Where should concurrent rotations take place to rebalance a distributed arbitrary search tree?

Luc Bougé
Joaquim Gabarró
Xavier Messeguer

Report LSI-95-57-R

Where should concurrent rotations take place to rebalance a distributed arbitrary search tree?

Luc Bougé*, Joaquim Gabarró†, Xavier Messeguer†

December 1, 1995

Abstract

We address the concurrent insertion and deletion of keys in binary almost balanced search trees (AVL trees). We show that this problem can be studied through the self-reorganization of distributed systems of processes controlled by local evolution rules in the line of the approach of Dijkstra and Scholten. In particular, we show that our approach encapsulates a number of previous attempts described in the literature. This solves in a very general setting an old question raised by H.T. Kung and P.L. Lehman: where should rotations take place to rebalance arbitrary trees?

Keywords: *Distributed algorithms, Search trees, AVL algorithm, Concurrent generalized rotations, Safety and liveness proofs.*

1 Introduction

AVL trees are among the first topics taught to freshmen in the algorithmic course. Since their introduction by Adel'son-Velskĭĭ and Landis [1, 6], they have been recognized as a major source of inspiration for everything connected to sorting and searching.

The main drawback of the original presentation is to be... sequential! Keys are inserted one after the other, and each insertion is composed of two phases: 1) *Percolation*, where the value to be inserted moves downwards within the tree to reach its right place; 2) *Balancing*, where the tree is recursively restructured, starting from the new nodes upwards. Many authors, as R. Bayer and M. Schkolnick [2] and C.S. Ellis [4], have found *concurrent* versions of this scheme. Unfortunately, these attempts have often resulted in complex descriptions and the number of subtle details to be mastered is actually so large that proving correctness becomes hardly possible.

The goal of this paper is to show that several of these algorithms derive in fact from a single basic framework by specializing of the nondeterministic evolution strategy. In such a description, the control is kept as non-deterministic as possible. Any rule can be selected and applied to the global structure in any order as soon as its guard is satisfied. The rules assume: *temporal atomicity* (an

*LIP, ENS Lyon, 46 Allée d'Italie, F-69364 Lyon Cedex 07, France. This work has been partly supported by the French CNRS Coordinated Research Program on Parallelism, Networks and Systems PRS and HCM under contract ERBCHGECT920009.

†LSI, Universitat Politécnica de Catalunya, FIB/FME, C/ Pau Gargallo, 5, E-Barcelona 08028, Spain. Partially supported by the ESPRIT BRA Program of the EC under contract no. 7141, project ALCOM II. Part of this work has been done while Joaquim Gabarró was a Visiting Professor at ENS Lyon in 1995.

‡Authors contact: Joaquim Gabarró. Internet: gabarro@lsi.upc.es

action should correspond to a fixed, small number of assignments and tests) and *spatial atomicity* (an action should necessitate the exclusive access to a fixed, small set of neighboring nodes). The correctness can be derived from a small number of invariants. The *safety* property expresses that, if no rule can apply, then a satisfactory final state has been reached. The *liveness* property expresses that eventually no rule applies. The *independence* property expresses that rules with disjoint support commute: they may safely be executed concurrently.

This atomic approach has been first undertaken by H.T. Kung and P.L. Lehman [7]. They give safe concurrent algorithms to deal with insertions and deletions in binary search trees. As they do not wish to restrict to any specific type of balanced tree such as the AVL tree, deciding where rotation should take place is not specified. In a footnote, they suggest to attack this problem. Later on, inspired by on-the-fly garbage collection algorithms [3], J.L.W. Kessels [5] found the first safe and live[4] concurrent insertion algorithm for AVL trees based on atomic rules only. The question about rotations is solved as follows: one rotates (if necessary) just after a carry propagation in order to maintain the balance. Later on, O. Nurmi, E. Soisalon-Soininen and D. Wood [9, 10] extended Kessels' work by giving safe and fair algorithms to deal with insertions and deletions in external AVL (i.e., the actual keys are stored only in the leaves of the tree). Finally, K.S. Larsen [8] modified the rules given in [9] in order to increase the degree of concurrency and to study the complexity of the rebalancing.

In our approach, we answer the question of Kung and Lehman about rotations as follows: a rotation *can* just take place at any unbalanced node. Therefore, when no more rotations are possible, then we have an AVL. This approach is intuitively clear but working out the technical details is far from obvious. For instance, if rotations are separated from propagations, then the local information can be arbitrarily unfaithful, and one may decide to rotate the wrong subtree. Liveness is questionable.

2 Self-balancing distributed search trees

2.1 General description

We consider binary trees whose nodes are labeled by integer (possibly non-distinct) keys. Such a tree is a *search* tree if the keys appear in sorted order in a depth-first, left-right traversal. Nodes are ranged by n, p, q, etc. The key stored at node n is denoted $\mathsf{key}(n)$. The *(real) height* of a node is the height of the subtree rooted at this node. The height of a leave is 1. The height of an empty tree is 0. A tree is *balanced* if for all node n, the height of its two subtrees differ at most by 1. We consider such a tree as a distributed system, where the processes are the nodes and the links are the father/son edges. Each node n is thus equipped with a number of *private registers* used to store the local information it holds about the global system:

$\mathsf{lefth}(n)$: The apparent height of the left son of n, at the best of its knowledge;

$\mathsf{righth}(n)$: The apparent height of the right son of n at the best of its knowledge.

