# AVL Trees with Relaxed Balance

Kim S. Larsen

Department of Mathematics and Computer Science, Odense University
Campusvej 55, 5230 Odense M, Denmark

## Abstract

*AVL trees with relaxed balance were introduced with the aim of improving runtime performance by allowing a greater degree of concurrency. This is obtained by uncoupling updating from rebalancing.*

*We define a new collection of rebalancing operations which allows for a significantly greater degree of concurrency than the original proposal. Additionally, in contrast to the original proposal, we prove the complexity of the rebalancing. If $N$ is the maximum size the tree could ever have, we prove that each insertion gives rise to at most $\lfloor \log_\phi(N+\frac{3}{2}) + \log_\phi(\sqrt{5}) - 3 \rfloor$ rebalancing operations and that each deletion gives rise to at most $\lfloor \log_\phi(N+\frac{3}{2}) + \log_\phi(\sqrt{5}) - 4 \rfloor$ rebalancing operations, where $\phi$ is the golden ratio.*

## 1 Introduction

A dictionary is a data structure which supports the operations search, insert, and delete, where the latter two are called the updating operations. One standard implementation of a dictionary is the AVL tree [1], which is a height balanced tree ensuring that all the operations can be performed in time logarithmic in the size of the tree. Recall that as an effect of an update, the AVL tree balance conditions may be violated, in which case rebalancing is necessary. In the worst case, this involves a traversal of a path from a leaf to the root.

When implementing data structures in a concurrent environment, parts of the data structure, which are to be changed, must be locked to prevent simultaneous access to the same data. In naïve concurrent implementations of search trees, this generally implies that the whole path from the root down to the leaf being updated must be locked, just in case rebalancing all the way up will turn out to be required. Clearly, such an approach leads to a very low degree of concurrency.

These considerations lead to the idea of uncoupling the updating from the rebalancing. This was first suggested in [4] and was first studied in connection with AVL trees in [5]. In connection with red-black trees [2, 4], a scheme similar to the one presented here was suggested in [7] and a proof of complexity for an improved collection of rebalancing operations was given in [3].

In [8], the data structure AVL tree with relaxed balance is proposed. While that proposal certainly makes updating faster, it is not clear how much harder rebalancing becomes, as no proof of complexity is included.

The major problem with the proposal of [8] is this lack of a proof of complexity. First of all, it is unsatisfactory not to know the complexity, since minor changes in the definition of the operations can change the complexity dramatically. In [3], it is demonstrated that the red-black tree proposal of [7] is $\Omega(k^2)$ for $k$ updates, but changing just one operation very slightly makes it $O(k \log k)$.

Secondly, the proposal [8] has too many restrictions on when rebalancing operations can be applied, thus limiting the degree of concurrency. The authors of [8] probably feared that fewer restrictions could ruin the complexity. Having a proof, one knows exactly how much freedom one has in these definitions.

Finally, in an actual implementation, one might want to make minor changes throughout the operations. Once a proof has been set up, it is fairly easy to go through the details of the proofs to check that such changes do not effect the overall complexity.

In this paper, we present a slightly modified collection of rebalancing operations, allowing for a larger degree of concurrency in the rebalancing. Additionally, we accompany our proposal with a proof of complexity. Suppose the original tree $T$ has $|T|$ nodes before $k$ insertions and $m$ deletions are performed. Then $N = |T| + 2k$ is the best bound one can give on the maximum number of nodes the tree ever has, since each insertion creates two new nodes (see below). We show that the tree will become rebalanced after at most $k\lfloor \log_\phi(N+\frac{3}{2}) + \log_\phi(\sqrt{5}) - 3 \rfloor + m(\lfloor \log_\phi(N +$



0-8186-5602-6/94 

$\frac{3}{2}) + \log_\phi(\sqrt{5}) - 4\rfloor)$ rebalancing operations, where $\phi$ is the golden ratio. This is $O(\log(N))$ per update.

