Concurrency Control in Database Structures with Relaxed Balance *

Otto Nurmi† Eljas Soisalon-Soininen†
Institut fur Angewandte Informatik
und Formale Beschreibungsverfahren
Universitat Karlsruhe
Postfach 6380, D-7500 Karlsruhe 1, West Germany

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

Abstract. We consider the separation of rebalancing from updates in several database structures, such as B-trees for external and AVL-trees for internal structures We show how this separation can be implemented such that rebalancing is performed by local background processes Our solution implies that even simple locking schemes (without additional links and copies of certain nodes) for concurrency control are efficient in the sense that at any time only a small constant number of nodes must be locked

1 Introduction

We consider various data structures for efficient implementation of a database, when concurrent insertions, deletions and member operations should be supported For efficient search the data structure should be balanced, as B-trees for external structures and AVL-trees (or, for example, the recently defined T-trees for storing in-core databases [13]) for internal structures However, when balancing is strictly connected with the update, that is, the search tree is always kept strictly in balance, it is not possible to split the dictionary operations into processes causing only local modifications in the structure. In such an environment simple locking schemes for concurrency control, as given, for example, in [4,5,16], are not very efficient, and the more advanced solutions [4,5,6,10,11,12,14,15] tend to become rather complex Recently, asynchronous balancing of search trees, that is, the uncoupling of the update and the balancing operations have been suggested [6,7,8,15] For AVL-trees the uncoupling has been defined by Kessels [8] in the case when only insertions and member operations are present B-trees or their variants are considered in [6,15] and different solutions are developed for handling "underflows" and "overflows" in nodes by separate local processes In particular, in the solution of Goodman and Sasha [6], temporary node overflows are simply allowed, and the method of Sagiv [15], while handling node overflows properly, is based on a special version of B-trees, called B^{link}-trees [12]

Our contributions in this area are based on a general concept of "relaxed balance" We define this concept for binary search trees and for general ordered multiway leaf-oriented trees, such as B^+-trees Besides the relaxed balance, no modifications in the structures are introduced, that is, search algorithms and the node size, for example, are preserved Every tree that is balanced in the relaxed sense may be transformed into a truely balanced tree by a finite number of invocations of local processes Using this new concept we are able to implement the uncoupling of updates and balancing efficiently and in a unified way for a large class of tree structures In particular, for binary search trees, our contribution is to allow the separation of restructuring also when deletions are present, not only in the case of insertions and member operations as in the solution by Kessels [8] Our solution for binary search trees can be carried over to their new variants [13] developed for storing in-core databases

*This research was supported by the Deutsche Forschungsgemeinschaft (O Nurmi), under a Natural Sciences and Engineering Research Council of Canada Research Grant No A-5692 (E Soisalon-Soininen and D Wood), and by the Alexander von Humboldt Foundation (E Soisalon-Soininen)

†On leave from the University of Helsinki, Department of Computer Science

2 Concurrency in B-trees

An efficient and widely used external structure for organizing data is a multi-way tree called B-tree [1] The B-tree has several variants, as B*-trees [9], B+-trees [2], B^{link}-trees [12], etc

Methods for concurrent operations on B-trees are discussed by several authors Most of them [e g 10,12,16] require that the tree is left balanced after updating This makes the search in the structure rapid but allows often only a small number of processes to operate concurrently, especially when simple locking schemes [e g 16] are used for avoiding undesired consequences The rate of concurrency can be increased by using more advanced locking strategies [e g 11,12,15] These strategies often require that parts of the tree are copied during the restructuring, or that the tree is modified by additional links [12,15] The amount of concurrency may also be increased by separating the restructuring from the update This has been proposed by Sagiv [15] and Goodman and Sasha [6] One restructuring operation is divided into small parts, as into "split" and "combine" operations, which modify the tree only locally The complete restructuring consists of several invocations of such local processes