We define three additional quantities:

[4] To get a fully correct proof of liveness, the variant function given by Kessels need to be slightly corrected along the lines of this paper (private communication with Joaquim Gabarró, May 1995).

lheight(n): The apparent local height of node n, at the best of its knowledge.

$$\mathsf{lheight}(n) = \max(\mathsf{righth}(n), \mathsf{lefth}(n)) + 1$$

bal(n): The apparent balance of node n, at the best of its knowledge.

$$\mathsf{bal}(n) = \mathsf{righth}(n) - \mathsf{lefth}(n)$$

delta(n): The difference between the height known by the father m of node n and the current apparent height of node n, at the best of its knowledge.

$$\mathsf{delta}(n) = \begin{cases} \mathsf{lefth}(m) - \mathsf{lheight}(n) & \text{if } n \text{ is the left son of } m \\ \mathsf{righth}(m) - \mathsf{lheight}(n) & \text{if } n \text{ is the right son of } m \end{cases}$$

By convention, we set $\mathsf{delta}(n) = 0$ if n is the root of the tree.

In the following sections we present a safe and live distributed algorithm to update AVL trees. It can be described easily by a set of evolution rules, in the style of the famous Dijkstra and Scholten's distributed termination algorithm. Any rule can be selected and applied in any order as soon as its guards are satisfied. The application of a rule modifies the values stored into the local registers. Observe that these quantities may be arbitrarily different from their *real* values. We do not try to define accurately what we mean by *real*. Informally, to get the *(real) height* or *(real) balance* we freeze the tree and we compute these values as usual. Whenever a local register is updated with the information sent to it (at some preceding moment) we call this information *apparent*. Of course, the value of the apparent information can be very different from the value of the real information. We say that node n is *(apparently) balanced* if $\mathsf{bal}(n)$ is 0, 1 or -1. Node n is apparently balanced if it is so according to the information transmitted to it by its sons at some preceding moments. As the information can be delayed by some of the sons, $\mathsf{bal}(n)$ roughly reflects the reality. Whenever $\mathsf{bal}(n) \notin \{-1, 0, 1\}$, we say node n is *(apparently) unbalanced*.

Let us say what we mean by *(apparently) faithful* information or *(apparent) stability*. Intuitively a node n has apparent faithful information if it is correct from the point of view of its neighbors. The quantity $\mathsf{delta}(n)$ serves to measure the degree of faithfulness. If needed, it can be considered as a new internal register of n. In this case, n knows when its father m has faithful information about $\mathsf{lheight}(n)$. If n is a left son of m, then node m has faithful information about $\mathsf{lheight}(n)$ whenever $\mathsf{lheight}(n) = \mathsf{lefth}(m)$ (that means $\mathsf{delta}(n) = 0$) otherwise it has unfaithful information. When $\mathsf{delta}(n) = 0$, node n does not need to send any new information to update the values of m. We say node n is *(apparently) stable*. Otherwise, $\mathsf{delta}(n) \neq 0$, and some useful information is still being held in n: we say n is *(apparently) unstable*.

As we will see later, a lot of unfaithful information may be generated along the whole tree. In order to guarantee a correct final result, we need to anchor the correct values of the local height at some nodes. A distributed search tree is *well-founded* if $\mathsf{lefth}(n) = 0$ (resp. $\mathsf{righth}(n) = 0$) for any node n with an empty left (resp. right) node. This means that (at least) the border nodes have an accurate knowledge about their height. We have the following easy result.

Lemma 1 *Let T be a distributed, well-founded search tree.*

- *If all nodes have faithful information, that is, are (apparently) stable, then the local information coincides with the real information. For all n we have* $\mathsf{lheight}(n) = \mathsf{real\text{-}height}(n)$ *and* $\mathsf{bal}(n) = \mathsf{real\text{-}balance}(n)$.

- *If 1) all nodes have faithful information, are (apparently) stable, and 2) all nodes are (apparently) balanced, then the tree is (really) balanced. It is thus an AVL in the usual sense.*

Proof *As* all *nodes are* apparently *stable, the* apparent *value* lefth(n) *and* righth(n) *are just the* real *heights of the sons of any node n: is it true for empty sons, as the tree is well-founded, and for all sons by induction. As the nodes are* apparently *balanced, they are* really *balanced.*

We can now state our basic framework. Consider a distributed, well-founded search tree T. We aim at transforming T through a sequence of atomic evolution rules into a tree T' with the following properties. (1) T' holds the same keys as T, also in sorted order, and T' is well-founded; (2) All nodes of T' are (apparently) stable and (apparently) balanced. By the lemma above, T' is an AVL, as wanted. Observe that we make absolutely no assumption on the initial shape of T and the knowledge of its internal nodes. In this sense, the behavior is *self-balancing*, very much in the same sense as for self-stabilizing distributed algorithms.

2.2 The rules

The algorithm should eventually *yield an AVL* in spite of *unfaithful information.* Two problems have to be faced. First, as the information can be delayed at any descendent m of n (we mean delta(m) $\neq 0$), node n has a very rough knowledge of its real height and balance. We must allow information to flow upwards (our tree is well-founded) to get faithful information. Lemma 1 states that local information coincides with real information when all information is faithful. Second, we have to end up with an AVL. Every node should thus try to improve the situation by doing rotations. To take a decision, node n can only access local information. A rotation around n may seem very good from the local point of view whereas it actually worsens the global situation. This leads to two kinds of evolution rules:

Propagation Rule: It propagates information upwards from a son to his father. Applying this rule increases the quality of the father's knowledge, that is the *global stability.*

Rotation Rules: They are a generalization of the single and double rotation rules of the sequential AVL algorithm. Applying this rule increases the *global balance* of the tree. In contrast with the original AVL algorithm, we have no control on rule application: we can no longer assume that the balance of a node to be rotated is 2 or -2. It may happen that the imbalance of a node suddenly reaches large values.