## 2 AVL trees with relaxed balance

The trees considered here are leaf-oriented binary search trees, so the keys are stored in the leaves and the internal nodes only contain routers which guide the search through the tree. The router stored in a node is greater than or equal to any key in the left subtree and less than any key in the right subtree. The routers are not necessarily keys which are present in the tree, since we do not want to update routers when a deletion occurs. The tree is a full binary tree, so each node has either zero or two children.

If $u$ is an internal node, then we let $u_l$ and $u_r$ denote the left and right child of $u$, respectively. The height of a node $u$ is defined by $h(u) = 0$, if $u$ is a leaf, and $h(u) = \max(h(u_l), h(u_r)) + 1$, otherwise.

The balance factor of an internal node $u$ is defined by $bf(u) = h(u_l) - h(u_r)$. Now, a tree is an AVL tree if the height of the children of any internal node $u$ differ with at most one, i.e., $bf(u) \in \{-1, 0, 1\}$.

We want to use AVL trees, but at the same time we want to uncouple updating from rebalancing. The question arises as to how we allow the tree to become somewhat unbalanced without loosing control completely. A technique introduced by Kessels [5] can be used in a modified form.

Every node $u$ will have an associated tag value $tag(u)$, which will be an integer greater than or equal to $-1$. The tag value of a leaf will always be greater than or equal to zero. We define the relaxed height of a node $u$ by $rh(u) = tag(u)$, if $u$ is a leaf, and $rh(u) = \max(rh(u_l), rh(u_r)) + 1 + tag(u)$, otherwise.

In analogy with the balance factor, the relaxed balance factor is defined as $rbf(u) = rh(u_l) - rh(u_r)$. A tree is now an AVL tree with relaxed balance (a relaxed AVL tree, for short) if for every internal node $u$, $rbf(u) \in \{-1, 0, 1\}$. Clearly, if all tags have the value zero, then the tree is an AVL tree.

We describe the operations which can be performed on relaxed AVL trees. The operations are given in the appendix. We do not list symmetric cases. In the appendix, relaxed balance factors are shown to the left of nodes and tag values to the right. For notational convenience, we use $l(v)$ and $r(v)$, where $v$ is an internal node, letting $l(v) = 1$ if the relaxed height of the left child of $v$ is larger than the relaxed height of the right child, and $l(v) = 0$ otherwise. We use $r(v)$ similarly. In other words, $l(v) = 1$ iff $rbf(v) = 1$, and $r(v) = 1$ iff $rbf(v) = -1$. They equal zero otherwise.

Searching can be carried out exactly as for standard leaf-oriented AVL trees. The same is true for insertion and deletion (see the appendix, operations 1 and 2), except that tag values are adjusted.

The purpose of the rebalancing operations is to modify a relaxed AVL tree until it is a standard AVL tree and, thus, guaranteed to be balanced. Obviously, this is done by eliminating nonzero tag values. The difficulty is to do this without violating the relaxed balance constraint that $rbf(u) \in \{-1, 0, 1\}$ for all $u$.

All the operations are local, i.e., before an operation is performed, only a small number of nodes must be locked. Any standard locking scheme can be used here. For example, the simple solution proposed in [7].

The rebalancing is carried out as follows. First, a problem is found, i.e., a node with nonzero tag value. The problem can be found by rebalancing processes randomly traversing the data structure as suggested in [7] or could, more efficiently, be taken from a problem queue. When a problem is found, rebalancing can be performed provided that the tag value of the parent node is at least zero (appendix, operations 3 and 4). Notice that in [8], $t_u$ must be zero, whereas we only require that $t_u$ be different from $-1$, thus allowing for a significantly higher degree of concurrency. If one of the two children of $u$ has tag value $-1$ and the other has tag value greater than zero, then we require that the negative tag value be taken care of first.

