Concurrent AVL revisited: self-balancing distributed search trees

Luc Bougé*, Joaquim Gabarró†‡, Xavier Messeguer†

October 28, 1995. Version written by Luc. File avl.tex

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. This yields a simple and abstract presentation of the insertion and deletion mechanisms. In particular, we show that our approach encapsulates a number of previous attempts described in the literature. They can in fact be seen as ad hoc specializations of the nondeterministic evolution rules. 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?

*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.

Contents

1 Introduction

AVL trees are among the first topics taught to freshmen in the algorithmic course. Since their introduction by Adel'son-Velskiĭ and Landis [1, 8], 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 have tried to find *concurrent* versions of this scheme, where several keys can be concurrently inserted in the tree, the percolation and balancing being run concurrently. Unfortunately, these concurrent versions are most often very complex, sometimes so complex that their correctness is questionable in several cases! Moreover, the underlying principles are often obscured by a large amount of low-level technical details.

The goal of this paper is to show that several of these algorithms derive in fact from a single basic framework by specializating of the nondeterministic evolution strategy. The correctness of this common underlying framework can be proven at a high level of abstraction. It guarantees the correctness of all its possible specializations.

The paper is organized as follows. A synthetic presentation of previous papers on this subject is given in Section 2. We present our abstract framework in Section 3 and prove its correctness. Then, we show in Section 4 how it can be used as a basis for a concurrent *insertion* algorithm. The problem of *deletion* is addressed in Section 5. It turns out that a number of algorithms described in the literature are specializations of our framework, as discussed in section 6.

2 Related works

For the last fifteen years, there has been a number of attempts to design concurrent management schemes for AVL trees: the goal is to allow concurrent insertions and deletions, at least as long as no race condition may occur. Locking groups of nodes in the tree during the critical updates can obviously not be avoided, but the goal is to keep those groups as small as possible, and to lock them for a time as short as possible. The early works of R. Bayer and M. Schkolnick [2] and C.S. Ellis [4] develop several solutions based on complete path optimizations. Concurrent accessing is obtained with a sophisticated locking technique with roll-backs in update. 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.

Later on, to avoid the preceding problems, most of the solutions are described by a set of evolution rules. 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 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 [9]. 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 using the dichromatic trees introduced by L.J. Guibas and R. Sedgewick [5]. Recall that, in the dichromatic approach, tree nodes are painted red or black. Red nodes do not count to compute height: for each internal node, all paths from it to external nodes contain the same number of black nodes. Later on, inspired by on-the-fly garbage collection algorithms [3], J.L.W. Kessels [7] found the first safe and live[4] concurrent insertion algorithm for AVL trees based on atomic rules only. In his approach, the height of every node n is called *real height* (p and q are the sons of n):

$$\mathsf{real\text{-}height}(n) = 1 + \max\{\mathsf{real\text{-}height}(p), \mathsf{real\text{-}height}(q)\}$$

It is adjusted with the contents of a private *carry* register such that $\mathsf{carry}(n) \in \{0, 1\}$. Based on it, J.L.W. Kessels defines a *dynamic height* as

$$\mathsf{dheight}(n) = 1 - \mathsf{carry}(n) + \max\{\mathsf{dheight}(p), \mathsf{dheight}(q)\}.$$

When $\mathsf{carry}(n) = 1$ the node n does not contribute to the dynamic height. This implements in fact the remark of Kung and Lehman about red and black trees, as we can identify $\mathsf{carry}(n) = 1$ with n is red and $\mathsf{carry}(n) = 0$ with n is black. In Kessels' work, a *balance factor* of a node, written $\mathsf{balance}(n)$, is defined in terms of the dynamic height of the subtrees. The eight local transformations are designed to maintain $\mathsf{balance}(n) \in \{-1, 0, +1\}$. To do it, it is necessary to couple the carry *propagation* with a *rotation*. Intuitively, a carry *propagation* improves the consistency of the data

[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).

structure: when all the nodes verify $\mathsf{carry}(n) = 0$, then the $\mathsf{dheight}(n)$ coincides with the real height. Otherwise, a *rotation* attempts to balance subtrees. In Kessels' approach, 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 [11, 12] 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). They introduced a *tag* such that $\mathsf{tag}(n) \in \{-1, 0, 1, 2, \ldots\}$, a relaxed height defined as

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

and a relaxed balance factor $\mathsf{rbalance}(n)$. As before the upward propagation of a unit of tag is followed by a rotation in order to maintain $\mathsf{rbalance}(n) \in \{-1, 0, +1\}$. Finally, K.S. Larsen [10] modified the rules given in [11] 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 it is harder to maintain any notion of local balance and we will get $\mathsf{balance}(n) \in \{\ldots, -2, -1, 0, 1, 2, \ldots\}$. Moreover, the local information can be arbitrarily unfaithful, and one may decide to rotate the wrong subtree. Liveness is questionable.

3 Self-balancing distributed search trees

3.1 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:

$\mathsf{lheight}(n)$: The apparent local height of node n, at the best of its knowledge.

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

$\mathsf{bal}(n)$: The apparent balance of node n, at the best of its knowledge.