For the sake of clarity, we adopt the following notation: $n(A, B)$ denotes the (sub)tree with root n, left son A and right son B (sons may possibly be empty). Also, we present only the leftwise versions of the rules. The rightwise version can be obtained by symmetry. As a convention, the final states of a node n is denoted n'. Unless specified, it is identical to the initial state. We name the rotation sub-rules after the position of the subtrees with dominant heights. Let us start with a rule allowing to any son n to update the information of its father concerning its apparent height.

Rule 1 (Propagation)

Guard: *A subtree* $m(n(A, B), C)$ *with* delta(n) $\neq 0$ *(Figure 1).*

Behavior: *Update m into* m' *with*

$$\text{lefth}(m') := \text{lheight}(n)$$

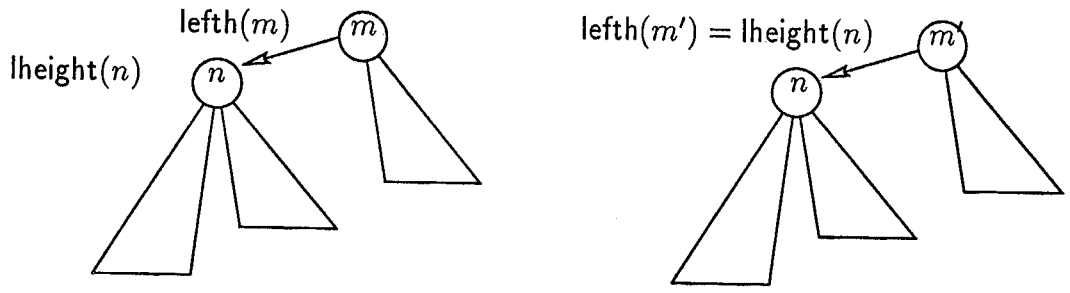

Figure 1: Propagation rule, left case

Spatial scope: *Node n and its father m.*

Note: *Symmetrically if n is the right son of m.*

We can easily prove that applying repeatedly the propagation rule to a well-founded tree will eventually correct the local values stored into the nodes. At the end, local information will coincide with the real information. Therefore, the Propagation Rule lets the correct information flow up. Now, we consider rotations to improve the balance of the tree. First of all, it makes no sense to require a locally better tree if even local data are unfaithful. We therefore require the sons to be stable. We could think that as rotations equilibrate (at least locally) subtrees, they also make these subtrees smaller (at least with respect to the local height). Unfortunately this is not always the case. This leads us to split single rotations into two cases according to this fact. Is easy to see that the local height of the rotated subtree strictly decreases when $\mathsf{bal}(p) \neq 0$, and it remains constant when $\mathsf{bal}(p) = 0$.

Rule 2 (Single rotation, left-left case)

Guard: *A subtree $n(p(A,B),C)$, $\mathsf{bal}(n) \leq -2$, p is stable, and $\mathsf{bal}(p) < 0$ (Figure 2).*

Behavior: *Restructure the tree into $p'(A, n'(B,C))$ with the obvious updating of the registers:*

$\mathsf{key}(n') := \mathsf{key}(n)$	$\mathsf{key}(p') := \mathsf{key}(p)$
$\mathsf{lefth}(n') := \mathsf{righth}(p)$	$\mathsf{lefth}(p') := \mathsf{lefth}(p)$
$\mathsf{righth}(n') := \mathsf{righth}(n)$	$\mathsf{righth}(p') := \mathsf{lheight}(n')$

Spatial scope: *Node n and its left son p.*

Note: *Symmetrically for the right-right case: $n(A, q(B,C))$, $\mathsf{bal}(n) \geq 2$, q is stable, and $\mathsf{bal}(q) > 0$.*

Rule 3 (Single rotation, left-equal case)

Guard: *A subtree $n(p(A,B),C)$, $\mathsf{bal}(n) \leq -2$, p is stable, and $\mathsf{bal}(p) = 0$ (Figure 3).*

Behavior: *Restructure the tree into $p'(A, n'(B,C))$ with the obvious updating of the registers:*

$\mathsf{key}(n') := \mathsf{key}(n)$	$\mathsf{key}(p') := \mathsf{key}(p)$
$\mathsf{lefth}(n') := \mathsf{righth}(p)$	$\mathsf{lefth}(p') := \mathsf{lefth}(p)$
$\mathsf{righth}(n') := \mathsf{righth}(n)$	$\mathsf{righth}(p') := \mathsf{lheight}(n')$

