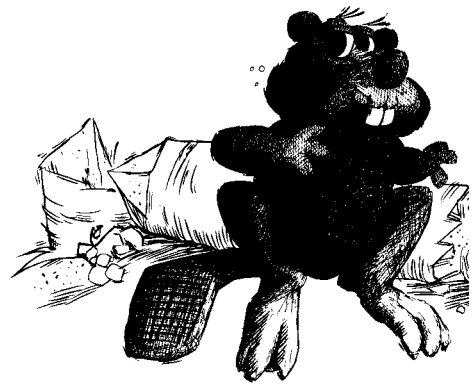
UNIVERSITY OF WATERLOO
UNIVERSITY OF WATERLOO
UNIVERSITY OF WATERLOO
COMPUTER SCIENCE DEPARTMENT
COMPUTER SCIENCE DEPARTMENT
COMPUTER SCIENCE DEPARTMENT



AVL Trees
On-The-Fly Restructuring and Concurrency

Eljas Soisalon-Soininen
Derick Wood

Data Structuring Group
CS-86-52

October, 1986

AVL Trees, On-The-Fly Restructuring, and Concurrency*

Eljas Soisalon-Soininen † Derick Wood ‡

October 19, 1986

Abstract

There is, presently, a renewed interest in in-core databases and hence, in techniques to organize them for a concurrent environment. In this paper we reconsider an old method for supporting insertions, deletions, and member queries — AVL trees. The novelty of our approach is that we carry out incremental or on-the-fly restructuring as a background maintenance task, rather than as a foreground user task. For this purpose we add tags to nodes in an AVL tree to indicate where local modifications need to be made. Since restructuring is not carried out immediately we obtain what we call *relaxed AVL trees*. We present the restructuring algorithms for relaxed AVL trees and, also, a concurrent solution based on these methods.

1 Introduction

There has been a recent resurge of interest in in-core databases and, hence, also in in-core search structures for the abstract data type DICTIONARY, see [12,13]. There is a difference between past and present interest however. In the past, search structures were studied predominantly in sequential environments, while present interest is in concurrent environments. External search structures have been well studied in this regard (see [3], the survey of [11], and the more recent papers of [4,15]), since databases have been

*This research was supported under a Natural Sciences and Engineering Research Council of Canada Research Grant No. A-5692.

†Department of Computer Science, University of Helsinki, Tukholmankatu 2, SF-02500 Helsinki, Finland

‡Data Structuring Group, Department of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada

assumed to be stored externally. By comparison there has been little investigation of internal search structures in a concurrent environment; see [6,7,8,10].

We contribute to the study of internal search structures in a concurrent environment by extending a method due to Kessels[8] — for AVL trees — to include deletions, that is the three DICTIONARY operations of MEMBER, INSERT, and DELETE. Kessels' approach is to decouple the search and restructuring processes in an update process and to implement restructuring as a simple, local restructuring process, that is, as straight-line code. In Kessels' solution [8] a binary tag is attached to each node in order to indicate the possible need for restructuring. When a node is inserted a negative tag (in our terminology) is attached to it. Using a modified definition of height called *relaxed height*, an insertion into an AVL tree results in what we call a *relaxed AVL tree*. Moreover, further insertions also leave a relaxed AVL tree. A separate restructuring process searches for negative tags, cancelling them, moving them upward in the tree possibly in conjunction with rotations, or leaving them alone. A negative tag in the root can always be cancelled and thus a finite number of restructuring steps always transforms a relaxed AVL tree into an AVL tree. This approach is called on-the-fly restructuring in analogy to on-the-fly compacting [2] and on-the-fly garbage collection [5]. It allows restructuring to be carried out as a background maintenance or housekeeping task. The local restructuring process accesses a small, fixed number of nodes that is independent of the size of the tree and it also maintains the consistency of the DICTIONARY. In contradistinction, the usual restructuring process accesses a number of nodes bounded by the height of the tree while maintaining consistency. The implication of these differences is that in a concurrent environment a higher degree of concurrency can be achieved without losing consistency.

To accomplish this extension we are forced to turn to the external-search variant of AVL trees, that is, records are stored in external nodes and internal nodes contain only separating keys. As the reader will see we actually use both positive and negative tags (at least, for ease of exposition). Essentially, when an external node is deleted, its sibling is given a tag one greater than its previous value. The separate restructuring process now searches for any non-zero tag and operates in different ways depending on local combinations of the tags. Because the negative and positive tags may interact it is not immediately clear that the insertion and deletion operations can be implemented in this way. Fortunately, however, we are able to define a background process for restructuring, which, after a finite number of steps, transforms any relaxed AVL tree into one that has only zero-valued tags and is, therefore, AVL.

In Section 2 we define external search AVL trees, while relaxed AVL

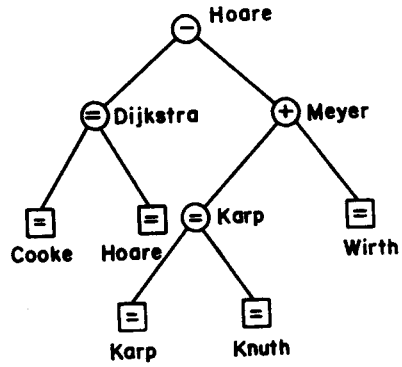


Figure 1: An example tree.

trees are defined in Section 3. We turn our attention to their updating in a concurrent environment in Section 4 and provide one possible locking protocol in Section 5. Let us stress that the main goal of this paper is to demonstrate that decoupling and local restructuring of AVL trees can be accomplished.