Applying operation 3 or 4 has the effect of either removing a problem (changing a $-1$ to 0, or decreasing a positive tag value), or moving it closer to the root. At the root, problems disappear as the tag value of the root can always be set to zero without violating the relaxed balance constraint. However, operations 3 and 4 might violate the constraints since the relaxed balance factor of the top node can change to 2 or $-2$. So, before the locks are released, this is checked. In case of a violation after having applied operation 3, one of the operations 5 to 8 are performed. If operation 4 is the cause of a violation, then we consider the node $w$, the sibling of $v$. Because of the requirement that negative values be taken care of first, we can assume that $t_w \geq 0$. If $t_w > 0$, then operation 9 is applied. If $t_w = 0$, then one of the operations 10 to 13 are applied. Only after this has been taken care of, are the locks released.

It is an easy exercise to check that the rebalancing operations will not violate the relaxed balance constraint. We merely state the result here.

**Lemma 1** *Assume that one of the operations 3 or 4 are applied to a relaxed AVL tree. If the relaxed balance factor of $u$ (appendix) changes to 2 or $-2$, if one*

*of the operations 5 to 13 are applied before locks are released, then the tree is still a relaxed AVL tree.*

*Additionally, insertion and deletion (operations 1 and 2) cannot be the cause of a violation of the relaxed balance constraint.*

One particular implication of this lemma which will be used later is the following.

**Corollary 2** *Operations leave the relaxed height of the top node involved in an operation unchanged.*

Finally, notice that if the tree has tag values which are nonzero, then one of the operations 3 and 4 can be applied. To see this, consider the path from the root down to a node with nonzero tag value. On this path, let $v$ be the first node with nonzero tag value. The tag value of the root is always zero, so $v$ is not the root. Thus, one of the operations 3 and 4 can be applied to $v$ and its parent.

In the next section, we bound how many times operations 3 and 4 can be applied. So, if updating stops at some point, then the tree will eventually become a standard AVL tree. As the next section will show, this will happen quickly.

## 3 Complexity

For the complexity analysis, we follow [8] in assuming that initially the search tree is an AVL tree, and then a series of search, insert, and delete operations occur. These operations may be interspersed with rebalancing operations. The rebalancing operations may also occur after all of the search and update operations have been completed; our results are independent of the order in which the operations occur. In any case, the search tree is always a relaxed AVL tree, and after enough rebalancing operations, it will again be an AVL tree. We need only bound the number of times that operations 3 and 4 are applied, as the operations 5 to 13 are only used as an immediate consequence of operations 3 or 4 giving rise to a violation of the relaxed balance constraint. So, the operations 5 to 13 are considered to be a part of the operation which gave rise to its application.

If some of the operations are done in parallel, they must involve edges and nodes which are completely disjoint from each other. The effect will be exactly the same as if they were done sequentially, in any order. Thus throughout the proofs, we will assume that the operations are done sequentially. At time 0, there is an AVL tree, at time 1 the first operation has just occurred, at time 2 the second operation has just occurred, etc.

It is well known that the minimum size of AVL trees is closely related to the Fibonacci sequence. In the following, we use the definition of the Fibonacci sequence from [6]. Let $F_0 = 0$, $F_1 = 1$, and for $n \geq 0$, let $F_{n+2} = F_{n+1} + F_n$. The Fibonacci sequence is closely related to the golden ratio.

**Proposition 3** *Let $\phi = \frac{1+\sqrt{5}}{2}$ and $\hat{\phi} = 1 - \phi$. Then $F_n = \frac{1}{\sqrt{5}}(\phi^n - (\hat{\phi})^n)$. Furthermore, $\frac{\phi^n}{\sqrt{5}}$ rounded to the nearest integer equals $F_n$.*

**Proof** See [6]. □

Clearly, if a node $u$ in an AVL tree has a large height, then the subtree in which $u$ is the root will also be large. In a relaxed AVL tree, however, a node could have a large relaxed height because many nodes below it have been deleted and have caused tag values to increase. In this case, $u$ may not have a large subtree remaining. It will be useful to somehow count those nodes below $u$ which have been deleted. Nodes are inserted and deleted and we want to associate every node which has ever existed with nodes currently in the tree.