Spatial scope: *Node n and its left son p.*

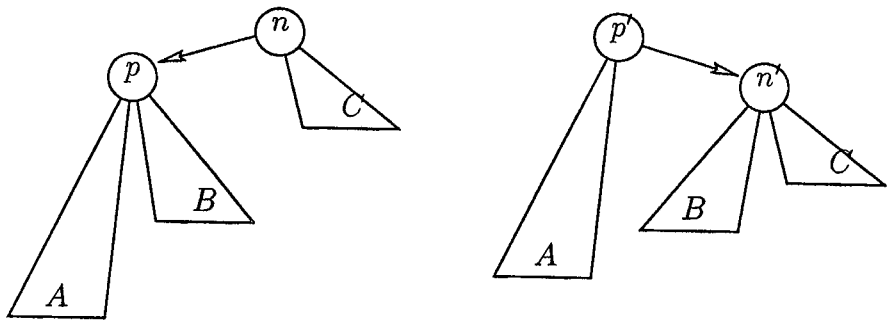

Figure 2: Single rotation rule, left-left case

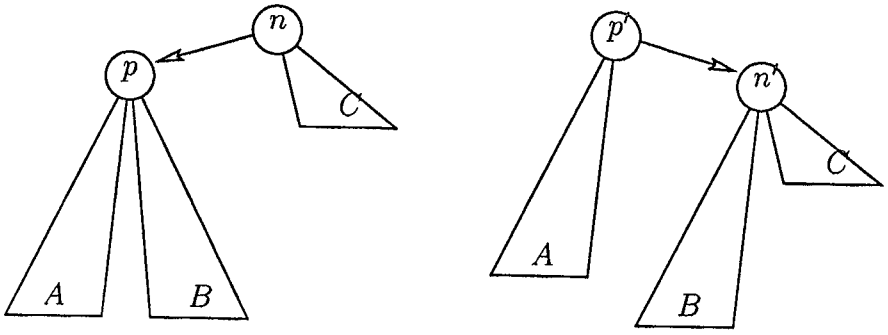

Figure 3: Single rotation rule, left-equal case

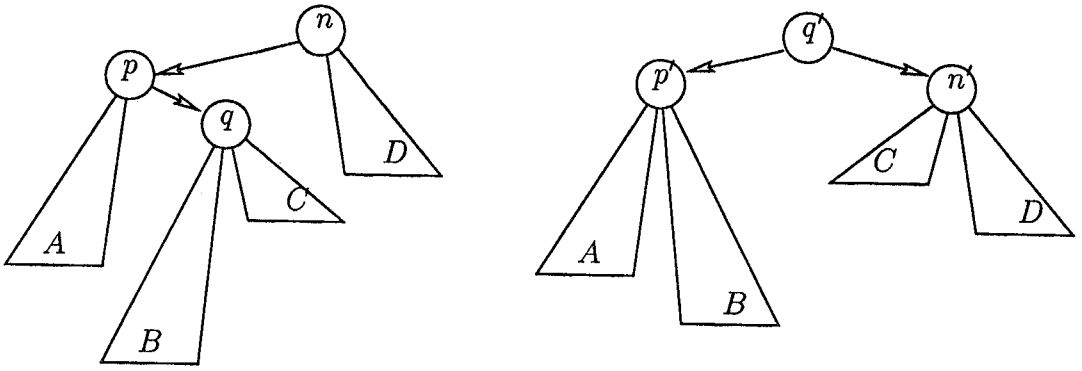

Figure 4: Double rotation rule, left-right case

Note: *Symmetrically for the right-equal case:* $n(A, q(B, C))$, $\mathsf{bal}(n) \geq 2$, q *is stable, and* $\mathsf{bal}(q) = 0$.

In contrast to single rotations (as we will see), double rotations always make the local height of the subtree to decrease.

Rule 4 (Double rotation, left-right case)

Guard: *A subtree* $n(p(A, q(B, C)), D)$, $\mathsf{bal}(n) \leq -2$, p *and* q *are stable,* $\mathsf{bal}(p) > 0$.

Behavior: *Restructure the tree into* $q'(p'(A, B), n'(C, D))$ *with the obvious updating of the registers:*

$\mathsf{key}(n') := \mathsf{key}(n)$	$\mathsf{key}(p') := \mathsf{key}(p)$	$\mathsf{key}(q') := \mathsf{key}(q)$
$\mathsf{lefth}(n') := \mathsf{righth}(q)$	$\mathsf{lefth}(p') := \mathsf{lefth}(p)$	$\mathsf{lefth}(q') := \mathsf{lheight}(p')$
$\mathsf{righth}(n') := \mathsf{righth}(n)$	$\mathsf{righth}(p') := \mathsf{lefth}(q)$	$\mathsf{righth}(q') := \mathsf{lheight}(n')$

Spatial scope: *Node* n, *its son* p *and* p*'s son* q.

Note: *Symmetrically for the right-left case:* $n(A, q(p(B, C), D))$, $\mathsf{bal}(n) \geq 2$, q *and* p *are stable, and* $\mathsf{bal}(q) < 0$.

We have decoupled the rules designed to improve the consistence of the local data from the rules aimed at equilibrating the tree. Propagation Rule deals with registers $\mathsf{delta}(n)$. These registers measure the faithfulness between local data. Rotation Rules work with quantities $\mathsf{bal}(n)$ measuring the imbalance.

2.3 Safety and Liveness

The Safety Property guarantees that, whenever we start with faithful information at the border nodes of a well-founded tree, any tree obtained through the evolution rules is fine.

Lemma 2 *Assume that* T *is a distributed, well-founded search tree. 1) Let* T' *be obtained by the application of some rule, then* T' *is a well-founded search tree, too. 2) Moreover, if no rule applies on* T*, as it is locally stable and locally balanced,* T *is an AVL.*