Sagiv [15] presents a separate "compression" process which removes the "underflows" arisen from deletions The "overflows" are removed with the aid of "level links" that cause the insert operations to become local The level-linked B-trees, B^{link}-trees, are adopted from [12] Goodman and Sasha [6] ignore the problem of overflow by allowing the nodes to contain temporarily more data than prescribed This can, however, lead to difficulties in the implementation, because the node size is often strictly limited by the hardware in which the database is stored Both methods allow temporary relaxing of the strict balance of the tree until a separate process removes the unbalance The B^{link}-trees used in the solution of Sagiv allow some relaxing already by their definition the lengths of the search paths from the root to the leaves can vary

We adopt the underflow handling from Sagiv [15] and Goodman and Sasha [6] The operations on overflows are made local by introducing new "insertion safe" nodes immediately above the nodes to be split This makes the use of very simple locking schemes efficient, because the distance from the safe node to the overflow node is always one link That is, without additional links, we obtain a structure which is as useful for concurrency as B^{link}-trees The later restructuring processes make the paths to the leaves to have equal lengths

We do not fix a special concurrency control algorithm in our solution because all of the schemes given in the literature [e g 10,11,16] are easily adopted to our structure However, in the following discussion we assume that a simple locking scheme as given in [16] will be used

3 Relaxed B-Trees

We shall consider ordered multi-way leaf-oriented trees All the data are stored in the leaves The internal nodes contain only pointers to subtrees and routing information to aid the search in the tree If a node has $j > 0$ sons it contains $j-1$ routers, and is often represented by a sequence $P_0K_1P_1K_2P_2 \quad K_{j-1}P_{j-1}$, where K_i is a router and P_i is a pointer The keys in the subtree pointed by P_{i-1} (P_i) are less than or equal to (greater than) K_i $(1 \le i \le j-1)$

Let u be a node Denote by $c(u)$ the number of pointers in u if u is an internal node, and the number of keys in u if u is a leaf If u is not the root, we denote its father by φu The function $l(u)$, defined by

$$l(u) = \begin{cases} 0, & \text{if } u \text{ is the root,} \\ l(\varphi u) + 1, & \text{otherwise,} \end{cases}$$

is the *level* of u

Let now $k \ge 2$ be an integer A *B-tree of order* $2k$ is an ordered multi-way leaf-oriented tree with the following properties

B1 $\begin{cases} 2 \le c(u) \le 2k, & \text{if } u \text{ is the root,} \\ k \le c(u) \le 2k, & \text{otherwise} \end{cases}$

B2 If u_1 and u_2 are two leaves of the tree, then $l(u_1) = l(u_2)$

B-trees defined in this way are often referred to as B+-trees

The condition B1 is a density condition Too empty nodes are forbidden After deletions, the tree must be *compressed* in order to fulfil B1 B2 is a balance condition After insertions, the tree must be restructured by *split* operations in order to fulfil B2

When compression and split operations are delayed the tree may not always satisfy the properties of a B-tree We therefore relax the conditions B1 and B2 appropriately

We shall allow the paths to the leaves to have different lengths We add a *tag bit* into the nodes The tag bit will later tell the "splitter" that an action should take place For convenience, we denote the status of the bit by -1 or 0

Let u be a node of the tree The *relaxed level* of u, denoted $rl(u)$, is defined by

$$rl(u) = \begin{cases} 0, & \text{if } u \text{ is the root,} \\ rl(\varphi u) + 1 + tag(u), & \text{otherwise} \end{cases}$$

If all tags have the value 0 along the path from the root to u, then $rl(u)$ equals to the level of u

Let $k \ge 1$ be an integer An ordered multi-way leaf-oriented tree is a *relaxed B-tree* of order $2k$, if it has the following properties

RB1 $\begin{cases} 0 \le c(u) \le 2k, & \text{if } u \text{ is a leaf,} \\ 1 \le c(u) \le 2k, & \text{otherwise} \end{cases}$

RB2 If u_1 and u_2 are two leaves of the tree, then $rl(u_1) = rl(u_2)$

Clearly a B-tree is a relaxed B-tree A relaxed B-tree is a B-tree, if the tag of each node is zero and the density condition B1 of B-trees is satisfied

The search for a key is carried out exactly in the same way as in B-trees The update operations are designed in such a way that they preserve the properties RB1 and RB2 These operations make only local changes in the tree (i e , only a small constant number of nodes are affected), while the number of nodes affected by conventional B-tree updating is a function of the number of nodes in the tree