**Definition 4** *Any node in the tree at time $t$ is associated with itself. When a node is deleted, the two nodes which disappear, and all of their associated nodes, will be associated with the node which remains, i.e., the sibling of the node to be deleted (this is node $v$ in the appendix, operation 2).*

Thus, every node that was ever in the tree is associated with exactly one node which is currently in the tree.

**Definition 5** *Define an A-subtree (associated subtree) of a node $u$ at a particular time $t$ to be the set of all the nodes associated with any of the nodes in the subtree with root $u$ at time $t$.*

We can now prove a strong connection between relaxed heights and A-subtrees.

**Lemma 6** *At time $t$, if $u$ is a node in the tree, then there are $F_{rh(u)+3} - 1$ nodes in the A-subtree of $u$.*

**Proof** By induction in $t$.

The base case is when $t = 0$. At that point, the tree is a standard AVL tree and $rh(u) = h(u)$. It is a standard result for AVL trees that the minimal tree of height $h(u)$ has $F_{h(u)+3} - 1$ nodes.

For the induction, assume that the lemma holds up until time $t$, where $t \geq 0$. By checking all the

operations, we prove that the lemma also holds at time $t+1$.

When an insertion takes place, two new nodes are added. These are given the tag value zero, which results in a relaxed height of zero. As $F_3 - 1 = 1$, the result holds here. The relaxed height of the top node is maintained while the size of its A-subtree is increased, so the result holds for that node too.

When a deletion occurs, two nodes are deleted. However, they remain in the A-subtree of the parent node. So, the size of the A-subtree of $u$ is unchanged as, by corollary 2, is $rh(u)$.

For the remaining operations, we first make some general observations. Notice first that the number of nodes is always preserved. So, by corollary 2, the result always holds for the top node. Additionally, if the tag value of a node is decreased while its subtree remains unchanged (or has its size increased), then the lemma cannot fail for that node either.

We have dealt with the top nodes and the following argument will take care of the major part of the remaining nodes. It turns out that a node which is given the tag value zero and which gets two subtrees that were not changed in the operation cannot make the lemma fail. To see this, assume that $v$ is the node in question and that it has children $v_l$ and $v_r$. Assume without loss of generality that $rh(v_l) \geq rh(v_r)$. Then $rh(v_l) = rh(v) - 1$. As we need not consider the top nodes, we know that the relaxed balance factor of $v$ belongs to $\{-1, 0, 1\}$, so we can assume that $rh(v_r) \geq rh(v) - 2$. The number of nodes in the A-subtree of $v$ is now at least $(F_{rh(v_l)+3} - 1) + (F_{rh(v_r)+3} - 1) + 1 \geq F_{rh(v)+2} + F_{rh(v)+1} - 1 = F_{rh(v)+3} - 1$.

The $v$ node in operation 6 and the $w$ node in operation 11 are the only nodes which have not been covered by one of the cases in the above. However, their relaxed heights are decreased while their A-subtrees remain unchanged. □

**Corollary 7** *The relaxed height of any node at any time is at most* $\lfloor \log_\phi(N + \frac{3}{2}) + \log_\phi(\sqrt{5}) - 3 \rfloor$.

**Proof** Notice first that as tag values cannot be smaller than $-1$, no relaxed height can be larger than the relaxed height of the root.

At any time, the number of nodes in the A-subtree of the root $r$ is bounded by $N$. As, by lemma 6, there are at least $F_{rh(r)+3} - 1$ nodes in the A-subtree of $r$, the inequality $F_{rh(r)+3} - 1 \leq N$ must hold. By proposition 3, $F_n \geq \frac{\phi^n}{\sqrt{5}} - \frac{1}{2}$, so $\phi^{rh(r)+3} \leq \sqrt{5}(N + \frac{3}{2})$, which implies that $rh(r) \leq \log_\phi(\sqrt{5}(N + \frac{3}{2})) - 3$. The result follows from the fact that $rh(r)$ must be an integer. □

The values of $\phi$ and $\log_\phi(\sqrt{5})$ are approximately 1.618 and 1.672, respectively.