2 External search AVL trees

We assume the reader is familiar with the definition of (extended) binary trees, see [9], the associated genealogical terminology, and the usual notion of an (internal) search binary tree or binary search tree.

Recall that in an extended binary tree there are both *internal* and *external* nodes; internal nodes have two children and external nodes have no children. We represent internal nodes with circles and external nodes with boxes. The *height* of a node in a binary tree is zero if it is an external node and is one more than the maximum of the heights of its two children, otherwise. *An AVL tree*, see [1,9], is a binary tree in which the heights of each pair of sibling nodes differ by at most one. We usually associate with each node in an AVL tree a *balance factor*, which is defined as zero if it is an external node and the height of its left child minus the height of its right child, otherwise. Clearly balance factors may take only the values -1, 0, and $+1$, which we denote by $-, =$, and $+$ in diagrams, see Figure 1.

An extended binary tree with n external nodes is an *external search binary tree* for keys $K_1, K_2, \ldots, K_n$, from some totally-ordered universe of keys, where $K_1 < K_2 < \cdots < K_n$, if:

1. K_1 is associated with the first external node from the left, K_2 with the second external node,..., and K_n with the nth external node.

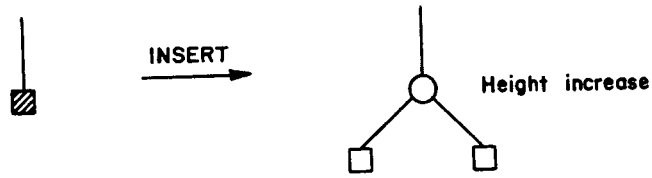


Figure 2: Addition of a key.

2. Each internal node has associated with it a key, from the universe of keys, that separates the keys in its left subtree from the keys in its right subtree. More precisely, the *separating key*, as it is called, is greater than or equal to the keys in the node's left subtree and less than the keys in the node's right subtree

Note that separating keys need not appear at external nodes. For example, the tree of Figure 1 is an external search tree for the keys Cooke, Hoare, Karp, Knuth, and Wirth. The separating keys are Dijkstra, Hoare, Karp, and Meyer. The advantage of this definition of separating key is that deletion of keys does not require the removal of a corresponding separating key. For example if we delete the key Hoare we need not remove the separating key Hoare from the root. This is a significant advantage over internal search binary trees where the deletion of a key can effect a node arbitrarily high in the tree.

Finally, an *external search AVL tree* is an AVL tree which is also an external search binary tree. From hereon in whenever we speak of an AVL tree we mean an external search AVL tree.

3 Relaxed AVL trees

In a concurrent environment it is assumed that many DELETE, INSERT, and MEMBER operations may be accessing a tree at any one time. Since these operations can interfere with each other, we introduce some form of concurrency control to ensure orderly access to nodes and their values. In Section 5 we use a straightforward locking protocol for this purpose, see also [15]. It is important to realize that even maintaining unrestricted binary search trees in a concurrent environment is a non-trivial task, see [10]. Because AVL trees may require structural changes after updating, maintaining them is even more difficult. The only results specific to AVL trees are those of [6,7,8]. In this and the following section we concentrate exclusively on the structural maintenance of AVL trees, we return to the concurrency control problem in Section 5. The basic idea behind our approach is caught by Figure 2 and Figure 3. The addition of a key increases the height of a

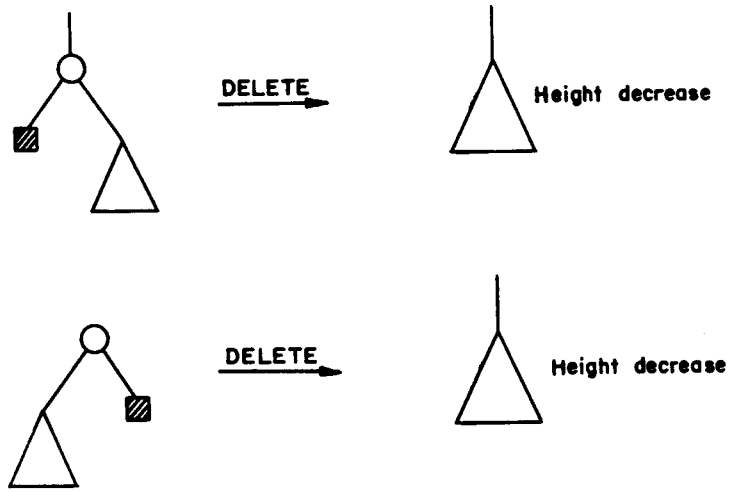


Figure 3: Removal of a key.

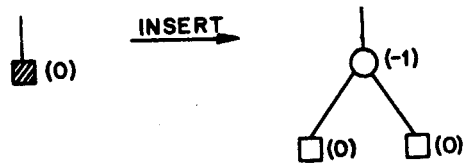


Figure 4: Addition forces a negative tag.

subtree and the removal of a key decreases the height of a subtree. In both cases the balance factors of some nodes are, at best, modified and, at worst, outside the range $-1..+1$. To avoid these difficulties, at least temporarily, on insertion we associate a negative tag with the new internal node, see Figure 4. Apart from a change in nomenclature this idea is to be found in [8]. The effect of the tag is to make the height of the new node zero as before, therefore the tree remains a valid AVL tree *if the tag is taken into account.* For this purpose, we define a modified notion of height, namely *relaxed height*, which takes tags into account. The *relaxed height* (or *rht*) of a node u, in a tagged binary tree is defined as:

$$rht(u) = \begin{cases} 0, \\ \quad \text{if } u \text{ is in an external node;} \\ 1 + tag(u) + \max(rht(left(u)), rht(right(u))), \\ \quad \text{otherwise} \end{cases}$$

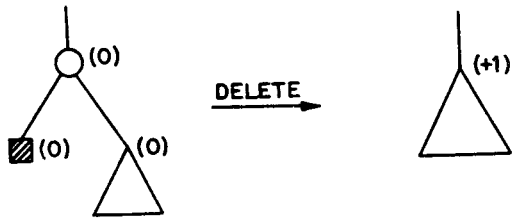


Figure 5: Removal forces a positive tag.

where $tag(u)$ is the tag value associated with node u. The *relaxed balance factor* of a node u, denoted by *rbf(u)*, is defined as:

$$rbf(u) = rht(left(u)) - rht(right(u))$$

A *relaxed AVL tree* is a tagged binary tree in which

$$-1 \leq rbf(u) \leq +1$$

for all nodes u in the tree.