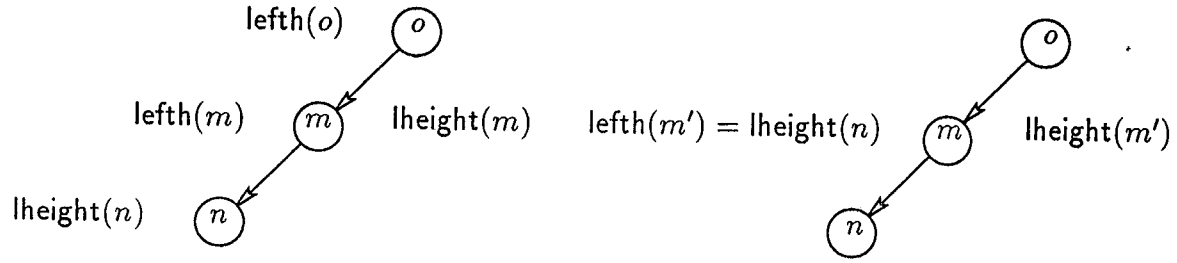

Figure 5: Propagation Rule, left case, 3-node configuration

It remains to prove the difficult part, that is, the evolution rules do not admit any infinite sequence of applications: *something good, e.g. termination, must eventually happen!* As mentioned above, Propagation and Rotation Rules can fight ones against others. Some oscillating behavior could result, where stability and balance alternatively improve. An in-depth study of their interaction is therefore necessary. First of all, observe that we now apply rotation rules to highly unbalanced nodes n (we mean $|\mathsf{bal}(n)| > 2$). The resulting tree is *not* balanced in general, in contrast with the usual AVL algorithm. Unless specified, all lemmas in this section are proved by easy but tedious case enumerations.

Lemma 3 (Evolution of lheight)

Single rotation rule, left-left case: $\mathsf{lheight}(p') = \mathsf{lheight}(n) - 1$.

Double rotation rule, left-right case: $\mathsf{lheight}(q') = \mathsf{lheight}(n) - 1$.

Single rotation Rule, left-equal case: $\mathsf{lheight}(p') = \mathsf{lheight}(n)$.

Thus, lheight may not increase on applying a rotation at node n, and it is left unchanged for all other nodes in the tree. Remark that rotations around n cannot decrease the stability of n (we mean $\mathsf{delta}(p') \geq \mathsf{delta}(n)$, where p is the new root). When $\mathsf{delta}(n) > 0$, we say that n *has positive unstability.* When $\mathsf{delta}(n) < 0$ we say that n *has negative unstability.* Intuitively,

$\mathsf{delta}(n) > 0$ $\equiv$ "The father of n believes that n is $\mathsf{delta}(n)$ units *higher* than n believes"
$\equiv$ "The father of n *overestimates* in $\mathsf{delta}(n)$ units the local height of n"
$\mathsf{delta}(n) < 0$ $\equiv$ "The father of n believes that n is $|\mathsf{delta}(n)|$ units *smaller* than n believes"
$\equiv$ "The father of n *underestimates* in $|\mathsf{delta}(n)|$ units the local height of n"

Say tree T is is *consistent* if $\mathsf{delta}(n) \geq 0$ for all nodes n. As rotations may not decrease unstability, they preserve consistency. It can be shown that propagations preserve consistency, too. This leads us to split our discussion in two parts.

Consistent trees The following result is crucial for analyzing the various cases.

Lemma 4 *None of the quantities* outside, delta, bal *is modified outside of the spatial scope of the rules.*

Although all nodes verify $\mathsf{delta}(n) \geq 0$, we underline that the following analysis only takes into account the local properties of the nodes.

Propagation Rule We consider the left case only (see Figure 5). Consider a node n, its father m and m's father o. By the precondition of the Propagation Rule, $\mathsf{lheight}(n) < \mathsf{lefth}(m)$. On propagating the height of n, one modifies $\mathsf{delta}(n)$, $\mathsf{lefth}(m)$, $\mathsf{lheight}(m)$ and $\mathsf{delta}(m)$. The global effect of a propagation on BAL and EXCESS depends therefore on the value of $\mathsf{delta}(m)$, and on the relative values of $\mathsf{lheight}(n) \leq \mathsf{lefth}(m)$ and $\mathsf{righth}(m)$. As the tree is consistent, we have $\mathsf{delta}(m) \geq 0$. The global positive unstability and the global unbalance progress in opposite directions. Remember that $\mathsf{lefth}(m') \leq \mathsf{lefth}(m)$ by hypothesis.

- If the left subtree was "very" dominant at m, then updating $\mathsf{lefth}(m)$ improves the balance $\mathsf{bal}(m)$ but decreases the height of m. Thus $\mathsf{delta}(m)$ increases of the same quantity as $\mathsf{delta}(n)$ decreases. Thus, BAL strictly decreases and EXCESS remains unchanged. Thus, TRADE_OFF strictly decreases.

- If the right subtree was dominant at m, then updating $\mathsf{lefth}(m)$ worsens the balance $\mathsf{bal}(m)$ by $\mathsf{delta}(n)$, and the leaves the height unchanged. Thus, BAL increases but EXCESS strictly decreases by the same quantity. Therefore TRADE_OFF strictly decreases.

- For intermediate cases, a detailed analysis shows that

$$\mathsf{BAL}' = \mathsf{BAL} + 2.\alpha - \mathsf{delta}(n) \qquad \mathsf{EXCESS}' = \mathsf{EXCESS} - \alpha$$

  for some α with $0 \leq \alpha \leq \mathsf{delta}(n)$, thus TRADE_OFF strictly decreases because $\mathsf{TRADE\_OFF}' = \mathsf{TRADE\_OFF} - \mathsf{delta}(n)$.

Rotation Rules A careful analysis with $\mathsf{delta}(n) \geq 0$ shows the following global effect.

- Left-left rotation: $\mathsf{TRADE\_OFF}' \leq \mathsf{TRADE\_OFF} - 1$ because $\mathsf{BAL}' \leq \mathsf{BAL} - 3$ and $\mathsf{EXCESS}' = \mathsf{EXCESS} + 1$.