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

$\mathsf{delta}(n)$: 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 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, $\mathsf{bal}(n)$ reflects roughly the reality. When $\mathsf{bal}(n) \notin \{-1, 0, 1\}$ we call n *(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 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 sorting 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* $\mathsf{lefth}(n)$ *and* $\mathsf{righth}(n)$ *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 knowledge of the internal nodes of T. In this sense, the behavior is *self-balancing*, very much in the same sense as for self-stabilizing distributed algorithms.

3.2 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 $\mathsf{delta}(m) \neq 0$), 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,

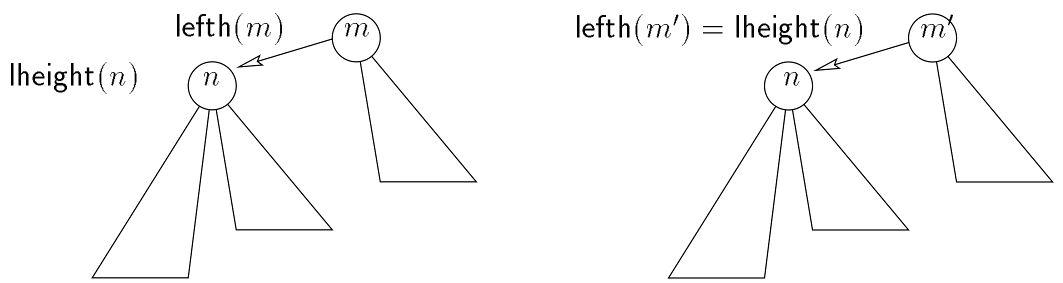

Figure 1: Propagation rule, left case

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* $\mathsf{delta}(n) \neq 0$ *(Figure 1).*

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

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

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

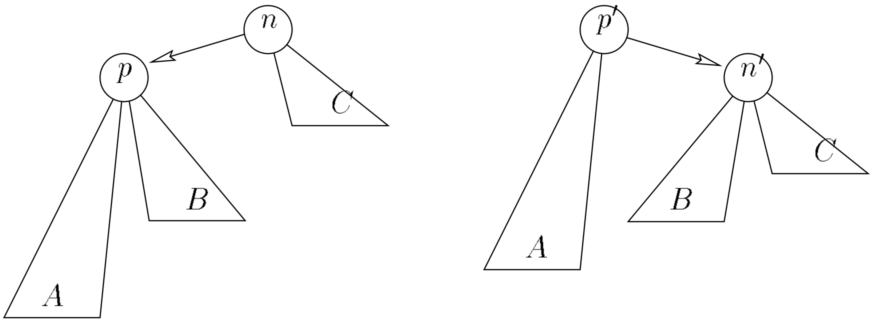

Figure 2: Single rotation rule, left-left case

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 (see Figures 2, 3). As we will see later, 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:*

$$\begin{array}{ll} \mathsf{key}(n') := \mathsf{key}(n) & \mathsf{key}(p') := \mathsf{key}(p) \\ \mathsf{lefth}(n') := \mathsf{righth}(p) & \mathsf{lefth}(p') := \mathsf{lefth}(p) \\ \mathsf{righth}(n') := \mathsf{righth}(n) & \mathsf{righth}(p') := \mathsf{lheight}(n') \end{array}$$

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:*

$$\begin{array}{ll} \mathsf{key}(n') := \mathsf{key}(n) & \mathsf{key}(p') := \mathsf{key}(p) \\ \mathsf{lefth}(n') := \mathsf{righth}(p) & \mathsf{lefth}(p') := \mathsf{lefth}(p) \\ \mathsf{righth}(n') := \mathsf{righth}(n) & \mathsf{righth}(p') := \mathsf{lheight}(n') \end{array}$$

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

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.

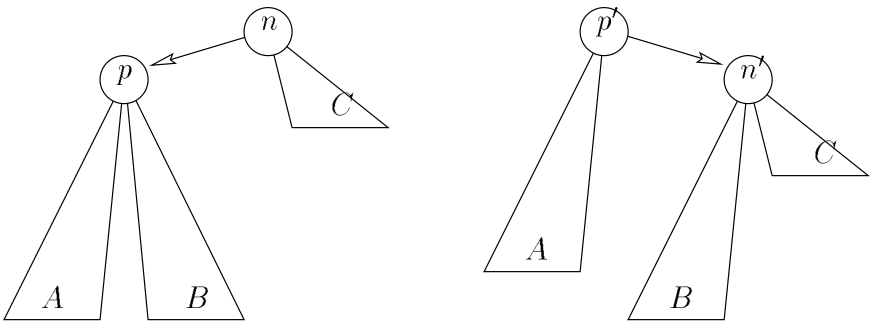

Figure 3: Single rotation rule, left-equal case

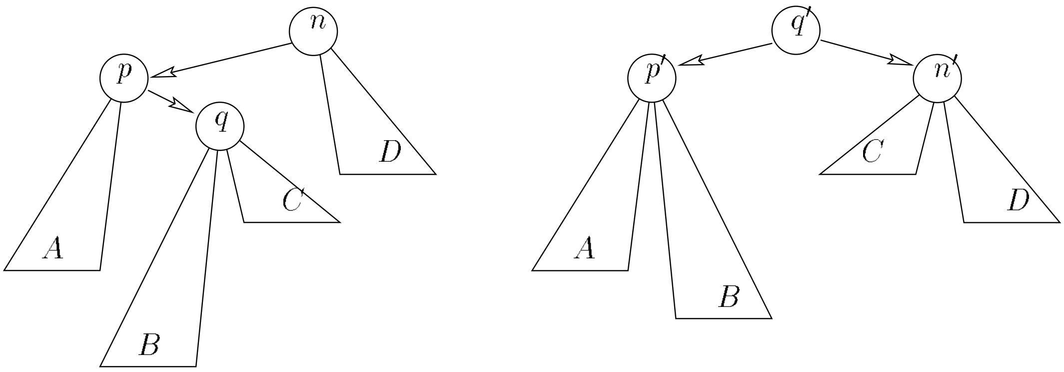

Figure 4: Double rotation rule, left-right case

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,* $\mathsf{bal}(p) > 0$ *(Figure 4).*

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.