In a concurrent environment multiple insertions can cause many negative tags to be associated with internal nodes. However each new negative tag is associated with exactly one new internal node, therefore, initially, no node has more than one negative tag associated with it. As we shall prove below this situation never changes.

But what have these tags bought us? First, each insertion process can be split into two separate processes — a search process and a restructuring process — because the tag indicates where an addition has taken place. This also means that restructuring need not be carried out immediately — indeed we will assume a background maintenance task takes over this duty. Second, as we shall see below, the restructuring process can be broken down into a succession of simpler and similar tag processes; each such process consists solely of straight-line code.

Deletion is, as almost always, a more difficult process to deal with. Since we associate negative tags or a debit with the addition of a key, we should, clearly, associate positive tags or a credit with the removal of a key.

Because we have removed the nodes that cause a local height decrease, see Figure 3 we should associate a positive tag with the root of the remaining sibling subtree; see Figure 5, for example. However because nodes are removed rather than added we cannot guarantee that each such node will have at most one positive tag.

- External nodes may accumulate a tag value greater than one. In Figure 6 this is caused by the deletion of B followed by the deletion of A.

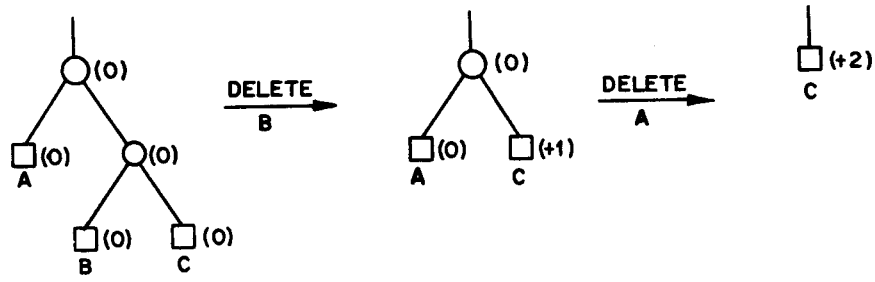


Figure 6: Tag value greater than one.

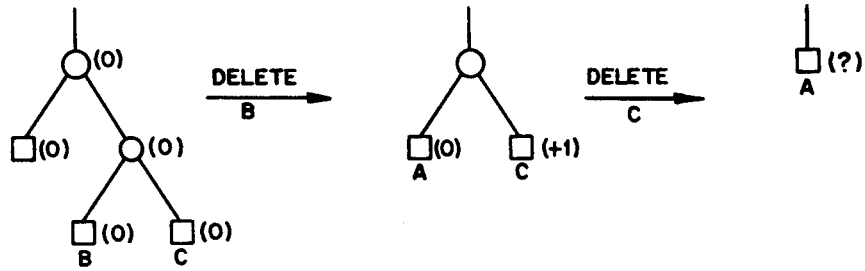


Figure 7: An external node can obtain a positive tag.

- An external node that is to be removed may have a positive tag associated with it. In Figure 7, after the deletion of B, C has a tag value of one. If C is deleted, what tag should be associated with the remaining node for A? As we will see, as a result of restructuring, internal nodes can also accumulate positive tags, even if the sibling of a deleted node is an internal node.

- A sibling, if it is internal, can have a negative tag associated with it.

- The parent of an external node that is to be removed may itself have a nonzero tag associated with it.

- The balance factor of the parent effects the result. In Figure 8 we display the three possibilities in a simple setting. Note that in all three cases the new value of the tag of the sibling is independent of the tag value of the deleted node.

In other words, the removal operation is not as simple as the addition operation. Fortunately, despite these difficulties, a removal can be effected. In Figure 9 we display all possible situations that can occur and how we deal with them. Furthermore, because an external node may have a positive tag, insertion needs to be modified slightly; see Figure 10. Thus we have attained

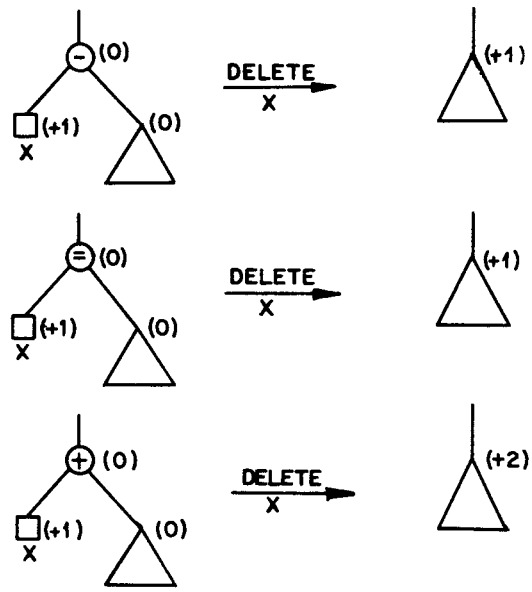


Figure 8: The balance factor influences the effect of a deletion.