4 Updates for Relaxed B-trees

In this section we shall describe the insert and delete operations for relaxed B-trees, and we shall delineate the restructuring process that transforms the relaxed B-tree closer to a conventional B-tree

Insert First, the leaf u is searched in which the new key should be stored If $c(u) < 2k$, the key is inserted into leaf u If $c(u) = k$ (overflow situation) we introduce two new leaves, move the leftmost half of the contents of u into the first new node, and the rightmost half into the second new node The new key is then inserted into the first or second new leaf Then, node u is put as the father of the new leaves and the routers of u are set accordingly Last, the tags of the leaves are set to zeroes, and the tag of u is set to -1 (see Figure 1) During this operation only u must be (exclusively) locked

Delete The leaf u which contains the key to be deleted is searched, and the key is deleted No further actions take place Note that the routing information always remains valid in spite of a key deletion During this operation only u must be locked

Next we shall describe the restructuring operations, which will keep the tree as a relaxed B-tree but modify it closer to a conventional B-tree The compress operation removes underflows from the nodes It is adopted from Sagiv [15] and Goodman and Sasha [6] The split operation moves the negative tags towards the root When a negative tag will reach the root, it is immediately set to zero because a negative tag in the root has no effect on possible differences in path lengths When all negative tags have disappeared the balance condition B2 of B-trees is satisfied Repeated invocations of these operations after an update will yield a conventional B-tree

We assume that the operations have valid candidate nodes, that is, the compress operation has a node whose son u has $c(u) < k$, and the split operation has a node whose son has a negative tag We shall discuss later how the candidates are found

Compress Let u be a node with $c(u) < k$ and φu its father The contents of u are moved to its left or right brother, if they have enough empty space, and node u is deleted The link pointing to u in φu, and the corresponding router, are deleted If the contents of u does not fit into its brothers, then u and the contents of its left (or right) brother are equally divided between these two nodes The routing information in φu is updated accordingly During this operation, φu, u, and the left (right) brother of u must be (exclusively) locked

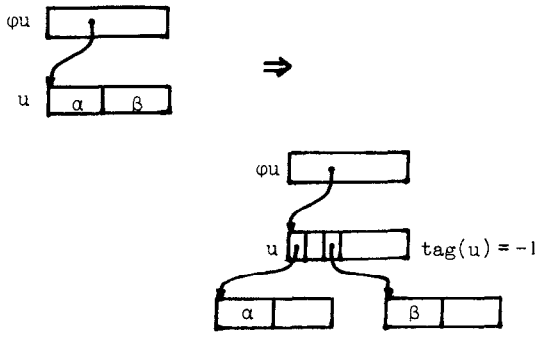

Figure 1 Insertion in an overflow situation

Split Let u be a node with negative tag, and φu its father We have two cases First, if $c(u) + c(\varphi u) \leq 2k$, then the contents of u is moved into φu, starting at the place of the link to u The node u is removed The tag of φu remains zero (see Figure 2(a)) Second, if $c(u) + c(\varphi u) > 2k$, a new node is introduced A similar sequence as in the previous case (having now length $2k < c(u) + c(\varphi u) < 4k$) is equally divided between u and the new node The node φu will have two pointers, one to u and one to the new node The router between the pointers is set accordingly The tags of u and the new node are set to zero and the tag of φu is set to -1, if φu is not the root (see Figure 2(b)) Otherwise $tag(\varphi u)$ remains 0 Two nodes, φu and u, must be locked during the operation

Finally, we sketch three methods for finding nodes for restructuring First, we may queue the candidates when they arise Both the updaters and the restructurers consult the queue The restructurers take candidates from the queue, check (after having locked the candidate and its appropriate son) whether the operation is still to be performed, perform the the operation, release the locks, and store the newly arisen candidates into the queue This solution was proposed in [6]

In the second method the restructurers traverse all the time in the tree searching for candidates When a process goes downwards in the tree, it uses the "lock coupling" technique (a lock on the father is not released before its son has been locked), and if a candidate is found, the corresponding operation is performed, and the search through the tree will be continued This method is similar to the one proposed by Sagiv [15], in which, however, the process advances along the level links