3.3 Safety

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: *nothing bad may happen!* Moreover, if the algorithm stops, then we get a good result.

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.*

3.4 Liveness

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.

Propagation Rule: On one side, this rule increases the quality of the father's knowledge, that is the *global stability*. This is good. But on the other side, it may reveal to it that it is in fact more unbalanced than previously believed. It may therefore decrease the *global balance* and this is bad.

Rotation Rules: Intuitively, this rule increases the *global balance* of the tree. This is good. But, applying a rotation at a node usually changes the height of this node. As a side effect, it invalidates the knowledge of its father. It may therefore worsen the *global stability*. This is bad.

As we have to cope with two contradictory effects, it is by no means obvious that the whole process eventually stops. Some oscillating behavior could result, where stability and balance alternatively improve. An in-depth study of their interaction is therefore necessary.

The proof of Liveness Property 2 proceeds through a number of lemmas to catch the precise action of each rule on stability and balance. Unless specified, all lemmas are proved by easy but tedious case enumerations. The key point is that a loss in one quantity is always matched by a larger gain in the other! 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. Yet, we can still easily catch the global effect by an exhaustive case enumeration: the height may not increase.

Lemma 3 (Evolution of $\mathsf{lheight}$)

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, $\mathsf{lheight}$ 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*. In this case $\mathsf{lefth}(n)$ has become smaller but this variation has not been yet communicated to m. This information is being delayed at n and m thinks that n is higher that really it is. Intuitively,

$$\begin{aligned} \mathsf{delta}(n) > 0 \quad &\equiv \text{“The father of } n \text{ believes that } n \text{ is } \mathsf{delta}(n) \text{ units } \textit{higher} \text{ than } n \text{ believes”} \\ &\equiv \text{“The father of } n \textit{ overestimates} \text{ in } \mathsf{delta}(n) \text{ units the local height of } n\text{”} \end{aligned}$$

The variation of the local apparent height through rotations has an interesting consequence: positive unstability is preserved by the rules. Say a tree is *consistent* if $\mathsf{delta}(n) \geq 0$ for all nodes n.

Lemma 4 *Let T be a search tree. Let T' be obtained by the application of some rule. If T is consistent, then T' is, too.*

Proof *This is clear if the rule is a rotation at node n: $\mathsf{delta}(n)$ may not decrease, and all other values are left unchanged . Assume it is a propagation at node n with father m, and that node m has a father o (see Figure 5). Then $\mathsf{delta}(n') = 0$. As $\mathsf{lheight}(n) < \mathsf{lefth}(m)$ by hypothesis, updating $\mathsf{lefth}(m)$ let $\mathsf{lheight}(m)$ decrease. As $\mathsf{lheight}(m) \leq \mathsf{lefth}(o)$, this lets $\mathsf{delta}(m)$ increase: $\mathsf{delta}(m') \geq \mathsf{delta}(m)$. Thus, all values remain positive.*

When $\mathsf{delta}(n) < 0$ we say that n *has negative unstability*. In this case the n apparent local height has grown up from the last message sent its the father m. Therefore m thinks that n is lower that really is it.

$$\begin{aligned} \mathsf{delta}(n) < 0 \quad &\equiv \text{“The father of } n \text{ believes that } n \text{ is } |\mathsf{delta}(n)| \text{ units } \textit{smaller} \text{ than } n \text{ believes”} \\ &\equiv \text{“The father of } n \textit{ underestimates} \text{ in } |\mathsf{delta}(n)| \text{ units the local height of } n\text{”} \end{aligned}$$

In the following two sections we deal separately with

Consistent trees: All the nodes satisfy $\mathsf{delta}(n) \geq 0$. In words, all fathers overestimate the height of their respective sons. Consistency is preserved through the rules.

General trees: At least one node satisfies $\mathsf{delta}(n) < 0$. At least one father underestimates the height of one of its sons.

3.4.1 Defining the variants

As we pointed out above, the lack of stability splits into two different cases.

Consistent trees All nodes satisfy $\mathsf{delta}(n) \geq 0$. Thus, all nodes overestimate the (apparent) height of their respective sons. Let us define a variant functions to deal with this case.

