain two kinds of structural problems which keep it from being a standard (symmetric) k-tree: *red binary nodes* and *groups containing more than two unary nodes.* Additionally, it may contain *empty external nodes.* We now describe the set of rebalancing operations which we use to remove these three problem types.

We only deal with red binary nodes having a black parent. If the parent of the red node is unary, we use a *contract* operation, which merges the node and the parent into a black binary node.

If the parent is binary, we first check if there is a unary node in its group. If so, we use the slide operation to make the parent unary, and then perform a contract operation.

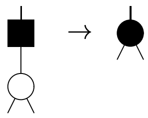

Figure 4: The contract operation.

If the parent is binary, and there is no unary node in its group, we apply the following operation, which makes the parent red and the node itself black. Furthermore, if the sibling is red, the operation makes it black. Otherwise, the operation inserts a unary node above the sibling:

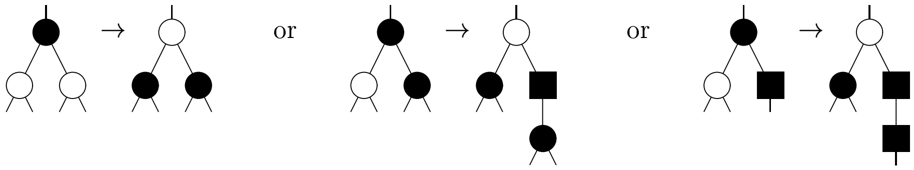

Figure 5: The split operation.

We call this operation a *split* operation, since it corresponds to the splitting of an i-ary node ($i \geq 3$) in a formulation with multi-way nodes instead of red/black colors.

For a group containing more than two unary nodes, we use the *merge* operation from Fig. 6 which merges two unary siblings into a black binary node. If the parent is black, it is converted to a unary node. If it is red, it is removed.

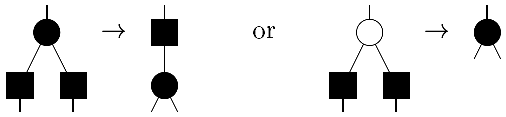

Figure 6: The merge operation.

Note that by using the slide operation within a group, we can decide freely which nodes within the group should be the unary ones. Note also that since the group contains at least four nodes (as $k \geq 2$), there will be at least two neighboring nodes which have parents belonging to the same group on the level above. Using the slide operation within that group, we can ensure that if it contains any binary node at all, it will be the parent of the two neighboring nodes. Thus, only if this group on the level above does not contain any binary nodes will we not be able to perform a merge operation on a group containing more than two unary nodes.

We note that split and merge operations will make the number of nodes in the affected group increase, respectively decrease, by one. To keep the group sizes within the required bounds, we use a policy similar to that of B-trees, i.e., a group which reaches a size of $4k + 1$ nodes is split evenly in two, and a group which reaches a size of $2k-1$ nodes will either borrow a node from a neighboring group, or, if this is not possible because no neighboring group of more than $2k$ nodes exist, will be merged with a neighboring group. This entails simply setting, removing, or moving a group border, i.e., a bit in a node. When borrowing a node from a neighboring group containing fewer than two unary nodes, we ensure that the borrowed node is binary, by first using a slide operation, if necessary. The maintenance of group sizes is performed as part of the split and merge operations.

Regarding empty external nodes, we note that these will always be children of black nodes; they are created that way, and no operation changes this. If the parent of the empty external node is binary, we remove the external node and make the parent unary. If the parent is unary, but a binary node exist in its group, we use the slide operation to make the parent binary, and then proceed as above. Only if the parent's group does not contain any binary nodes will we not be able to remove the empty external node.

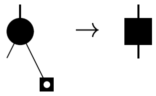

Figure 7: The removal of an empty external node.

For problems immediately below the buffer level, special *root operations* apply. If the problem is a red node, we use the contract operation, and as usual use a slide to make the parent of the red node unary, if necessary. However, if no unary node exists in the buffer level, this is not possible. In that case, a new buffer level consisting entirely of unary nodes is inserted above the previous buffer level. We then use the split operation to move the red node past the previous buffer level, and then use a contract operation on the new buffer level to remove the red node. Note that this maintains the invariant that the buffer level should contain at least one binary node.

Conversely, if a merge operation removes the last binary node of the buffer level, we first check if any of the unary nodes in the buffer level has a red child. If so, we perform a contract operation on that node. If this is not possible, we remove the nodes in the current buffer level (these are all unary), and let the black level below be the new buffer level. As the merge operation introduced a binary node on this level, the invariant that the buffer level should contain at least one binary node is maintained.

Note that the black height of the tree can only change via a root operation.