The third possibility is to invoke a separate restructuring process for a candidate immediately after creating it In this case, before actual restructuring the process must check whether it is still needed This solution requires that the number of separate processes is not limited

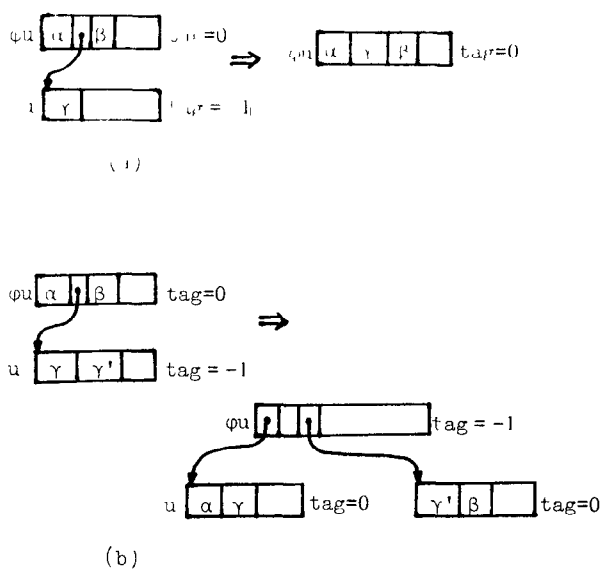

Figure 2 The split operation

5 Relaxed Height-Balanced Binary Trees

One method for increasing the efficiency of a database system is to enlarge the main memory and to try to store the whole database in it The data structures developed for external storage are not the only candidates for efficient implementation of main memory databases Lehman and Carey [13] have investigated the efficiency of several data structures in main memory environment Besides the typical external storage structures they proposed AVL-trees, T-trees, and arrays as candidates for implementing such databases The T-trees [13] are essentially AVL-trees having nodes as in B-trees The simple B-tree type restructuring is often sufficient to keep the T-trees in balance If not, AVL-type rebalancing is performed

In this and the following section we shall show how relaxed balancing can be applied in the case of AVL-type rebalancing This makes it possible to uncouple the rebalancing and the update operations, thus increasing possibilities for concurrency in a database, in which AVL-type restructuring is needed By "AVL-type rebalancing" we mean here that the tree is a binary search tree, and the balance criterion is based on the differences between the heights of the subtrees of the nodes Our work is based on the results of Kessels [8], who introduced the term "on-the-fly optimization" of data structures in analogy to "on-the-fly garbage collection" [3] By the term Kessels means that the restructuring is performed as a background process As a major example Kessels considered height-balanced trees and restructuring after insertions In his solution, a binary tag were attached to the nodes, indicating the possible necessity of a rotation We extend his method to include also deletions We attach to the nodes tags having an integer value greater than or equal to -1, whereas (in our terminology) Kessels used only tags of value 0 or -1 The task of the restructuring process is to decrease the sum of the absolute values of the tags When this sum is 0, the tree is balanced

We assume again that all the data are stored in the leaves The internal nodes contain two pointers, one pointing to the left and one to the right subtree, and a router value The router value is assigned so that the data in the left (right) subtree are less than or equal to (greater than) the data in the other subtree Besides this information the nodes of *relaxed* trees contain a tag field for the tag of the node The internal nodes store also a balance factor, an integer between -1 and $+1$

Let u be a node of a binary search tree We denote u's father by φu, u's left child by λu, and u's right child by ρu The *height* of u, $h(u)$ is defined by

$$h(u) = \begin{cases} 1, & \text{if } u \text{ is a leaf,} \\ \max(h(\lambda u), h(\rho u)) + 1, & \text{otherwise} \end{cases}$$

If u is an internal node, its *balance factor*, $bf(u) = h(\lambda u) - h(\rho u)$ The tree is *height-balanced*, if for all internal nodes u, $-1 \leq bf(u) \leq 1$

If u is a node, we denote its tag by $tag(u)$ The *relaxed height* of u, $rh(u)$ is defined by

$$rh(u) = \begin{cases} 1 + tag(u), & \text{if } u \text{ is a leaf,} \\ \max(rh(\lambda u), rh(\rho u)) + & \\ \qquad 1 + tag(u), & \text{otherwise} \end{cases}$$