Balance: We define the *global (apparent) balance B* as

$$\mathsf{BAL} = \sum_n |\,\mathsf{bal}(n)|$$

If $\mathsf{BAL} = 0$, then the tree is balanced. Now, observe that on a left-equal single rotation, the global balance strictly betters *if we disregard balanced nodes*. We therefore define

$$\mathsf{RBAL} = \sum_{|\,\mathsf{bal}(n)|>1} |\,\mathsf{bal}(n)|$$

Positive unstability: The overestimation of the apparent height of the whole tree from the father's point of view is given by:

$$\mathsf{EXCESS} = \sum_{\mathsf{delta}(n)>0} \mathsf{delta}(n)$$

Applying the rules sometimes result in nice interactions between BAL and EXCESS. Let us sketch a case appearing in the propagation. Let n be the left son of m and assume that m overestimates the height of n (we mean $\mathsf{delta}(n) > 0$) and that the right subtree of m dominates the balance. As we will see in the following section, when propagation from n to m takes place, we get $\mathsf{EXCESS}' = \mathsf{EXCESS} - \mathsf{delta}(n)$ but $\mathsf{BAL}' = \mathsf{BAL} + \mathsf{delta}(n)$. The increase in stability (the variant EXCESS decreases by $\mathsf{delta}(n)$ units) has been compensated by a similar increase of the unbalance (BAL increases by $\mathsf{delta}(n)$ units). The transformation of inconsistency into unbalance is acceptable if it occurs in a more or less similar amount. We therefore define a new variant combining balance and unstability with different weights:

$$\mathsf{TRADE\_OFF} = \mathsf{BAL} + 2.\,\mathsf{EXCESS}$$

In the preceding case variant $\mathsf{TRADE\_OFF}$ strictly decreases because $\mathsf{TRADE\_OFF}' = \mathsf{TRADE\_OFF} - \mathsf{delta}(n)$. Let us define:

$$\boxed{\text{Variant for consistent trees: } (\mathsf{TRADE\_OFF}, \mathsf{RBAL})}$$

In the following section we will prove:

Property 1 (Consistent Liveness) *For consistent trees, the variant* $(\mathsf{TRADE\_OFF}, \mathsf{RBAL})$ *strictly decreases for the lexicographic order on any rule application and it is is greater than* $(0, 0)$*. Therefore, if we start with a consistent tree, no infinite sequence of rule applications is possible.*

General trees We now relax the hypothesis of consistency. Any father can have an over or under estimation of the apparent height of some of its sons. In addition to the phenomenon appearing in the consistent case (we will need all the preceding variants), we need to consider negative unstability.

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. The rotations help decreasing negative unstability, as they cannot increase the apparent height of subtrees. Upwards flowing can be captured by weighting the lack of stability with a measure of its depth within the tree. Kessels [7] 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.

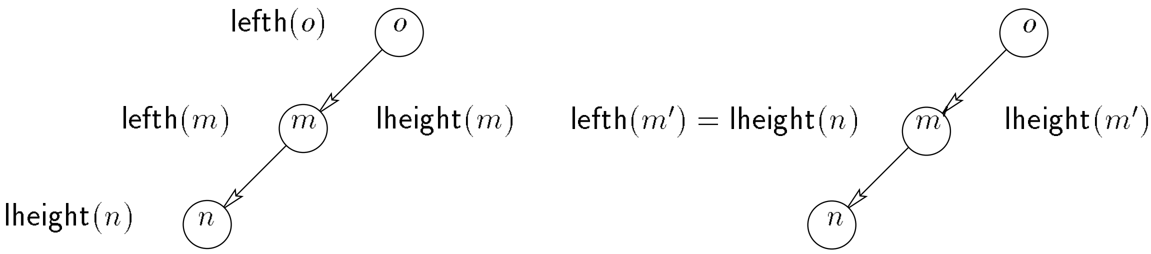

Figure 5: Propagation Rule, left case, 3-node configuration

Let us define:

$$\boxed{\text{Variant for general trees: } (\mathsf{LOSS}, \mathsf{TRADE\_OFF}, \mathsf{RBAL})}$$

In the following section we will prove:

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.*

3.4.2 Proof of Liveness

The following result is crucial for analyzing the various cases.

Lemma 5 *None of the quantities* outside*,* delta*,* bal *is modified outside of the spatial scope of the rules.*

Consistent trees 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 $\mathsf{TRADE\_OFF}$ 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 3 (Variant function) *For any rule application, one of these 2 cases holds:*

1. $\mathsf{TRADE\_OFF}$ *strictly decreases;*
2. $\mathsf{TRADE\_OFF}$ *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 Consider a node n and its father m, let us make a case analysis based on its sign of $\mathsf{delta}(n)$. We assume that there are nodes with negative and positive unstability but the analysis will be done only for nodes of the first case. This is feasible because for nodes with positive unstability we repeat the analysis of the consistent trees: recall that we have only taken into account the local properties of the nodes, and that these local properties are unchanged in this kind of nodes. Now, we undertake a case analysis based on values of the $\mathsf{delta}(n)$ not yet considered in the consistent case.

Case $\mathsf{delta}(n) < 0$.

As claimed above, this case is easier, as the Propagation Rule and the Rotation Rules cooperate. On applying the Propagation Rule, the negative unstability is moved one node upwards: the associated **outside** is thus strictly decreased. Thus $\mathsf{LOSS}' \leq \mathsf{LOSS} - 1$. On applying a Rotation Rule, the height of the subtree decreases, and the associated **delta** is thus decreased. The only case where there is no strict decrease is the Rotation Rule, left-equal case. Thus

- Left-left and left-right rotations: $\mathsf{LOSS}' \leq \mathsf{LOSS} - 1$.
- Left-equal rotation: $\mathsf{LOSS}' = \mathsf{LOSS}$ and $\mathsf{TRADE\_OFF}' = \mathsf{TRADE\_OFF}$.