It is clear from inspection that the update and rebalancing operations do not violate the invariant that all external nodes have the same black depth. The set of rebalancing operations is also complete in the following sense:

Lemma 1 On any relaxed k-tree which is not a standard symmetric k-tree without empty external nodes, at least one of the rebalancing operations can be applied on the tree.

Proof A rebalancing operation can always be performed at the topmost red node and at the topmost group which contains more than two unary nodes. If no group with more than two unary nodes exists, an empty external node can always be removed. □

5 Complexity Analysis

The number of rebalancing operations per update is amortized constant:

Theorem 2 During a sequence of i insertions and d deletions performed on an initially empty relaxed k-tree, at most $6i + 4d$ rebalancing operations can be performed before the tree is in balance.

Proof The number of removals of empty external nodes clearly cannot exceed d. To bound the rest of the operations, we define a suitable potential function on any relaxed k-tree T. Let the unary potential of a group be $|u - 1|$, where u denotes the number of unary nodes in the group. Denote by $\Phi_1(T)$ the sum of the unary potential of all groups in T (the buffer level does not constitute at group, and neither do the levels above it). Denote by $\Phi_2(T)$ the number of red nodes in T, by $\Phi_3(T)$ the number of groups in T containing $2k$ nodes, and by $\Phi_4(T)$ the number of groups in T containing $4k$ nodes. Define the potential function $\Phi(T)$ by

$$\Phi(T) = 3 \cdot \Phi_1(T) + 6 \cdot \Phi_2(T) + 1 \cdot \Phi_3(T) + 2 \cdot \Phi_4(T).$$

By a lengthy inspection it can be verified that all rebalancing operations, including any necessary group splitting, merging or sharing, will decrease $\Phi(T)$ by at least one. We analyze one case to give the flavor of the argument, and leave the rest to the reader. Consider the case of the split operation depicted in the middle of Fig. 5. As the group of the top node in the operation does not contain any unary nodes before the operation, the added unary node after the operation reduces $\Phi_1(T)$ by one. The number of red nodes do not change, so neither does $\Phi_3(T)$. The group size increases by one, hence $\Phi_4(T)$ may grow by one, or the group may have to be split (if the size of the group raises to $4k+1$). In the latter case, the sizes of the new groups will be $2k$ and $2k + 1$, which makes $\Phi_3(T)$ grow by one while reducing $\Phi_4(T)$ by one, and the new groups will contain zero and one unary node, respectively, which will make $\Phi_1(T)$ grow by one (for a total change of zero). By the weights of $\Phi_1(T), \ldots, \Phi_4(T)$ in $\Phi(T)$, this gives a reduction of $\Phi(T)$ of at least one in all cases.

By inspection, it can also be seen that each insert operation increases $\Phi(T)$ by at most six, and that each delete operation either increases $\Phi(T)$ by at most three, or does not change $\Phi(T)$, but introduces an empty external node, the removal of which later increases $\Phi(T)$ by at most three. As $\Phi(T)$ is zero for the empty tree and is never negative, the result follows. □

The proof above may be refined to give the following:

Theorem 3 During a sequence of i insertions and d deletions performed on an initially empty relaxed k-tree, at most $O((i+d)(6/7)^h)$ rebalancing operations can be performed at black height h before the tree is in balance.

Proof The idea of the proof is to define a potential functions $\Phi^h(T)$ for $h = 0, 1, 2, 3, \ldots$, where $\Phi^h(T)$ is defined as $\Phi(T)$ in the proof above, except that it only counts potential residing at black height h. By inspection, it can be verified that a rebalancing operation at black height h always decreases $\Phi^h(T)$ by some amount $\Delta \geq 1$, and that, while it may increase the value of $\Phi^{h+1}(T)$, it will never do so by more than $6\Delta/7$. As the $\Phi^h(T)$'s are initially zero and are never negative, this implies the statement in the theorem. □

Theorem 4 If n updates are made on a balanced relaxed k-tree containing N keys, then at most $O(n \log(N+n))$ rebalancing operations can be made before the tree is again balanced.

Proof The problems in a non-balanced tree consist of red nodes and excess unary nodes in groups. Assigning to each such problem a height equal to the black height of its corresponding node, it can be verified that each rebalancing operation which does not reduce the number of problems will increase the height of some problem by one, and that no rebalancing operation will decrease the height of any problem.

Problems arise during updates at black height zero, and each update introduce at most one problem. Thus, if n updates are performed on an initially balanced tree T, the number of update operations cannot exceed n times the maximum black height of T since the start of the sequence of updates.