During updates values are assigned to tags in such a way that the relaxed height of the father of the modified subtree remains constant

The relaxed height-balance condition is defined in analogy to the usual strict condition Let u be an internal node The *relaxed balance factor* of u, $rbf(u)$, is defined by

$$rbf(u) = rh(\lambda u) - rh(\rho u)$$

The tree is *relaxed height-balanced*, if for all internal nodes u, $-1 \leq rbf(u) \leq +1$

Clearly a height-balanced tree is relaxed height-balanced, and a relaxed height-balanced tree is height-balanced if all tags have the value zero The inverse of the latter statement is not true, a tree can be height-balanced even though some of the tags were non-zeroes

During insertions and deletions the pointers are manipulated as in conventional binary leaf search trees A deletion is always a local operation, and no routers must be updated (Leaf oriented trees would also solve the difficulties in deletions in the algorithm by Kung and Lehman [10] Another solution is proposed by Manber and Ladner [14]) Search algorithms need no modifications

An *insertion* adds an internal node and a leaf into the tree The tags of the new leaf and its brother are set to zero, and the tag of the new internal node is obtained by decreasing the tag value of the brother by one (see

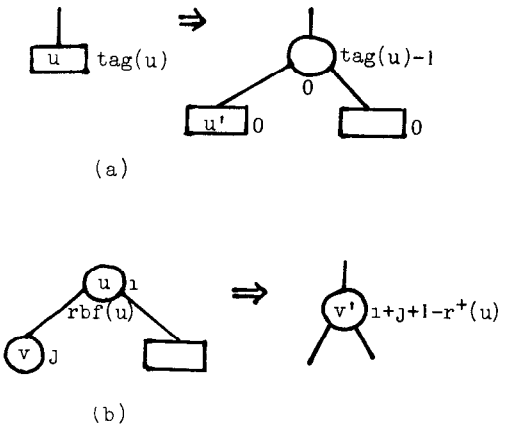

Figure 3 (a) Insertion yields a right brother (b) The right brother is deleted Tags are shown on the right of the nodes, relaxed balanced factors below the nodes After the manipulation original node t is denoted by t'

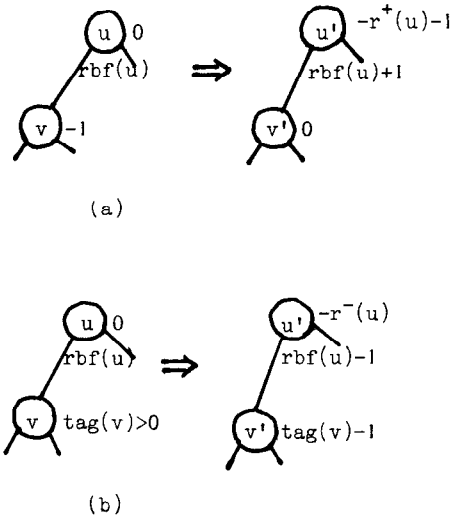

Figure 4 (a) $tag(v) = -1$ (b) $tag(v) > 1$

Figure 3(a)) Due to this tag manipulation, the relaxed height of the father of the new internal node remains unaltered, and thus insertions do not cause any violation of the relaxed balance

A *deletion* removes a leaf and an internal node Let u be the internal node and v the brother of the deleted leaf The new tag of v depends on the relaxed balance factor of u For brevity, we define two auxiliary functions, $r^+(u)$ and $r^-(u)$, where u is an internal node

$$r^+(u) = \min(0, rbf(u)), \quad r^-(u) = -\max(0, rbf(u))$$

(The value of $r^+(u)$ ($r^-(u)$) is -1, if $rbf(u) = -1$ ($rbf(u) = 1$), and 0, otherwise) Now the tag value of the brother v will be $tag(u) + tag(v) + 1 - r^+(u)$ ($tag(u) + tag(v) + 1 - r^-(u)$), if v was the left (right) brother of u (see Figure 3(b)) Again the relaxed height of the father remains unaltered

Note that the negative tags cannot accumulate Thus the tag values remain greater than or equal to -1