Case $\mathsf{delta}(n) \geq 0$ and $\mathsf{delta}(m) < 0$.

A careful but tedious case analysis shows the following effect.

- If the left subtree of m is dominant, then the height of m decreases, and $\mathsf{LOSS}' \leq \mathsf{LOSS} - 1$.
- If the right subtree is dominant, then $\mathsf{delta}(m)$ does not change: $\mathsf{LOSS}' = \mathsf{LOSS}$. The unbalance at m gets worse: $\mathsf{BAL}' = \mathsf{BAL} + \mathsf{delta}(n)$. But the global positive unstability betters: $\mathsf{EXCESS}' = \mathsf{EXCESS} - \mathsf{delta}(n)$. Therefore $\mathsf{TRADE\_OFF} = \mathsf{TRADE\_OFF} - \mathsf{delta}(n)$.

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

Property 4 (Variant function) *For any rule application, one of these 3 cases holds:*

1. LOSS *strictly decreases;*
2. LOSS *remains unchanged and* $\mathsf{TRADE\_OFF}$ *strictly decreases;*
3. LOSS *and* $\mathsf{TRADE\_OFF}$ *remain unchanged: this occurs only in the case of a left-equal rotation. In this case* RBAL *strictly decreases.*

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

This concludes the proof of Liveness.

It seems to be very difficult to give any general description of the successive "shapes" of the tree on the 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 6 *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. We have: $\mathsf{LOSS} \leq N^2$, $\mathsf{EXCESS} = 0$, $\mathsf{BAL} = 0$, $\mathsf{RBAL} = 0$.

It is clear that our tree checks the condition of the lemma. Then, any tree derived from it do so. In particular, at any step, $\mathsf{lefth}(n), \mathsf{righth}(n) \leq \mathsf{lheight}(n) \leq \mathsf{real\text{-}height}(n) \leq N$ for any node. Thus $\mathsf{LOSS} \leq N^3$, $\mathsf{BAL} \leq N^2$, $\mathsf{EXCESS} \leq N^2$, $\mathsf{RBAL} \leq \mathsf{BAL} \leq N^2$. The variant function can thus take at most $2.N^7$ values. This gives an obviously extremely rough bound of the actual number of steps needed to reach balance.

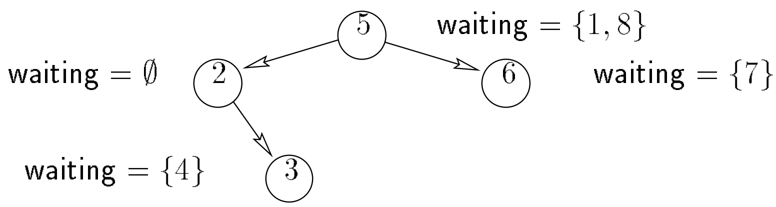

Figure 6: A strongly sorted tree with its waiting bags

4 Concurrent insertion of keys

Using this basic framework, we can easily describe an algorithm to manage the concurrent insertion of keys in distributed sorted tree. 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. $a \geq \mathsf{key}(m)$) (see Figure 6). 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, as shown on Figure 7. An easy patch is the following.

Rule 6 (Rotation with waiting bags)

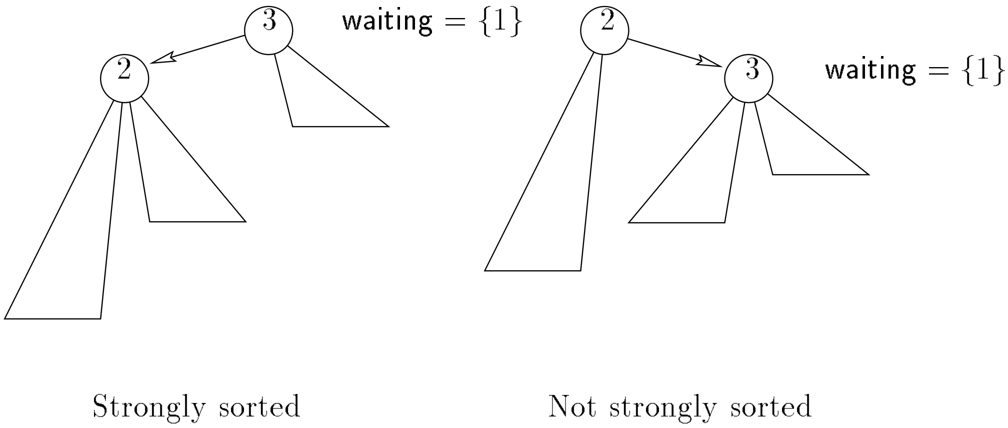

Figure 7: Left-left rotations do not preserve strong sortedness

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.