In order to bound the number of operations which can occur, it is useful to look at a positive tag value $t_u$ as consisting of $t_u$ positive units. Likewise, a negative tag value will be referred to as a negative unit. We can now, for the purpose of the complexity proof, assume that units have identities such that they can be followed around in tree from the moment they are created until they disappear. If, as the effect of an operation, a negative or positive unit is deleted from some node only to be introduced at another, we say that the unit has been moved.

**Proposition 8** *An insertion can create at most one extra negative unit and a deletion can create at most one extra positive unit. No other operation creates positive or negative units.*

**Proof** The claim for insertion is trivially fulfilled. For deletion, observe that in order to make the tag value negative, $t_u$ and $t_v$ must both be $-1$, in which case, we actually get rid of at least one negative unit. The question is whether a deletion can create more than one positive unit. Clearly, this will happen if and only if $t_u \geq 0$, $t_v \geq 0$, $r(u) = 1$, and $t_w = 0$ (if $t_w > 0$, then we would loose at least one positive unit when deleting $w$). Now, $t_w = 0$ implies that $rh(w) = 0$, so as $rh(u) = 1$, we have that $rh(v) = -1$. But that is impossible since the relaxed height of a leaf is at least zero and the relaxed height of an internal node is at least the relaxed height of its children.

In the operations 3, 4, 6, 8, 9, 11, and 13, negative or positive units are either moved or they disappear.

It could appear that operations 5 and 7 might introduce a new positive unit at the top node. However, these operations are only applied immediately after operation 3 and only in the case where $b_u$ becomes 2. But in that case, $r(u) = 0$, so a negative unit is moved to the top node $u$. Adding one to the tag value of $u$ in an immediately succeeding operation will simply make that negative unit disappear.

Likewise, operations 10 and 12 might increase the tag value of the top node. However, these operations are only applied immediately after operation 4 in case $b_u$ becomes $-2$. In that case, $l(u) = 0$, so a positive unit disappeared at first, but then if operation 10 or 12 are applied, it may be added again, and we simply consider it to have moved. □

We now prove that whenever a positive or negative unit is moved to another node, then the relaxed height of the new node will be larger than the relaxed height

of the old node. Because there is a limit to how large relaxed heights can become, this is a crucial step towards limiting the number of times operations can be applied.

**Lemma 9** *When a negative unit is involved in operation 3, it either disappears or it is moved to a node with a larger relaxed height. Additionally, no operation will decrease the relaxed height of a node with tag value* $-1$.

**Proof** Consider operation 3 and assume that the negative unit does not disappear. As $t_u \geq 0$, clearly $rh(u)$ before the operation is larger than $rh(v)$. By corollary 2, the negative unit is moved to a node with larger relaxed height. It might be necessary to apply one of the operations 5 to 8 afterwards, but again by corollary 2, they preserve the relaxed height of the top node.

For the remaining operations, insertion does not involve an existing negative unit, and a deletion removes negative units unless $t_u = t_v = -1$, in which case one disappears and the other is associated with the node $v$ with relaxed height $rh(u)$.

The remaining operations may involve a negative unit associated with the top node, but this unit will also be associated with the top node after the operation, which, by corollary 2, is safe. Finally, operations 8 and 13 involve a negative unit $tag(w)$ and $tag(x)$, respectively, but this unit disappears. □

**Lemma 10** *When a positive unit is involved in operation 4, it either disappears or it is moved to a node with larger relaxed height. Additionally, if some other operation decreases the relaxed height of a node by decreasing the tag value, then it decreases the tag value by at least the same amount.*

**Proof** Consider operation 4 and assume that the positive unit does not disappear. As $t_u \geq 0$, clearly $rh(u)$ before the operation is larger than $rh(v)$. By corollary 2, the positive unit is moved to a node with larger relaxed height. It might be necessary to apply one of the operations 9 to 13 afterwards, but again by corollary 2, they preserve the relaxed height of interest here.