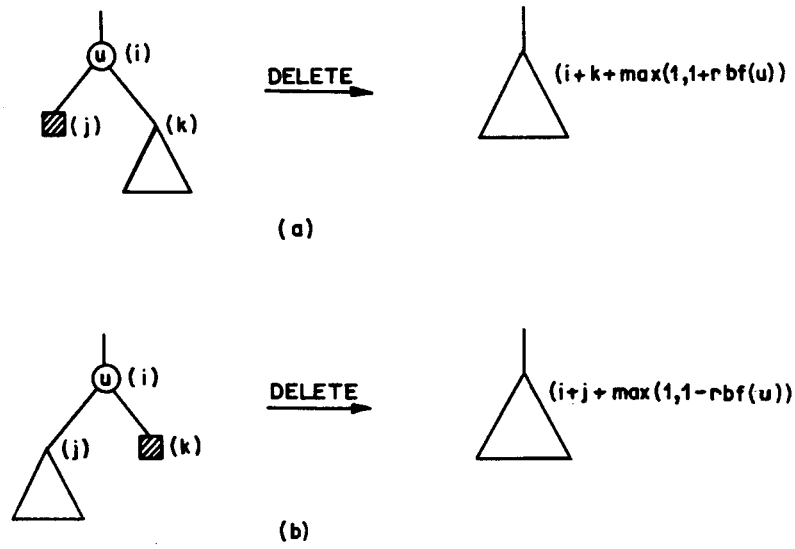


Figure 9: Removal: The total picture.

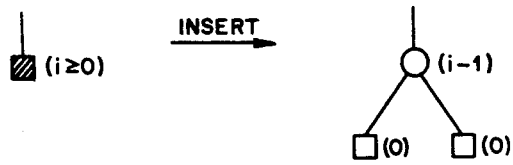


Figure 10: Modified addition.

our goal of separating deletion into two separate processes: the search (and actual removal of a value) and the restructuring process.

In the next section we introduce a background maintenance task that modifies tags in a relaxed AVL tree.

4 On-the-Fly Restructuring

We posit the existence of a background maintenance task that visits internal nodes of the tree according to some discipline and initiates a restructuring process at each such node. The restructuring process examines the node and its two children to determine if any action should be taken.

- If action should be taken it locks the necessary nodes, carries out the action, and terminates.
- If no action should be taken the process terminates immediately.

Our only concern here is the restructuring process — will each action lead to a tree that is closer to being an AVL tree? First, we describe the restructuring process and, second, we prove that each application produces a tree closer to an AVL tree — in other words we have convergence.

The process is designed so that each invocation of the process will either decrease the total sum of the absolute values of the tags in the tree, or this sum remains the same but a node loses some of its absolute tag value in favor of a node outside its subtree. Formally, we define $OUTSIDE(u)$ to be the number of nodes not in the subtree rooted at node u. Then, for a node u,

$$VALUE(u) = abs(tag(u)) \cdot OUTSIDE(u)$$

where $abs(tag(u))$ denotes the absolute value of the tag attached to u. For a tree T, the ***tag-value sum***, denoted by $VALUE(T)$, is defined as:

$$VALUE(T) = \sum_{u \in T} VALUE(u)$$

Note that $VALUE(T) = 0$ if and only if T is an AVL tree.

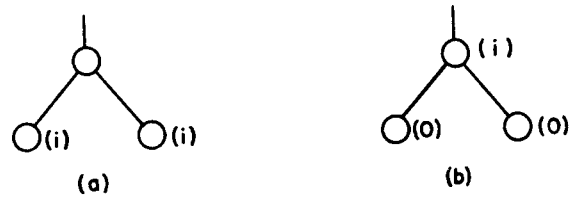


Figure 11: Case 1: Moving tags upward.

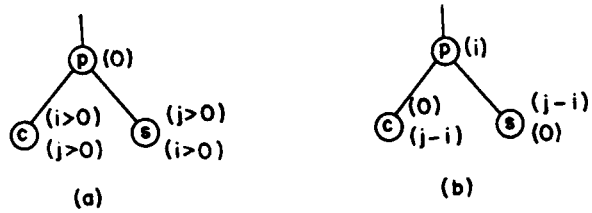


Figure 12: Case 2: Dealing with positive tags.

An essential feature of the restructuring process is that whenever the root of the tree obtains a non-zero tag it is changed back to zero. This preserves the relaxed AVL property of the tree, since it indicates that the root is either lower or higher than it appeared to be and neither of these effect its balance factor. Thus, we define the rest of the process such that it always searches for subtrees of height one, where the root of the subtree has a zero tag. Depending on the tag combinations within such a subtree the process performs different tasks. We have the following five cases.

Case 1. In this case we may simply move the tags upward and still have a relaxed AVL tree; see Figure 11.

Case 2. Here the notation means that either $tag(c) = i$ and $tag(s) = j$, or $tag(c) = j$ and $tag(s) = i$. Assuming that $j > i$, we obtain Figure 12.

Case 3. This is the case which was solved by Kessels [8]; see Figure 13.

Case 4. We consider the case $tag(c) = -1$ and $tag(s) = i$ in Figure 14; the reverse case is similar.

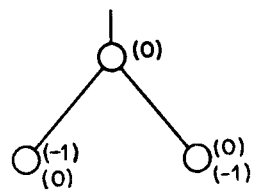


Figure 13: Case 3: A single negative tag.