- Left-equal rotation: $\mathsf{TRADE\_OFF}' = \mathsf{TRADE\_OFF}$ because $\mathsf{BAL}' = \mathsf{BAL}$ and $\mathsf{EXCESS}' = \mathsf{EXCESS}$.

- Left-right rotation: $\mathsf{TRADE\_OFF}' \leq \mathsf{TRADE\_OFF} - 1$ because $\mathsf{BAL}' \leq \mathsf{BAL} - 3$ and $\mathsf{EXCESS}' = \mathsf{EXCESS} + 1$.

We can sum up all the analysis above by the following property.

Property 1 (Variant function) *For any rule application, one of these 2 cases holds:*

1. TRADE_OFF *strictly decreases;*

2. TRADE_OFF *remain unchanged: this occurs only in the case of a left-equal rotation. In this case* RBAL *strictly decreases.*

Therefore, the variant function $(\mathsf{TRADE\_OFF}, \mathsf{RBAL})$ *strictly decreases for the lexicographic order on any rule application and it is is greater than* $(0, 0)$.

General trees We now relax the hypothesis of consistency. We only sketch the general analysis, which is a smooth extension of the one above.

Negative unstability: Recall that a node has negative unstability if its father underestimates its apparent height. Intuitively, *negative* unstability flows upwards in the tree and eventually reaches the root where it vanishes. Upwards flowing can be captured by weighting the lack of stability with a measure of its depth within the tree. Kessels [5] introduces the following quantity for this purpose: $\mathsf{outside}(n)$ is the number of nodes *not* in the subtree rooted at node n (it is 0 if n is the root). We define

$$\mathsf{LOSS} = \sum_{\mathsf{delta}(n)<0} \mathsf{outside}(n).|\,\mathsf{delta}(n)|$$

The tree is consistent if $\mathsf{LOSS} = 0$. This is thus a measure of the inconsistency of the tree. This last fact allows us to reduce the liveness of general trees to the liveness of consistent trees.

Property 2 (General Liveness) *The variant* $(\mathsf{LOSS}, \mathsf{TRADE\_OFF}, \mathsf{RBAL})$ *strictly decreases for the lexicographic order on any rule application and it is is greater than* $(0,0,0)$. *Therefore, no infinite sequence of rule applications is possible.*

It seems to be very difficult to give any general description of the successive "shapes" of the tree on its way towards balance. But we can at least give a bound on the register values as follows. Let $\mathsf{real\text{-}height}(n)$ be the *real height* of the subtree rooted at node n.

Lemma 5 *Let T be a distributed search tree such that* $\mathsf{lheight}(n) \leq \mathsf{real\text{-}height}(n)$ *for all nodes n in T. Let T' be obtained by any rule application. Then, this holds for T', too.*

This lemma could be used to give a very rough bound on the number of steps needed to get an AVL from any search tree with N nodes: just defines initially $\mathsf{lefth}(n) = \mathsf{righth}(n) = 0$ for all the nodes: it makes it a well-founded (but *not* consistent!) tree. At any point in the reshaping, we shall have: $\mathsf{lefth}(n), \mathsf{righth}(n) \leq \mathsf{lheight}(n) \leq \mathsf{real\text{-}height}(n) \leq N$ for any node.

3 Concurrent insertion and deletion of keys

Using this basic framework, we can describe an algorithm to manage the concurrent insertion and deletion of keys in distributed sorted tree.

Insertion The idea is to simulate the percolation of keys in the original sequential algorithm with a new register $\mathsf{waiting}(n)$ which holds the keys waiting at node n for downwards percolation. To handle the possibility of equal keys, $\mathsf{waiting}(n)$ is managed as a *bag*. Operation $+$ adds a key to the bag. Operation $-$ removes it. This register is called the *waiting bag* at n. As usual, $|\,\mathsf{waiting}(n)|$ denotes the number of keys held in bag $\mathsf{waiting}(n)$.

We say that a distributed, search tree is *strongly sorted* if the following condition holds: If n is in the left (resp. right) subtree of m, and $a \in \mathsf{waiting}(n)$, then $a \leq \mathsf{key}(m)$ (resp. $a \geq \mathsf{key}(m)$). As a simple example of such a tree, consider a single node n and N keys $a_1, \ldots, a_N$ with

$$\mathsf{key}(n) := a_1 \qquad \mathsf{waiting}(n) := \{a_2, \ldots, a_N\} \qquad \mathsf{lefth}(n) := 0 \qquad \mathsf{righth}(n) := 0$$

Rule 5 (Percolation)

Guard: *Node n, key $a \in \mathsf{waiting}(n)$, $a \leq \mathsf{key}(n)$.*

Behavior: *If n has a left son p, then*

$$\mathsf{waiting}(n) := \mathsf{waiting}(n) - a \qquad \mathsf{waiting}(p) := \mathsf{waiting}(p) + a$$

Otherwise, create a new node p, left son of n, with the following registers:

$$\mathsf{waiting}(n) := \mathsf{waiting}(n) - a \quad \mathsf{key}(p) := a \quad \mathsf{waiting}(p) := \emptyset \quad \mathsf{lefth}(p) := 0 \quad \mathsf{righth}(p) := 0$$

Spatial scope: *Node n and the potential new node p.*

Note: *Symmetrically with $a \geq \mathsf{key}(n)$ and node q the right son of n.*

It remains to refine the Propagation Rule and the Rotation Rules to take into account the waiting bags. The former causes no problem. But simply leaving the waiting bags hanging at the rotated nodes does not preserve strong sortedness in the latter. An easy patch is the following.