Lemma 7 *Let T be a distributed, well-founded and strongly sorted search tree. Let T' be the tree obtained by applying any rule. Then, T' is well-founded and strongly sorted, too.*

Property 5 (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.*

Proof *As Percolation Rule does not apply, then all waiting bags are empty. As Propagation Rule does not apply, all nodes are stable. As Rotation Rules do not apply and all the waiting bags are empty, then all nodes are balanced.*

Lemma 8 *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

$$\mathsf{NUMBER} = \sum_n 1$$

Quantity NUMBER is the current number of nodes of the tree. Define also

$$\mathsf{WAIT} = \sum_n \mathsf{inside}(n).|\,\mathsf{waiting}(n)|$$

Quantity 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. We can therefore define

$$\boxed{\text{Variant for general trees: } (\mathsf{NUMBER}, \mathsf{WAIT}, \mathsf{LOSS}, \mathsf{TRADE\_OFF}, \mathsf{RBAL})}$$

Property 6 (Liveness)
The variant function (NUMBER, WAIT, LOSS, TRADE_OFF, RBAL) *strictly decreases on any rule application.*

5 Concurrent insertion and deletion of keys

In this section, 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 $\mathsf{waiting}$ 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 $\mathsf{alive}(n)$, to mark the nodes to be rotated down and removed. We say that a node is *alive* if $\mathsf{alive}(n) = true$, otherwise it is said *dead*.

Let $\{a_1, a_2, \ldots, a_N\}$ the keys to be deleted. We start by locating them into the $\mathsf{waiting}$ 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 $a \in \mathsf{waiting}(n)$ and should be deleted.*

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

Spatial scope: *Nodes n and p.*

The following rule deals with the *dead* nodes: it sends down the marked nodes by means of rotations and eventually removes them. We underline that the rotations considered here are not the rules explained in previous sections, they are the obvious transformations that send down the root and leave invariant the order of keys.

Rule 8 (Garbage Percolation)

Guard: *Node n is dead. If both sons are empty then $\mathsf{waiting}(n)$ does not contains keys to be inserted, and if both sons are non empty then one of them, say p, at least, is alive, stable and $\mathsf{waiting}(p) = \emptyset$.*

Behavior: *Let m be the father of n.*

If both sons of n are empty then node n is removed.

If only one son of n is empty, then node n is removed and the other son p is attached to m and $\mathsf{waiting}(p') := \mathsf{waiting}(p) + \mathsf{waiting}(n)$.

If both sons are non empty and only the left son p is alive, stable and $\mathsf{waiting}(p) = \emptyset$, then a right rotation around n is made and $\mathsf{waiting}(p') := \mathsf{waiting}(m)$ and $\mathsf{waiting}(m') := \emptyset$.

If both sons are alive, stable and have an empty waiting bag a nondeterministic selection can be done between right and left rotations. The registers should then be updated as suggested in the previous paragraph.

Spatial scope: *Node n and its sons.*

Theorem 1 (Safety) *Let T be a distributed, sorted, well-founded search tree. Assume no rule applies to T. Then, T is an AVL whose all waiting bags are empty.*

We now turn to the Liveness property. Note that the WAIT function has been slightly modified by including the keys to be deleted.

Lemma 9 *The Percolation Rule strictly decreases function* WAIT*.*

Let $|T|$ be the number of initial items of the tree T. We define the variant function

$$\mathsf{DEAD} = \sum_{\mathsf{alive}(n)=\text{false}} \mathsf{inside}(n)$$

Lemma 10 *The Garbage Percolation Rule strictly decreases function* $(\mathsf{WAIT}, \mathsf{DEAD})$*.*

Proof *Function* WAIT *decreases in the first two cases (one son, at least, of the dead node is empty) and remains unchanged in the last two cases.*

Function DEAD *strictly decreases if the node is removed. Assume that n is dead and that it is sent down by a rotation around itself and leaves its son p as the new root, then* $\mathsf{WAIT}(n)$ *strictly decreases because, at least, the node p, that is alive by the guard, has been subtracted.*

Before establishing the total correctness, we add to the rotation rules an important condition that must avoid to send up a dead node: the nodes involved should be alive. This condition determines the prior claim of the removing rule than the equilibrating or inserting rules. We define:

$$\boxed{\text{Variant for general trees: } (\mathsf{WAIT}, \mathsf{DEAD}, \mathsf{LOSS}, \mathsf{TRADE\_OFF}, \mathsf{RBAL})}$$

Theorem 2 (Total correctness) *The Propagation, Rotation, Percolation and Garbage Percolation Rules strictly decrease the function* $(\mathsf{WAIT}, \mathsf{DEAD}, \mathsf{LOSS}, \mathsf{TRADE\_OFF}, \mathsf{RBAL})$*. There does not exist any infinite sequence of rule applications. One eventually get an AVL.*

6 Discussion

The approach given here has a very high degree of concurrency and atomicity. We can use this fact to emulate some of the other existing algorithms based on local rules. Recall that an algorithm A can be emulated by an algorithm B if any rule of A can be simulated by a concatenation of a fixed and finite number of rules of B. We keep the discussion informal, but it could be rewritten formally with the notion of homomorphisms between models of parallel computation systems introduced by T. Kasai and R. Miller [6]. We will only take into account the reconstruction phase of the tree (when all the keys have been transformed into new leaves).

6.1 Sequential algorithm

In the original sequential algorithm, every node n holds a register $\mathsf{balance}(n)$ with values in $\{-1, 0, 1\}$. If n has left and right sons p and q, then it holds $\mathsf{balance}(n) = \mathsf{real\text{-}height}(q) - \mathsf{real\text{-}height}(p)$. Essentially, sequential insertion reconstructs the tree bottom up in order to maintain balances. Let us consider what happens with our approach. Assume that a new key has just been added to the tree at the end of the *percolation* process. We can apply the propagation rule bottom up changing the value of $\mathsf{balance}(n)$ along the nodes of the restructuring path (the path going from the new leaf to the last node of the insertion path with a non-zero balance [4]). When we arrive at the critical node (last node having non zero balance), only a rotation can be applied if ever necessary. Therefore, the distributed algorithm mimics a bottom-up version of the sequential one.