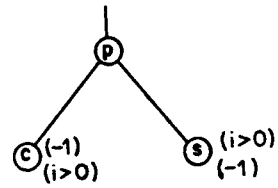


Figure 14: Case 4.

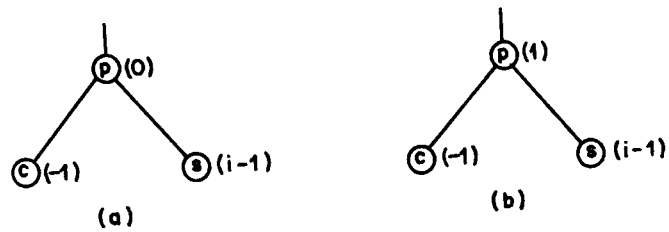


Figure 15: Case 4.1.

Case 4.1. $rbf(p) = 0$. We simply decrease $tag(s)$ by one; see Figure 15(a).

Case 4.2. $rbf(p) = -1$. In this case we transform it as shown in Figure 15(b).

Case 4.3. $rbf(p) = +1$, that is, c is the higher child. The situation may be depicted as shown in Figure 16. The relaxed heights of the nodes p, c, and s are $h+1, h$, and $h-1$, respectively. Now there are three cases depending on $rbf(c)$.

Case 4.3.1. $rbf(c) = 0$. In this case we do a single rotation; see Figure 17. Notice here that the tag-value sum decreases, because $VALUE(c)$ decreases and $VALUE(u)$ remains the same, for all other nodes u.

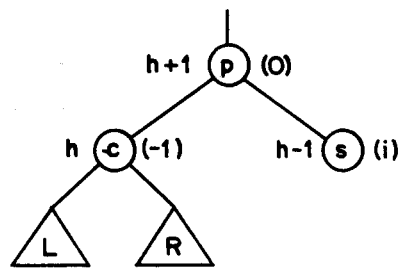


Figure 16: Case 4.3.

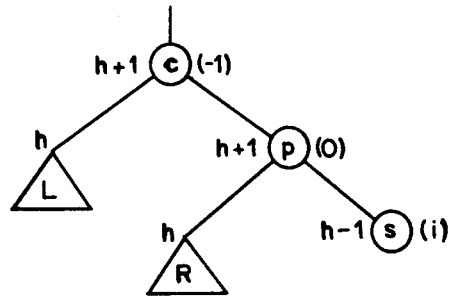


Figure 17: Case 4.3.1.

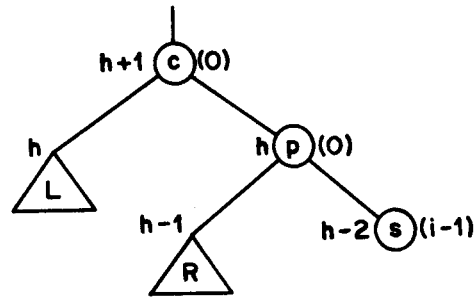


Figure 18: Case 4.3.2.

Case 4.3.2. Similarly, if $rbf(c) = +1$, we do a single rotation; see Figure 18.

Case 4.3.3. The case in which $rbf(c) = -1$ may require a double rotation. We depict the original situation again in Figure 19. We have three further subcases depending on $tag(g)$.

Case 4.3.3.1. $tag(g) = 0$. We do the double rotation shown in Figure 20.

Case 4.3.3.2. If $tag(g) = -1$, we also do a double rotation. If the relaxed heights of $L2$ and $R2$ are both h, we obtain Figure 21. The two remaining cases, that is, the relaxed height of either $L2$ or $R2$ is $h - 1$, are similar.

Case 4.3.3.3 It remains for us to consider the case $tag(g) = j > 0$. We simply decrease $tag(g)$ by one. This means that the negative tag at c may be cancelled, that is, $tag(c)$ becomes zero. No rotation is needed in this case.

Case 5. The only situation not yet considered is shown in Figure 22. We consider the case $tag(c) = i$ and $tag(s) = 0$; the reverse case is similar.

Case 5.1. $rbf(p) = 0$. This case is similar to Case 4.1.

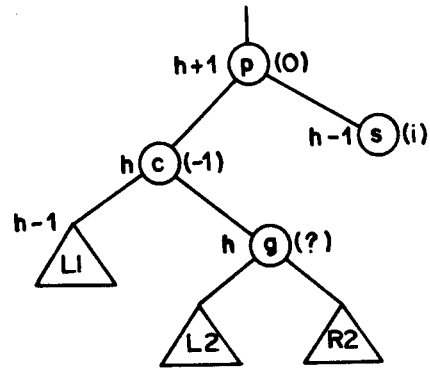


Figure 19: Case 4.3: The original situation.

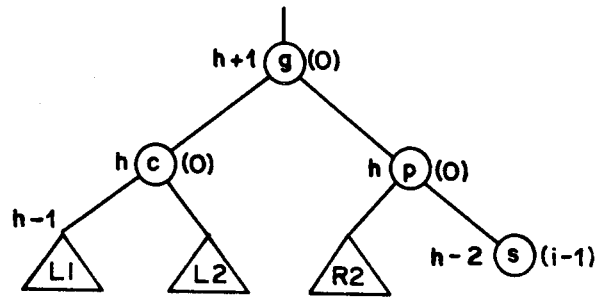


Figure 20: Case 4.3.3.1.

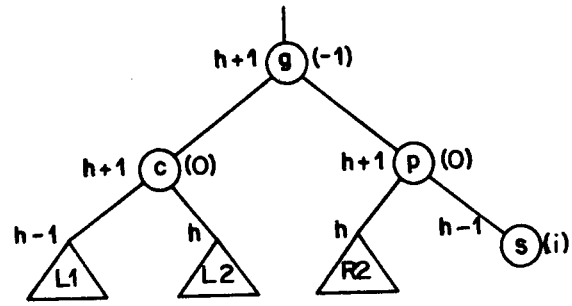


Figure 21: Case 4.3.3.2.