Rule 6 (Rotation with waiting bags)

Additional guard: *A Rotation Rule may be applied only if the waiting bags at the nodes in the spatial scope of the rule are all empty.*

It is easy to check that the restricted version of Rotation Rules preserves strong sortedness. It is also clear that the global effects on LOSS, BAL, EXCESS and RBAL are not altered.

Property 3 (Safety) *Let T be a distributed, well-founded and strongly sorted search tree. Assume no rule applies to T. Then, T is an AVL whose all waiting bags are empty.*

Lemma 6 *Let T be a distributed, well-founded and strongly sorted search tree. Let T' be the tree obtained by applying a Propagation Rule or a Rotation Rules. Then* $(\mathsf{LOSS}', \mathsf{TRADE\_OFF}', \mathsf{RBAL}') < (\mathsf{LOSS}, \mathsf{TRADE\_OFF}, \mathsf{RBAL})$.

To prove liveness, it remains to find an additional variant function to cover the two cases of Percolation Rule. We introduce the following quantity: $\mathsf{inside}(n)$ is the number of nodes in the subtree rooted at n. Define first

$$\mathsf{NUMBER} = \sum_n |\mathsf{waiting}(n)| \qquad \text{and} \qquad \mathsf{WAIT} = \sum_n \mathsf{inside}(n).|\mathsf{waiting}(n)|$$

Quantity NUMBER is the current number of waiting keys and WAIT is the current number of waiting keys, somehow weighted by their respective distances from the leaves. It is easy to check the that the first case of Percolation Rule lets NUMBER remain invariant and WAIT strictly decrease. Moreover, NUMBER and WAIT are invariant under the Propagation and Rotation rules (recall that a rotation may only take place if all bags in the spatial scope are empty). Also, the second case of Percolation Rule lets NUMBER strictly decrease.

Property 4 (Liveness)
The variant function (NUMBER, WAIT, LOSS, TRADE_OFF, RBAL) *strictly decreases on any rule application.*

Insertion and deletion We add the possibility to delete keys in a concurrent schema with the previous concurrent insertion algorithm. We start by letting the key to be deleted percolate down until the node with equal key is found and marked. Then, the node is sent down by successive rotations around it, until one of its sons, at least, gets empty. The node can then be safely removed from the tree.

The waiting bag contains now two kind of keys, the keys to be inserted and the keys to be deleted, which are differentiated by a flag. We need another flag, denoted alive(n), to mark the nodes to be rotated down and removed. We say that a node is *alive* if alive(n) = *true*, otherwise it is said *dead*.

Let $\{a_1, a_2, \ldots, a_N\}$ the keys to be inserted or deleted. We start by locating them into the waiting bag of the root of the tree and by setting to true all the flags alive. We modify the Percolation rule in order to consider the new kind of keys.

Rule 7 (Percolation)

Guard: *Node n, key $a \in$* waiting(n) *and should be deleted.*

Behavior: *a is extracted form* waiting(n). *If $a =$* key(n) *then* alive(n) *becomes false. Assume that $a <$* key(n) *and n has left son p, then* waiting(p) := waiting(p) + a, *but if n has no left son then a is removed. The case $a >$* key(n) *is handed in the same way.*

Spatial scope: *Nodes n and p.*

The next step then is to add a new rule, called *Garbage Percolation Rule*, to remove dead nodes. We only sketch the idea here. Dead-marked nodes are sent down to the border of tree by means of specific rotations, and there, they are eventually removed. These specific rotations basically shift away the root of the subtree to a neighbor. Observe that the order of the keys is left unchanged. Then we define a new variant quantity DEAD to measure how many dead nodes are still to be removed. We finally get the following result.

Theorem 1 (Total correctness) *The Propagation, Rotation, Percolation and Garbage Percolation Rules strictly decrease the function* (NUMBER, WAIT, DEAD, LOSS, TRADE_OFF, RBAL). *There does not exist any infinite sequence of rule applications. One eventually get an AVL.*

4 Discussion

Let us consider the emulation of some distributed algorithms [5, 9, 10, 8] based on local rules. To approach the problem we introduce the *up apparent height*, written up-height(n) of a node n, defined as up-height(n) = lheight(n) + delta(n). Given a node n having left son p and right son q we have up-height(p) = lefth(n) and up-height(q) = righth(n). This function behaves as an approximate height because it verifies up-height(n) = 1 + delta(n) + max{up-height(p), up-height(q)}. Remark that delta(n) $\in \{\ldots, -2, -1, 0, 1, 2, \ldots\}$. Moreover the balance can be rewritten as:

$$\text{bal}(n) = \text{up-height}(q) - \text{up-height}(p)$$

J.L.W. Kessels' approach Let us see how to emulate the algorithm [5]. We start giving a brief description of it. Every node n has the registers balance(n) $\in \{-1, 0, 1\}$, carry(n) $\in \{0, 1\}$ and there is a dynamic height defined as: dheight(n) = 1 − carry(n) + max{dheight(p), dheight(q)} Once all the keys have been inserted, the tree can be highly unbalanced and the algorithm starts a

restructuring process. Eventually, all nodes n satisfy $\mathsf{carry}(n) = 0$. Then, $\mathsf{real\text{-}height}(n) = \mathsf{dheight}(n)$ and the tree is an AVL. This process is described by the transformations (a), (b) and (c) (Figures 1, 2 and 3 in [5]). The comparison between $\mathsf{dheight}(n)$ and $\mathsf{up\text{-}height}(n)$ suggests us the identifications:

$$\mathsf{carry}(n) = -\mathsf{delta}(n) \qquad \mathsf{dheight}(n) = \mathsf{up\text{-}height}(n)$$

Let us see how to emulate Transformations (a), (b) and (c) with our rules. All the cases given in (a) can be emulated by our propagation. The cases appearing in (b) and (c) can be emulated concatenating a propagation with a rotation.