The update operations are local during an insertion only 2 nodes must be locked, and during a deletion it is enough to lock 3 nodes, even if a simple locking scheme without copying and without extra links is used

6 Restructuring Relaxed Height-Balanced Trees

The task of the restructuring processes is to get the tree (more) balanced in the conventional sense This is achieved by trying to decrease the sum of the absolute values of the tags in the tree If the root of the tree gets a non-zero tag, it is immediately set to zero, because it cannot affect the balance Thus the processes try to decrease the tag value sum or to move the tags towards the root

We assume that the restructuring process has a valid candidate, that is, a node u with $tag(u) = 0$ but having a son v with $tag(v) \neq 0$ Because the tag of the root is always zero, valid candidates exist whenever all the tags are not zero

Each invocation of the process performs some local operations in the tree in such a way that the following two conditions hold

- R1 Either the sum of the absolute values of the tags decreases, or the sum remains the same but the tree has a subtree such that the absolute value of the tag in the root of the subtree increases in favor of another node in the subtree
- R2 The relaxed height of the father of u remains unaltered

Clearly, R1 implies that a finite number of invocations of the process yields a tree in which all tags have the value zero, and thus a truely height-balanced tree Condition R2 prevents the propagation of possible violations against the relaxed balance The need of rotations does not ascend in the tree as when maintaining the strict balance

Given a candidate u with $tag(u) = 0$ and its son v with $tag(v) \neq 0$, we decrease the absolute value of $tag(v)$ by one Then we set the tags so that R2 will hold, and calculate $rbf(u)$ If $rbf(u)$ gets the value 2 (-2), a rotation to the right (left) may be necessary For brevity, we explain the operations only in the case in which v is the left son of u The other case is a reflection of the explained one Similarly, we explain the rotation only to the right Two cases arise depending on whether $tag(v) = -1$ or $tag(v) > 0$ The new tags and balance factors are shown in Figure 4 If one of the sons of u has tag -1 and the other a positive tag, we first remove the negative tag

Let us now assume that in the above process $rbf(u)$ got the value 2 If $tag(v)$ is still greater than 0, we can

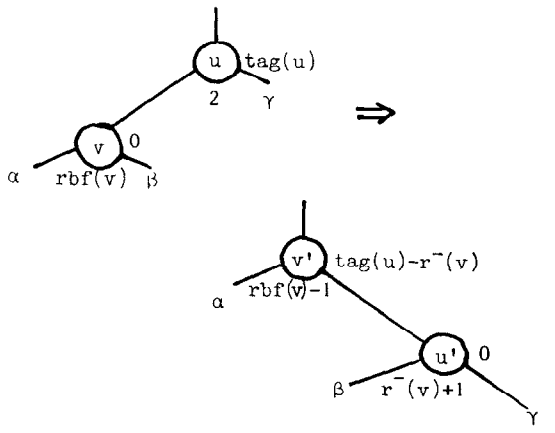

Figure 5 A single rotation

simply apply the operation depicted in Figure 4(b) in order to retain the relaxed balance Thus we can assume that $tag(v) = 0$ Because $rbf(u)$ got the value 2 in the modifications shown in Figure 4, we conclude that the new value of $tag(u)$ is either 0 or -1

The following operations depend on $rbf(v)$ If $rbf(v) \geq 0$, a single rotation is needed The tags are again calculated in such a way that the relaxed height of the root of the subtree remains unaltered (see Figure 5) The tag value of v can increase from 0 to 1 But v becomes the root of the subtree and the modification of Figure 4 decreased the tag of another node in the subtree Thus we conclude that R1 still holds

If $rbf(v) = -1$, a double rotation may be necessary The operations to be carried out depend on $tag(w)$, where w is the right son of v If $tag(w) > 0$, adjusting the tags appropriately will retain the balance (see Figure 6) Otherwise, a double rotation is needed, and the tags are again calculated so that the relaxed height of the root of the subtree remains constant (Figure 7) As