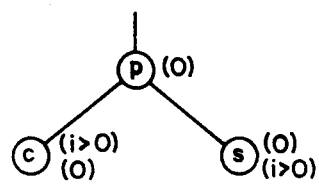


Figure 22: Case 5: A single positive tag.

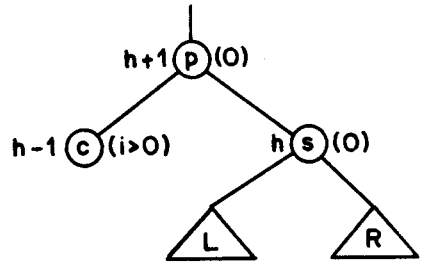


Figure 23: Case 5.3: The original situation.

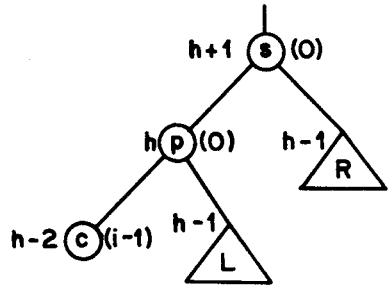


Figure 24: Case 5.3.1.

Case 5.2. $rbf(p) = +1$. This case is similar to Case 4.2.

Case 5.3. $rbf(p) = -1$, that is, s is the higher child. In this case we depict the situation as shown in Figure 23. There are three subcases to consider depending on $rbf(s)$.

Case 5.3.1. $rbf(s) = 0$. In this case we do the single rotation of Figure 24.

Case 5.3.2. If $rbf(s) = -1$, we also do a single rotation to obtain Figure 25.

Case 5.3.3. If $rbf(s) = +1$, we may require double rotation. The original situation is now depicted as given in Figure 26. The subcases again depend on the value of $tag(g)$.

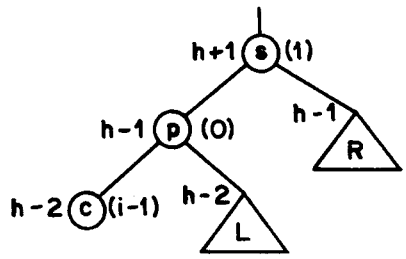


Figure 25: Case 5.3.2.

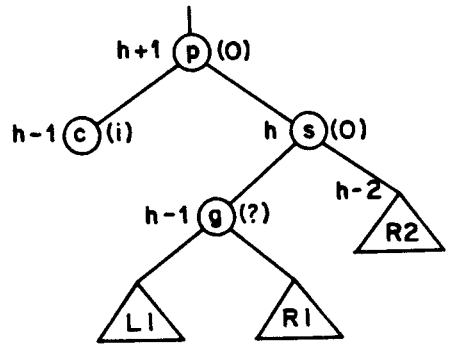


Figure 26: Case 5.3.3.

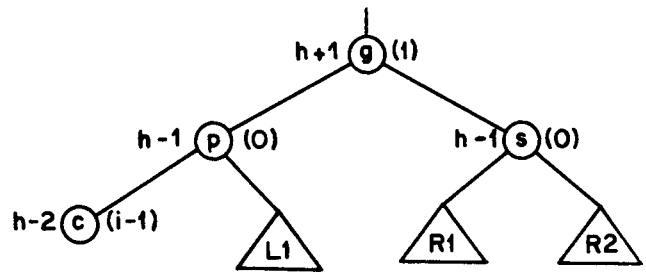


Figure 27: Case 5.3.3.1.

Case 5.3.3.1. $tag(g) = 0$. We do the double rotation of Figure 27.

Case 5.3.3.2 $tag(g) = -1$. The result is the same as in Case 5.3.3.1 but $tag(g)$ now has the value zero.

Case 5.3.3.3. If $tag(g) = j > 0$, we do no rotation, but we adjust the tag values as shown in Figure 28.

We are now in a position to prove the following theorem.

Theorem 4.1 *Let T be a relaxed AVL tree. Then, repeated application of the restructuring process given above, will, under the assumption that no further insertions and deletions take place, transform T into an AVL tree T'.*

Proof: Each nontrivial application of the restructuring process reduces $VALUE(T)$, by the case analysis above. Since no further insertions into or deletions from T take place, this implies that $VALUE(T)$ never increases. Therefore, after a finite number of such applications a relaxed AVL tree T' is obtained such that $VALUE(T') = 0$. But this means that T' is an AVL tree. □

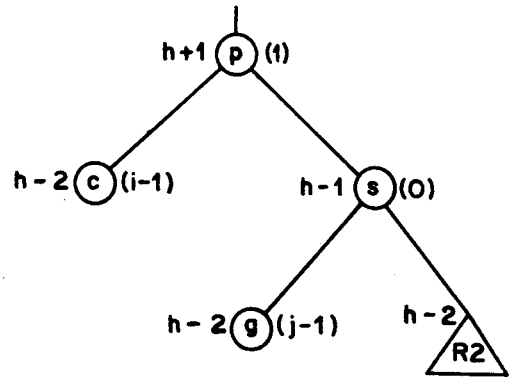


Figure 28: Case 5.3.3.3.