Let us consider the emulation of some distributed algorithms based on local rules [7, 11, 12, 10]. As we have seen in section 2 these algorithms are based on some new approximate notion of height. To approach the problem we introduce the *up apparent height*, written $\mathsf{up\text{-}height}(n)$ of a node n, defined as:

$$\mathsf{up\text{-}height}(n) = \mathsf{lheight}(n) + \mathsf{delta}(n)$$

Given a node n having left son p and right son q we have $\mathsf{up\text{-}height}(p) = \mathsf{lefth}(n)$ and $\mathsf{up\text{-}height}(q) = \mathsf{righth}(n)$. The apparent height $\mathsf{up\text{-}height}(n)$ give us the information known by the father of n about the height of its son n. This function behaves as an approximate height because it verifies:

$$\mathsf{up\text{-}height}(n) = 1 + \mathsf{delta}(n) + \max\{\mathsf{up\text{-}height}(p), \mathsf{up\text{-}height}(q)\}$$

Remark that $\mathsf{delta}(n) \in \{\ldots, -2, -1, 0, 1, 2, \ldots\}$. Moreover the balance can be rewritten as:

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

6.2 J.L.W. Kessels' approach

Let us see how to emulate the algorithm [7]. We start giving a brief description of it. Every node n has the registers $\mathsf{balance}(n) \in \{-1, 0, 1\}$, $\mathsf{carry}(n) \in \{0, 1\}$ and there is a dynamic height defined as:

$$\mathsf{dheight}(n) = 1 - \mathsf{carry}(n) + \max\{\mathsf{dheight}(p), \mathsf{dheight}(q)\}$$

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 [7]). All these transformations maintain the balance and the carry constraints. The first one moves the carry up when this fact does not unbalance the tree. The second one couples a carry updating with a single rotation. The last one modifies the carry adequately under double rotations. For instance the Transformation (b) can be rewritten as follows:

Rule 9 (Transformation (b))

Guard: *Node n is a left son of m and* $\mathsf{balance}(n) < 0$, $\mathsf{carry}(n) = 1$, $\mathsf{balance}(m) < 0$, $\mathsf{carry}(m) = 0$.

Behavior: *A right rotation takes place (left-left case) and the new values of balance and carries are* $\mathsf{balance}(n') = \mathsf{balance}(m') = 0$ *and* $\mathsf{carry}(n') = \mathsf{carry}(m') = 0$.

Spatial Scope: *Nodes n and m.*

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

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) can be emulated concatenating a propagation with a single rotation. Let us consider in more detail the five cases appearing in (c). The first three of them can be emulated with a propagation concatenated with a double rotation. The last two cases cannot be obtained in this way (because a node having carry 1 is located in the wrong place), however our approach (concatenating a propagation with a double rotation) gives us alternative and equally correct solutions. As we identify $\mathsf{delta}(n) = -\,\mathsf{carry}(n)$ and $\mathsf{carry}(n) \in \{0, 1\}$ we get $\mathsf{delta}(n) \in \{0, -1\}$. Therefore any emulated transformation needs to maintain $\mathsf{delta}(n) \in \{0, -1\}$. Let us explain what it means. The condition $\mathsf{carry}(n) \in \{0, 1\}$ holds before and after the application of any transformation. It does not make sense to take $\mathsf{carry}(n)$ into account during the transformation (a transformation can be thought as instantaneous and without internal structure). Therefore, in our approach, the constraint $\mathsf{delta}(n) \in \{0, -1\}$ holds before and after the emulation of any rule. As any transformation is emulated by a concatenation of a small number of rules, the emulated transformation has an "internal structure" induced by this concatenation. We relax the constraint $\mathsf{delta}(n) \in \{0, -1\}$ during the emulation. For instance, in the emulation of b, just after the propagation and before a rotation we can have $\mathsf{delta}(n) = -2$. A rotation takes place precisely to correct this situation.

6.3 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 [12] (see also [11]). 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:

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

As before, we consider only the restructuring process. Let n be a node with father m. The transformations assume $\mathsf{tag}(n) \neq 0$ and $\mathsf{tag}(m) = 0$ and have two phases. Phase 1 is designed to decrease the tag value of n. At the end of this phase, we can have $\mathsf{rbalance}(m) \in \{-2, 2\}$. Phase 2 adjusts $\mathsf{rbalance}(m)$ readjusting tags and making a rotation if necessary. The algorithm will eventually reach $\mathsf{tag}(n) = 0$. As they are local and need to maintain the relaxed balance, only small variations on the tag values can be correctly absorbed with rotations and tag adjustments. Therefore, the authors consider only a tag decrease of one unit, $\mathsf{tag}(n') = \mathsf{tag}(n) \pm 1$ (maintaining the constraint $\mathsf{tag}(n') \in \{-1, 0, 1, 2, \ldots\}$). This small decrease of the tag adds a level of complexity in the description of the transformations. Let us consider how to emulate this algorithm. 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 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. In this way node n can transmit only a very small part of its information, keeping an account of the remaining one to be transmitted. 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. As a major level of detail has been added, rules are more complex. For instance, with a little bit of thought, we get the following new rule to propagate one unit of $\mathsf{delta}(n)$ (we give the rule for both, left and right case because of the lack of symmetry in the evolution of balance field):

Rule 10 (One unit propagation)

Guard: *Node n is a left son of m and* $\mathsf{delta}(n) \neq 0$.