O. Nurmi et al.'s approach Now we consider the following extension of the J.L.W. Kessels' algorithm developed by O. Nurmi, E. Soisalon-Soininen and D. Wood in [10] (see also [9]). Every node holds the registers $\mathsf{rbalance}(n) \in \{-1, 0, 1\}$ and $\mathsf{tag}(n) \in \{-1, 0, 1, 2, \ldots\}$ and they define a relaxed height as:

$$\mathsf{rheight}(n) = 1 + \mathsf{tag}(n) + \max\{\mathsf{rheight}(p), \mathsf{rheight}(q)\}$$

The definitions of $\mathsf{rheight}(n)$ and $\mathsf{up\text{-}height}(n)$, suggest us the identifications:

$$\mathsf{rheight}(n) = \mathsf{up\text{-}height}(n) \qquad \mathsf{rbalance}(n) = \mathsf{bal}(n) \qquad \mathsf{tag}(n) = \mathsf{delta}(n)$$

A problem appears with these identifications. As in [10] tag values can only decrease in one unit, we are forced to have $\mathsf{delta}(n') = \mathsf{delta}(n) \pm 1$. However our Rule 1 allows node n to propagate all the information in only one step (for any value $\mathsf{delta}(n) \neq 0$ we have $\mathsf{delta}(n') = 0$). In order to accept small modifications like $\mathsf{delta}(n') = \mathsf{delta}(n) \pm 1$, we add a new private register $\mathsf{delta}(n)$. This register keeps the information to be transmitted to the father in order to update its knowledge about the height. The information about the height flows slowly along the tree and in a propagation we have $\mathsf{up\text{-}height}(n') = \mathsf{up\text{-}height}(n) \pm 1$. All our rules can be rewritten to take care of this fact.

K.L. Larsen's approach Finally let us consider the approach taken by K. L. Larsen [8]. Recall that, given a node n having a father m, the transformations given by O. Nurmi, E. Soisalon-Soininen and D. Wood in [10] require precondition $\mathsf{tag}(n) \neq 0$ and $\mathsf{tag}(m) = 0$. This precondition has been relaxed by Larsen accepting nonzero values for $\mathsf{tag}(m)$. We can emulate this approach in the same lines as above.

5 Conclusion

Many previously proposed algorithms can be seen as specializations of ours: they amount to imposing a number of additional constraints on the scheduling of the rules. Their correctness is thus a direct consequence of the correctness of our algorithm. Because they restrict themselves to "efficient" scheduling, they can guarantee good performances. Kessels' algorithm deserves a special discussion. Kessels only considers the insertion of leaves at the bottom of the tree. Our algorithm extends Kessels' method on several points. It allows the concurrent insertion of leaves by explicitly describing the percolation of the items to be inserted. It also caters for the concurrent deletion by introducing the notion of "anti-items", which annihilate with their positive fellows. We allow the information to be propagated upwards fully concurrently with the rebalancing process (and also with insertions and deletions!). It turns out that these two goals may conflict one with the

other: propagating information upwards may reveal some imbalance; rebalancing a subtree modifies its height and invalidates the information of its ancestors. We have shown that this conflict nevertheless converges, but this may take a very large number of steps because of the complexity of the interferences. The key result is that even tough balance and knowledge accuracy conflict, a suitable tradeoff between them nevertheless improves: TRADE_OFF = BAL +2. EXCESS. Understanding more deeply the role of this sort of "hocus-pocus" formula would be very interesting: we are currently working in this direction.

Acknowledgments We thank David Cachera for his careful proofreading.

References

1. G.M. Adel'son-Vel'skiĭ and E.M. Landis. An algorithm for the organization of the information. *Soviet Mathematics Doklay*, (3):1259–1263, 1962.
2. R. Bayer and M. Schkolnick. Concurrency of operations on B-trees. *Acta Informatica*, 9(1):1–21, 1977.
3. E.W. Dijkstra, L. Lamport, A.J. Martin, C.S. Scholten, and E.F.M. Steffens. On-the fly garbage collection: an exercice in cooperation. *Comm. ACM*, 21(11):966–975, November 1978.
4. C.S. Ellis. Concurrent search and insertion in AVL trees. *IEEE Trans. Comp.*, C-29(9):811–817, September 1980.
5. J.L.W. Kessels. On-the-fly optimization of data structures. *Comm. ACM*, 26(11):895–901, 1983.
6. D.E. Knuth. *The art of computer programming, Sorting and Searching*, volume 3. Addison-Wesley, 1973.
7. H.T. Kung and P.L. Lehman. Concurrent manipulation of binary search trees. *ACM Trans. Database Systems*, 5(3):354–382, September 1980.
8. K.S. Larsen. AVL trees with relaxed balance. In *Proc. Int. Parallel Processing Symposium*, number 8, pages 888–893. IEEE Comp. Soc., 1994.
9. O. Nurmi, E. Soisalon-Soininen, and D. Wood. Concurrency control in database structures with relaxed balance. In *ACM PODS*, pages 170–176. ACM, 1987.
10. O. Nurmi, E. Soisalon-Soininen, and D. Wood. Concurrent balancing and updating of AVL trees. Technical Report 1992ITKO-B76, Helsinki University of Technology, Department of Computer Science, 1992.