5 Concurrency in Relaxed AVL Trees

In this section we consider the problem of supporting several processes operating concurrently on an internal database stored as a relaxed AVL tree. Our solution to this concurrency control problem is related to the algorithms of Ellis [6] designed for concurrent updates in AVL trees, but we benefit from the uncoupling of the updates and the restructuring of the tree. In fact, we design a concurrency control method which is based only on locking of nodes, but still allows a high degree of concurrency. The environment of a relaxed AVL tree makes it possible for each concurrent process to hold a lock on a small number of nodes at any time. This principle holds also for the algorithms of Kung and Lehman [10] and Manber and Ladner [14] for concurrent updates in binary search trees.

In allowing an arbitrary number of processes to operate concurrently in a relaxed AVL tree, we assume that each of these processes progresses at a finite, but undetermined speed, and is performing one of the following operations:

SEARCH(K): To determine whether the key K is in the tree; if it is then it reports success, otherwise it reports failure.

INSERT(K): To add the key K to the tree if K is not already present in the tree.

DELETE(K): To remove the key K from the tree if K is present in the tree.

RESTRUCTURE: To perform one tag manipulation step as defined in Section 4. This may or may not involve a rotation in the tree.

In our solution three different lock types are used: read-locks (r-locks), write-locks (w-locks), and exclusive-locks (e-locks). These locks may interact in

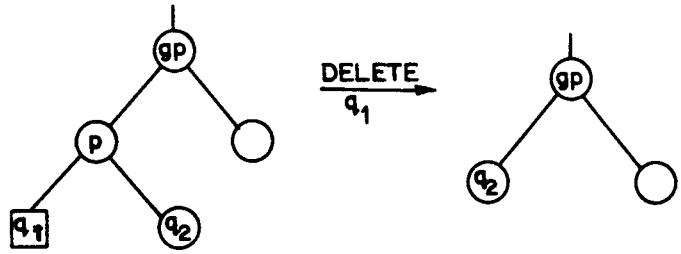


Figure 29: An example of deletion.

the following way. Several processes can hold an r-lock on a node even if it has one w-lock. Only one process can hold a w-lock or an e-lock on a node, and r-locks are not allowed with e-locks. A w-lock on a node may be converted into an e-lock.

We now describe the concurrent algorithms for $SEARCH$, $INSERT$, $DELETE$, and $RESTRUCTURE$. We use the term *lock-coupling* to mean that on the way down the tree the current node is always locked and the next node in the path is always locked before its parent is unlocked. Unlocking the parent may be done immediately after the child has been locked. In lock-coupling a process holds a lock on at most two nodes at a time. *Lock-tripling* is defined analogously, but in lock-tripling a lock is released from its parent only when a grand-child has a lock on it. In lock-tripling a process holds a lock on at most three nodes at any time.

$SEARCH(K)$: The algorithm uses r-lock-coupling on the way down the tree. At an external node the process reports success or failure depending on whether the external node has key K, then the process releases the r-lock on the external node and terminates.

$INSERT(K)$: The algorithm uses w-lock coupling from the root. If an external node is found which already has key K, then the process releases its w-lock on the external node and terminates. Otherwise an external node is found which should be expanded. The w-lock on the external node is converted into an e-lock and an insertion is performed. Note that this can be accomplished without changing the pointer in the parent node and thus the parent node need not be locked. Finally, the e-lock is released and the process terminates.

$DELETE(K)$: Deletion uses w-lock-tripling from the root. We assume that an external node is found which should be deleted; for example, see Figure 29. After checking that $key(q_1) = K$ the process still holds a w-lock on the parent p. The w-locks on p and q_1 are converted into e-locks, and the node q_2 is e-locked, too. Now we remove q_1 without changing the pointer field of the node p. Note that a lock on q_2 is

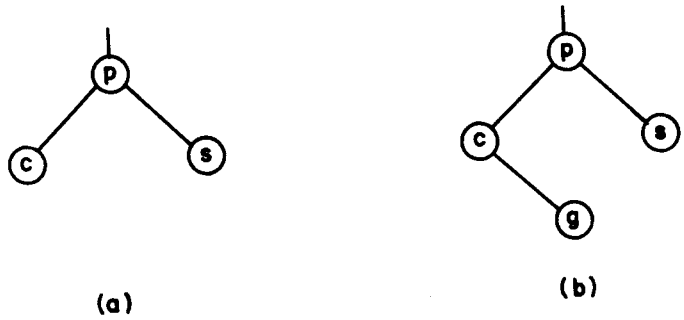


Figure 30: A three-node subtree.

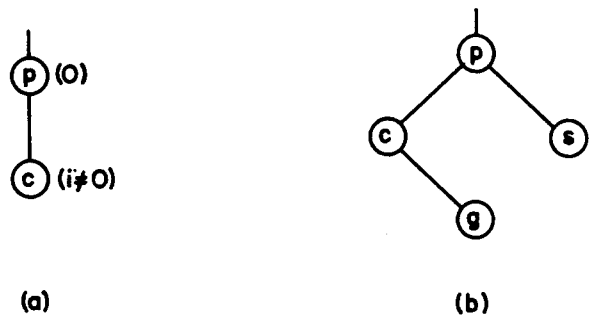


Figure 31: How restructuring is carried out.

necessary in order to guarantee that the tag of q_2 is not changed during deletion. Finally, all locks are released and the process terminates.

RESTRUCTURE: The restructuring algorithm involves a three-node subtree; see Figure 30(a). Moreover, in the case of a double rotation the right child of node c or the left child of node s will be affected. For example, assume that the nodes involved are those shown in Figure 30(b). The subtrees of s and g change their position, but this effect is obtained by modifying the pointers of the nodes p, s, and g. Rebalancing of the nodes p, c, s, and g is obtained by interchanging the keys of the nodes and by modifying the pointers appropriately. This means that the parent of p is not changed during the process.