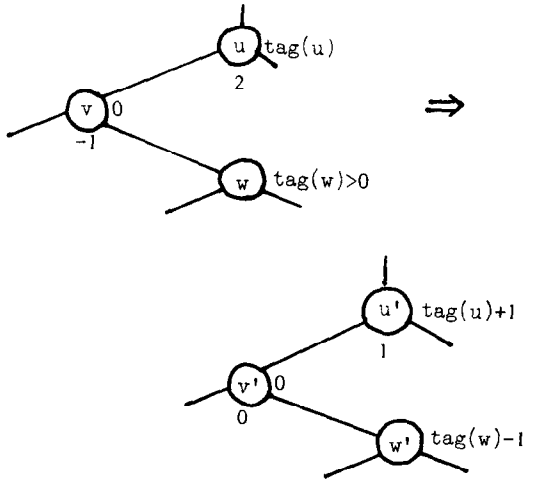

Figure 6 $tag(w) > 0$, no rotations are needed

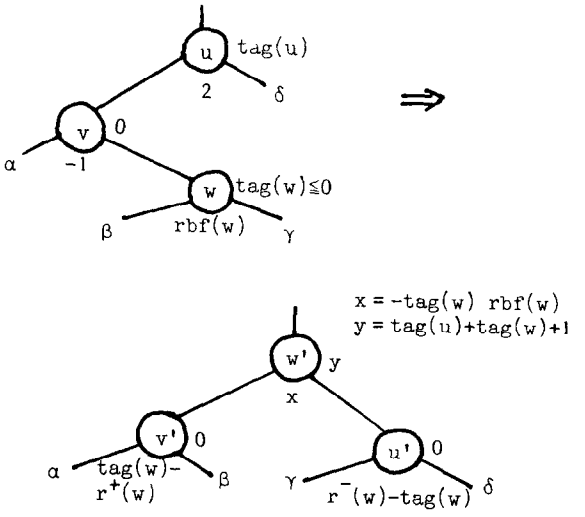

Figure 7 $tag(w) \leq 0$, a double rotation is needed

before, it is immediately clear that condition R1 holds after the modifications

During the restructuring operations the contents of at most 4 nodes are changed (u, v, w, and the brother of v) If the simple locking scheme without additional links and copies is used, these nodes must be locked during the operation

The candidates for rebalancing processes are found in a similar way as in the case of B-trees If the simple locking scheme is used for concurrency control, then for searching the candidates we must use lock-tripling (the lock on the grandfather is not released before locking a node) instead of lock-coupling

References

[1] Bayer,R and E McCreight, Organization and maintenance of large ordered indexes *Acta Inf* **1** (1972), 173–189

[2] Comer,D , The ubiquitous B-tree *ACM Comput Surveys* **11** (1979), 121–138

[3] Dijkstra,E W , L Lamport, A J Martin, C S Scholten, and E F M Steffens, On-the-fly garbage collection An exercise in cooperation *Comm ACM* **21** (1978), 966–975

[4] Ellis,C S , Concurrent search and insertion in AVL trees *IEEE Trans on Computers* **C-29** (1980), 811–817

[5] Ellis,C S , Concurrent search and insertion in 2-3 trees *Acta Inf* **14** (1980), 63–86

[6] Goodman,N and D Sasha, Semantically based concurrency control for search structures *Proc Fourth ACM SIGACT-SIGMOD Symp on Principles of Database Systems*, 1985, pp 8–19

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

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

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

[10] Kung,H T and P L Lehman, A concurrent database manipulation problem binary search trees *ACM Trans Database Syst* **5** (1980), 339–353

[11] Kwong,Y S and D Wood, A new method for concurrency in B-trees *IEEE Trans Soft Eng* **SE-8** (1982) 211–222

[12] Lehman,P L and S B Yao, Efficient locking for concurrent operations on B-trees *ACM Trans Database Syst* **5** (1981), 650–670

[13] Lehman,T J and M J Carey, Query processing in main memory database management systems *Proc ACM SIGMOD Conf*, 1986, pp 239–250

[14] Manber,U and R E Ladner, Concurrency control in a dynamic search structure *ACM Trans Database Syst* **9** (1984), 439–455

[15] Sagiv,Y , Concurrent operations on B-trees with overtaking *Journal of Computer and System Sciences* **33** (1986)

[16] Samadi,B , B-trees in a system with multiple users *Inform Process Lett* **5** (1976), 107–112