For the remaining operations, insertion may remove a positive unit, while a deletion might move the positive units $t_v$ to a node with larger relaxed height.

The remaining operations may involve a positive unit associated with the top node, but this unit will also be associated with the top node after the operation, which, by corollary 2, is safe. Finally, the operations 6, 9, and 11 move a positive unit to a node with larger relaxed height.

Operation 9 is the only operation which decreases the relaxed height of a node with positive tag value. However, the tag value of that node is decreased by the same amount. □

At this point, we have all the partial results that will enable us to prove the main theorem.

**Theorem 11** *Assume that an AVL tree $T$ initially has $|T|$ nodes. Furthermore, assume that $k$ insertions, $m$ deletions, and a number of rebalancing operations are performed. If $N = |T| + 2k$, then the number of rebalancing operations is at most*

$$(k+m)\lfloor \log_\phi(N+\tfrac{3}{2}) + \log_\phi(\sqrt{5}) - 3 \rfloor - m.$$

**Proof** If an insertion creates a negative unit, then this unit is associated with a node, which has relaxed height zero. From lemma 9, we know that unless the negative unit disappears, it is associated with a node of larger relaxed height every time operation 3 has been applied involving this unit. From corollary 7, it therefore follows that a particular negative unit can be moved by operation 3 at most $\lfloor \log_\phi(N+\frac{3}{2}) + \log_\phi(\sqrt{5}) - 3 \rfloor$ times. As operation 3 can only be applied if a negative unit is involved, it follows from proposition 8 that operation 3 can be applied at most $k\lfloor \log_\phi(N+\frac{3}{2}) + \log_\phi(\sqrt{5}) - 3 \rfloor$ times.

The proof for deletion is similar (lemma 10 is used instead of lemma 9). However, when a positive unit is created, it is associated with a node of relaxed height at least one, so the bound becomes $m(\lfloor \log_\phi(N+\frac{3}{2}) + \log_\phi(\sqrt{5}) - 4 \rfloor)$.

The theorem follows from adding together the bounds for the operations 3 and 4. □

## Acknowledgements

The author would like to thank Joan Boyar for helpful comments on an earlier draft of the paper.

## References

[1] G. M. Adel'son-Vel'skiĭ and E. M. Landis, "An Algorithm for the Organisation of Information", *Dokl. Akad. Nauk SSSR*, 146, 263-266, 1962.

[2] R. Bayer, "Symmetric Binary B-Trees: Data Structure and Maintenance Algorithms", *Acta Inform.*, 1, 290-306, 1972.

[3] J. F. Boyar and K. S. Larsen, "Efficient Rebalancing of Chromatic Search Trees", *Lecture Notes in Computer Science*, 621: SWAT, 151-164, 1992.

[4] L. J. Guibas and R. Sedgewick, "A Dichromatic Framework for Balanced Trees", *19th IEEE FOCS*, 8-21, 1978.

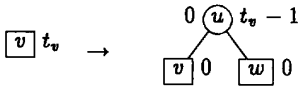
[5] J. L. W. Kessels, "On-the-Fly Optimization of Data Structures", *Comm. ACM*, 26, 895-901, 1983.

[6] D. E. Knuth, *Fundamental Algorithms*, Vol. 1 of The Art of Computer Programming, Addison-Wesley, 1968.

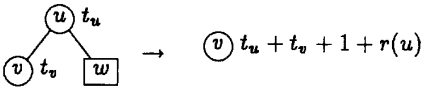
[7] O. Nurmi and E. Soisalon-Soininen, "Uncoupling Updating and Rebalancing in Chromatic Binary Search Trees", *ACM PODS*, 192-198, 1991.