The restructuring algorithm is implemented as follows. The algorithm searches for a pair of nodes as in Figure 31(a), where the node c has a non-zero tag and the parent p has a zero tag; c can be either the left or right child of p. We consider the case in which c is the left child, the other case is analogous. When such a pair is found, the process starts tag manipulation and possible rotations, if no other processes are holding the nodes that are possibly subject to change. A simple

solution to this is to lock all these nodes. Assume, for example, that these nodes are as given in Figure 31(b); then a w-lock is requested on all nodes p, c, s and g, in this order. (It is important to w-lock p first, then s, before locking g to avoid deadlocks with other processes applying w-locks.) When all these w-locks have been granted they are converted into e-locks, in the same order to avoid deadlocks with search processes. At this stage the restructuring process is free to perform its tasks, after which all locks are released and the process terminates.

For the correctness of the solution the following properties should hold:

P1. The insertion, deletion and search processes should interact correctly. That is, if at time t a change indeed has been carried out, then at any time $t + k$, $k > 0$, this change is taken into account by other processes not finished before time t.

P2. The rotation processes should interact correctly with respect to themselves and with respect to other processes. This means that concurrent rotation processes should not make changes to the same part of the tree and that parts of the tree involved in a change due to a deletion or an insertion are not changed by a rotation process. Moreover, any process advancing down the tree is not affected by a rotation process.

Property P1 is clear because all insertion and and deletion processes will be totally ordered. Pure readers do not obey this total ordering but the exclusive lock before any actual insertion or deletion prevents incorrect reports.

For Property P2 first note that exclusive locking applied by rotation processes and insertion and deletion processes imply that the actual changes will not be affected by each other. Similarly, exclusive locking by a rotation guarantees that no process advancing down the tree can be present at any node subject to a change in the rotation.

6 Concluding remarks

Relaxed AVL trees are so wide a class that every binary tree can be viewed as a relaxed AVL tree. In Figure 32 we demonstrate how a maximal height tree can be made to appear AVL using only negative tags. In Figure 33 we use only positive tags to achieve the same result. Therefore one obvious question is: Why use AVL trees at all? This leads to: Since a relaxed AVL tree can be as unbalanced as an arbitrary binary tree why not use binary trees? The difference between the two is that we expect relaxed AVL trees to be close to AVL trees throughout their lifetime. (An investigation of this

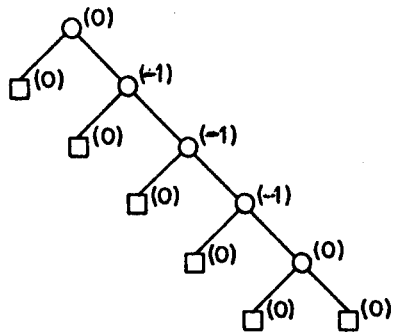


Figure 32: Maximal height binary trees are relaxed AVL.

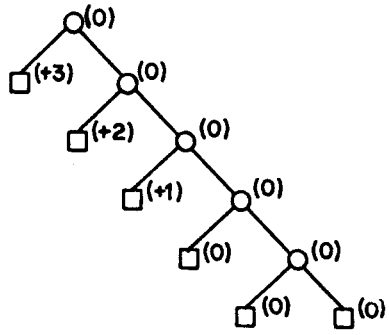


Figure 33: Using only positive tags to obtain relaxed AVL trees.

expectation is currently underway), whereas arbitrary binary trees may be far from balanced.

References

[1] G.M. Adel'son-Vel'skii and Y.M. Landis. An algorithm for the organization of information. *Doklady Akademi Nauk*, 146:263–266, 1962.

[2] G.B. Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21:280–294, 1978.

[3] R. Bayer and M. Schkolnick. Concurrent operations on B-trees. *Acta Informatica*, 9:1–21, 1977.

[4] A. Biliris. *Concurrency Control on Database Indexes: The mU Protocol.* Technical Report 85/014, Boston University, Computer Science Department, 1985.

[5] E.W. Dijkstra, L. Lamport, A.J. Martin, C.S. Scholten, and E.F.M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *Communications of the ACM*, 21:966–975, 1978.

[6] C.S. Ellis. Concurrent search and insertion in AVL trees. *IEEE Transactions on Computers*, C-29:811–817, 1980.

[7] L.J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pages 8–21, 1978.

[8] J.L.W. Kessels. On-the-fly optimization of data structures. *Communications of the ACM*, 26:895–901, 1983.

[9] D.E. Knuth. *The Art of Computer Programming, Vol.3: Sorting and Searching.* Addison-Wesley Publishing Co., Reading, Mass., 1973.

[10] H.T. Kung and P.L. Lehman. A concurrent database manipulation problem: binary search trees. *ACM Transactions on Database Systems*, 3:339–353, 1980.

[11] Y.S. Kwong and D. Wood. Approaches to concurrency in B-trees. In *Mathematical Foundations of Computer Science, Lecture Notes in Computer Science 88*, pages 402–413, 1980.

[12] T.J. Lehman and M.J. Carey. *Query Processing in Main Memory Database Management Systems.* Technical Report, University of Wisconsin, 1986.

[13] T.J. Lehman and M.J. Carey. *A Study of Index Structures for Main Memory Database Management Systems.* Technical Report, University of Wisconsin, 1986.

[14] U. Manber and R.E. Ladner. Concurrency control in a dynamic search structure. *ACM Transactions on Database Systems*, 9:439–455, 1984.

[15] Y. Sagiv. Concurrent operations on B-trees with overtaking. *Journal of Computer and System Sciences*, 1986.