Behavior: *When* $\mathsf{delta}(n) > 0$ *we have* $\mathsf{delta}(n') = \mathsf{delta}(n) - 1$. *If n is a left son of m then* $\mathsf{lefth}(m') = \mathsf{lefth}(m) - 1$, $\mathsf{delta}(m') = \mathsf{delta}(m) + (\mathsf{bal}(m) < 0)?1 : 0$ *else n is a right son of m and* $\mathsf{righth}(m') = \mathsf{righth}(m) - 1$, $\mathsf{delta}(m') = \mathsf{delta}(m) + (\mathsf{bal}(m) > 0)?1 : 0$. *All other registers remain constant.*

When $\mathsf{delta}(n) < 0$ *we have* $\mathsf{delta}(n') = \mathsf{delta}(n) + 1$. *If n is a left son of m then* $\mathsf{lefth}(m') = \mathsf{lefth}(m) + 1$, $\mathsf{delta}(m') = \mathsf{delta}(m) - (\mathsf{bal}(m) \leq 0)?1 : 0$ *otherwise n is a right son of m and* $\mathsf{righth}(m') = \mathsf{righth}(m) + 1$, $\mathsf{delta}(m') = \mathsf{delta}(m) - (\mathsf{bal}(m) \geq 0)?1 : 0$. *All other registers remain constant.*

Spatial Scope: *Nodes n and m.*

We can get (with patience!) new versions for rules dealing with rotations and double rotations which take into account the small variations of $\mathsf{delta}(n)$. Using these new rules we can mimics the restructuring transformations given in [12] mainly coupling propagations and rotations.

6.4 K.L. Larsen's approach

Finally let us consider the approach taken by K. L. Larsen [10]. Recall that, given a node n having a father m, the transformations given by O. Nurmi, E. Soisalon-Soininen and D. Wood in [12] require precondition $\mathsf{tag}(n) \neq 0$ and $\mathsf{tag}(m) = 0$. This precondition has been relaxed by Larsen accepting nonzero values for $\mathsf{tag}(m)$. To do it, K.L. Larsen modifies the preceding transformations to avoid the accumulation of negative values in $\mathsf{tag}(m)$. As before he also couples a propagation with the rotation. More precisely, he defines 13 rules and he needs an scheduler to joint sequentially some of them. For instance, transformation 3 given in the Appendix of [10] is a propagation in the case $\mathsf{delta}(n) = \mathsf{tag}(n) = -1$, Transformation 4 is a propagation when $\mathsf{delta}(n) = \mathsf{tag}(n) = 1$ and Transformation 5 is a right rotation. As before we can emulate this approach.

7 Conclusion

Much work has been devoted to designing a distributed version of the original AVL algorithm. The difficulty is to tackle the problem at a sufficiently high level of abstraction, because of the so many technical details involved with the rotations and their possible interactions. A precise operational description of the behavior is almost impossible. The rule-based approach, as proposed by Dijkstra and Scholten for distributed termination algorithms, seems very promising. It has been first proposed in the framework of AVL trees by Kessels. His method is based on red-black dichromatic trees.

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. Also, his rules are quite restricted, on the sense that the effect of adding a new leaf may be propagated upwards in the tree only after the current subtree has reached back its balance: if only one single insertion is considered, this is thus exactly the original sequential behavior. 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.

On a more fundamental level, our algorithm allows more concurrency than Kessels' one. In effect, 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: $\mathsf{TRADE\_OFF} = \mathsf{BAL} + 2.\,\mathsf{EXCESS}$. At this time, we are not able to give any intuitive explanation of this sort of "hocus-pocus" formula. Also, we cannot prove that our variant function are optimal: it may be possible to find more complex variant to catch the global effects more tightly, resulting in better upper bounds. We still have no precise idea of what a "worst behavior" could be. Extensive computer simulations may help 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] L.J. Guibas and R. Sedwick. A dichromatic framework for balanced trees. In *Proc. Ann. Symp. Foundations of Computer Science*, number 19, pages 8–21. IEEE Comp. Soc., 1978.

[6] T. Kasai and R. Miller. Homomorphisms between models of parallel computation. *J. Comp. Sys. Sci.*, 25:285–331, 1982.

[7] J.L.W. Kessels. On-the-fly optimization of data structures. *Comm. ACM*, 26(11):895–901, 1983.

[8] D.E. Knuth. *The art of computer programming, Sorting and Searching*, volume 3. Addison-Wesley, 1973.

[9] H.T. Kung and P.L. Lehman. Concurrent manipulation of binary search trees. *ACM Trans. Database Systems*, 5(3):354–382, September 1980.

[10] K.S. Larsen. AVL trees with relaxed balance. In *Proc. Int. Parallel Processing Symposium*, number 8, pages 888–893. IEEE Comp. Soc., 1994.

[11] O. Nurmi, E. Soisalon-Soininen, and D. Wood. Concurrency control in database structures with relaxed balance. In *ACM PODS*, pages 170–176. ACM, 1987.

[12] 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.