To bound this maximum height, we recall that the black height of the tree can only increase during root operations. Specifically, if the black height of the tree reaches some value h, then there has been a root operation at black height $h-1$. It is easily verified that the value of $\Phi(T)$ for a balanced tree T is linear in the number of keys N in the tree. During the n updates, this value may only grow by $O(n)$, by the analysis in the proof of Theorem 2. By an argument similar to that in the proof of Theorem 3, the maximum black height since the start of the sequence of updates is $O(\log_{7/6}(N+n))$, which proves the theorem. □

6 Comments on Implementation

In the previous section, we have been concerned with the number of operations which have to be carried out, and we have discussed configurations in the tree at a fairly abstract level. In order to carry out each operation efficiently, it is necessary to be able to find other nodes at the same level, to find least common ancestors, and to locate problems in the tree.

First, we note that by maintaining parent pointers and level links between the black nodes sharing the same black height, all rebalancing operation can be performed in $O(k)$ time, when not counting the time to find the necessary least common ancestors during a slide operation. The same is true for a group resizing operation.

We now discuss how to find the necessary least common ancestor (LCA) of a neighboring pair of black nodes participating in a slide operation.

One approach is *heuristic*: simply search upwards for each of these LCAs. The worst case time for this is poor (the search may take time proportional to the height of the tree for each LCA), but should be good on average in the following sense: If on some level i in a complete binary tree we consider k neighboring nodes, then the LCAs of these k nodes will reside in at most two subtrees with roots at most $\lceil \log(k) \rceil$ levels above i, *except* that one of the LCAs may reside higher (it could be the root of the entire tree). If by δ we denote the difference between $i - \lceil \log(k) \rceil$ and the level of this singular LCA, then it is easily verified that the expected value of δ over all possible start positions of the k neighboring nodes on level i is $O(1)$. As the parts of the two subtrees residing above level i may be traversed in $O(k)$ time, the time for a randomly placed slide involving k nodes is expected $O(k)$ in a binary tree. As a k-tree is structurally close to a binary tree (especially for large k), we therefore believe that the time for finding the LCAs during a slide is not likely to be a problem, unless during the use of a relaxed k-tree we allow the tree to become very unbalanced before rebalancing again.

Another approach is to *maintain explicit links* from every black node to the two LCAs between itself and its two black neighbors, allowing each LCA to be found in constant time during a slide operation. These links then have to be updated for the black nodes on the leftmost and rightmost path on the subtrees exchanged between neighboring black nodes during a slide. Assuming that the black nodes on such a left- or rightmost path can be accessed in constant time per node, this gives a time for a slide involving k neighboring nodes which is proportional to k times the black height at which the slide takes place. However, as $\sum_{h=1}^{\infty} h(6/7)^h = O(1)$, Theorem 3 implies that the amortized rebalancing work is still $O(k)$ per update.

So, we must be able to traverse only the black nodes on the left- and rightmost paths mentioned above. For every black node, we keep all its immediate red descendants (those forming a single node in a formulation with multi-way nodes instead of red/black colors) in a doubly linked list. The list is ordered (the list may be seen as adding in-order links to all red connected components of the tree), and the front and rear of the list is pointed to from the black node rooting the red connected component. Using these front and rear pointers, it is now possible to jump from black to black level during a traversal of right- and leftmost paths, as assumed above. Furthermore, it can be verified that these list can be maintained during update and rebalancing operations, including slides.

Finally, locating problems in the tree is complicated by the main feature of relaxed balance, namely that the rebalancing is uncoupled from the updating. Hence, an update operation simply ignores any problem which arises as a consequence of the update. To be able to return to these problems later, a *problem queue* can be maintained. For problems discovered during an update, a pointer is stored in the queue. One pointer per group suffices. Since update and rebalancing operations may remove other problems, it is convenient to be able to remove problems from the queue. Thus, each group must have a back-pointer into the problem queue. Rebalancing operations start by dequeuing a pointer, which is then followed, and the appropriate rebalancing operation is performed. Rebalancing operations should also insert new pointers when they move problems upwards in the tree, unless the receiving group is already in the queue. Note that when a problem is added to the tree, it can be verified in $O(k)$ time whether the affected group has a problem already.

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] 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.

[3] Joan F. Boyar and Kim S. Larsen. Efficient Rebalancing of Chromatic Search Trees. *Journal of Computer and System Sciences*, 49(3):667–682, 1994.

[4] 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.

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

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

[7] H. A. Maurer, Th. Ottmann, and H.-W. Six. Implementing Dictionaries using Binary Trees of Very Small Height. *Information Processing Letters*, 5(1):11–14, 1976.

[8] 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.

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

[10] 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.

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