[8] O. Nurmi, E. Soisalon-Soininen, and D. Wood, "Concurrency Control in Database Structures with Relaxed Balance", *ACM PODS*, 170-176, 1987.

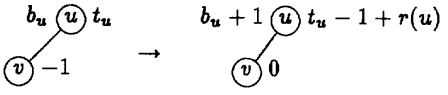
## Appendix



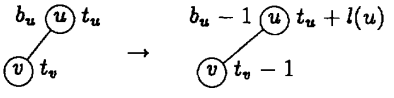

1) Insertion of $w$ to the right of $v$.



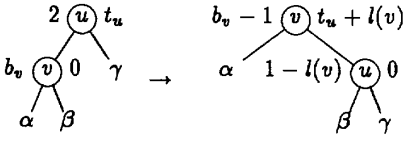

2) Deletion of $w$.



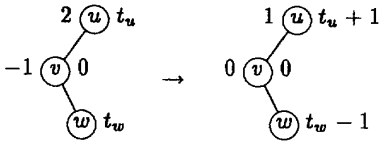

3) Moving up negative tags. Cond: $t_u \geq 0$.



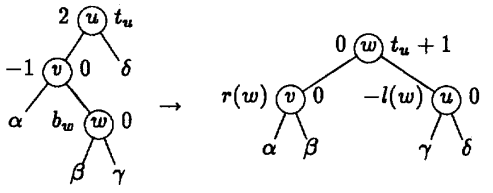

4) Moving up positive tags. Cond: $t_u \geq 0$, $t_v > 0$.



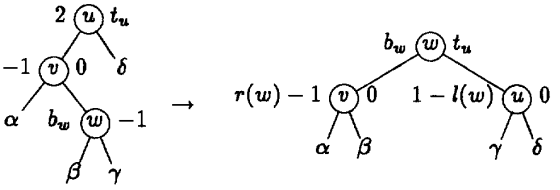

5) Too large rbf: single rotation. Cond: $b_v \geq 0$.



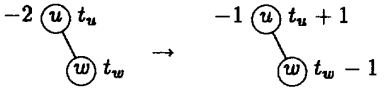

6) Too large rbf: moving tags. Cond: $t_w > 0$.



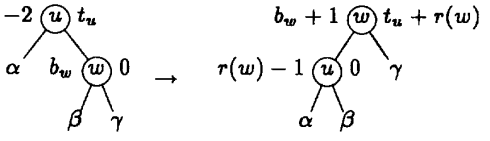

7) Too large rbf: double rotation ($t_w = 0$).



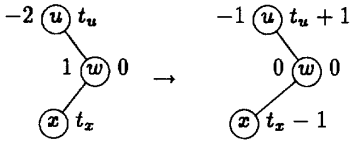

8) Too large rbf: double rotation ($t_w = -1$).



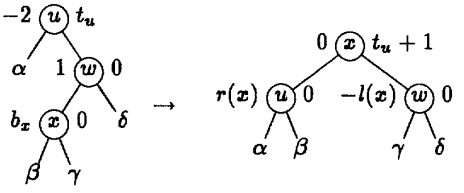

9) Too small rbf: moving tags. Cond: $t_w > 0$.



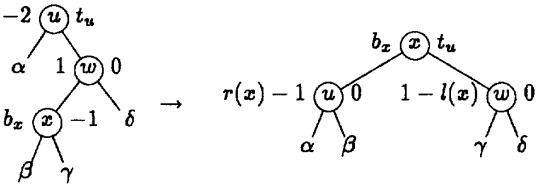

10) Too small rbf: single rotation. Cond: $b_w \leq 0$.




11) Too small rbf: moving tags. Cond: $t_x > 0$.




12) Too small rbf: double rotation ($t_x = 0$).




13) Too small rbf: double rotation ($t_x = -1$).