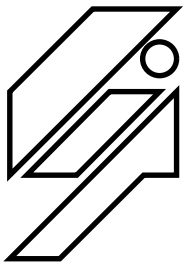


Laboratoire de l'Informatique du Parallélisme

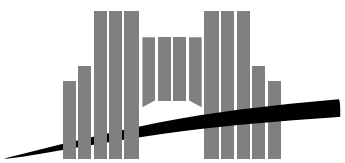
Ecole Normale Supérieure de Lyon
Unité de recherche associée au CNRS n°1398

Concurrent rebalancing of AVL trees: A fine-grained approach

Luc Bougé,
Joaquim Gabarró,
Xavier Messeguer and
Nicolas Schabanel

Mai 1997

Research Report N^{O} 97-13



Ecole Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : (+33) (0)4.72.72.80.00 Télécopieur : (+33) (0)4.72.72.80.80
Adresse électronique : lip@lip.ens–lyon.fr

Concurrent rebalancing of AVL trees: A fine-grained approach

Luc Bougé,
Joaquim Gabarró,
Xavier Messeguer and
Nicolas Schabanel

Mai 1997

Abstract

We address the concurrent rebalancing of almost balanced binary search trees (AVL trees). Such a relabalancing may for instance be necessary after successive insertions and deletions of keys. We show that this problem can be studied through the self-reorganization of distributed systems of nodes controlled by local evolution rules in the line of the approach of Dijkstra and Scholten. This yields a much simpler algorithm that the ones previously known. As a by-product, 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 search trees?

Keywords: Concurrent algorithms, Search trees, AVL trees, Concurrent insertions and deletions, Concurrent generalized rotations, Safety and liveness proofs.

Résumé

Ce rapport présente un algorithme concurrent pour la gestion dynamique d'un arbre binaire de recherche équilibré (Arbre AVL). Une série d'insertions et de suppressions de clés dans un tel arbre peut lui donner une forme arbitraire. Nous montrons dans ce papier que le rééquilibrage de la structure peut être vu comme une auto-réorganisation d'un système distribué formé par les nœuds, dirigée par quelques règles d'évolutions locales. Cette approche mène à un algorithme bien plus simple que les solutions précédentes connues. De plus, ce résultat permet de répondre à la question posée par H.T. Kung et P.L. Lehman : "Peut-on effectuer les rotations dans un ordre quelconque pour rééquilibrer un arbre arbitraire ?". La réponse est : "Oui".

Mots-clés: Algorithmes concurrents, Arbres de recherche, Arbre AVL, Insertions et suppressions concurrentes, Rotations concurrentes généralisées, Preuves de terminaison distribuée.

Concurrent rebalancing of AVL trees: A fine-grained approach

Luc Bougé[1] Joaquim Gabarró[2] Xavier Messeguer[2] Nicolas Schabanel[1]

May 21, 1997

Abstract

We address the concurrent rebalancing of almost balanced binary search trees (AVL trees). Such a relabalancing may for instance be necessary after successive insertions and deletions of keys. We show that this problem can be studied through the self-reorganization of distributed systems of nodes controlled by local evolution rules in the line of the approach of Dijkstra and Scholten. This yields a much simpler algorithm that the ones previously known. As a by-product, 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 search trees?

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

1 Introduction

Search trees are the key in implementing large data structures where keys are searched, inserted and deleted. As the performances are directly related to the height of the tree, sophisticated schemes have been devised to to keep it as small as possible. This is usually done by reorganizing the whole tree after each access, moving subtrees around so as to minimize the height whilst keeping the keys in a sorted order.

The scheme introduced by Adel'son-Velskĭĭ and Landis [AL62, Knu73], nowadays known as the *AVL scheme*, consists in keeping all internal nodes balanced, that is, the height of their subtrees differing at most by one. Search trees with this property are called *AVL trees*. Even though an AVL tree is not of minimal height among all search trees with the same set of keys, it turns out that this organization yields very good asymptotic worst-case and average performances. Moreover, rebalancing an AVL tree after an insertion boils down to a small number of pointer manipulations known as *AVL rotations* (deletion is slightly more difficult). Starting from the inserted leaf upwards to the root, the nodes are recursively rotated by moving up high subtrees.

Adapting this scheme to support highly concurrent updates (insertion and deletion), such as in large databases accessed asynchronously by users, is a valuable challenge. In effect, rebalancing a node may alter and worsen the balance of its ancestors. Concurrent rebalancing may thus lead to "very unbalanced" search trees. In an early attempt, Ellis [Ell80] proposed a complex machinery based on a locking technique combined with roll-back updates (see also [BS77] for a similar approach for concurrent B-trees). The explicit low-level manipulation of locks leads to a very operational

[1]LIP, ENS Lyon, 46 allée d'Italie, F-69364 Lyon Cedex 07, France.
[2]LSI, Universitat Politécnica de Catalunya, FIB/FME, C/ Pau Gargallo, 5, E-Barcelona 08028, Spain.

(and error-prone!) description. Moreover, the root of the tree is a dramatic bottleneck in this approach, which can be qualified as "coarse-grained".

Later attempts have thus explored higher-level approaches. Kessels [Kes83] proposes to see the reorganization of the data structure as a side effect of the access. The reorganization is split into atomic steps involving a small number of nodes, so as to allow concurrency. Additional registers, local on each node, are used to store the necessary information on intermediate states. At each update, a reorganizing process is launched, which proceeds asynchronously. This approach is extended by Nurmi, Soisalon-Soininen and Wood [NSS96, NSSW87, NSSW92]. Finally Larsen [Lar94] shows that the reorganization process converges in $O(k.\log(n+2.k))$ steps in a tree with n nodes updated with k insertions. Each step consists in propagating a piece of information from a son to its father, and applying the appropriate sequence of rotations to the father so as to restore its balance. These transformations are described by 9 different rules, depending on the values of the local registers. As an atomic step may include complex operation, this approach can be qualified as "medium-grained".

The contribution of this paper is to go one step further towards a "fine-grained" solution to concurrent updates of AVL search trees. As for Kessels, our source of inspiration is the seminal work by Dijkstra, Lamport at al. [DLM+78] on concurrent "on the fly" garbage-collection. Their key idea is to completely uncouple the reorganization of the data structure (collecting the garbage cells and linking them into the free list) from its updates (creating garbage cells by pointer manipulations).

Taking this viewpoint in our problem lets us consider the insertions or deletions of keys in the data structure as external *perturbations* done by mutator processes on which we have no control. Reorganizing the data structure is the job of asynchronous daemons which have no knowledge about the ongoing mutations. They just compete with the mutator processes to access the data structure. The only consistency restriction on their behavior is that it should respect the invariant on which the insertion and deletion protocols are based: the keys appear in sorted order.

We even go one step further by splitting the macro-steps of Larsen into finer atomic steps, and we define two kinds of daemon actions:

Propagation: Flowing the information about current updates upwards in the tree, from the leaves to the root.

Rotation: Rebalancing the nodes according to their best knowledge about the shape of the tree.

It turns out that only one rule suffices to specify propagations, and three for the rotations, which makes our approach significantly simpler than the previous ones. Moreover, as the daemons have no knowledge of the keys, the rules apply to any binary tree, without regards to the way it has been obtained by successive insertions and deletions. This is the reason why we have coined the term *AVL rebalancing*. As a byproduct, it answers in a very general setting an old question raised by H.T. Kung and P.L. Lehman [KL80]: where should rotations take place for to rebalance arbitrary trees? The answer is: anywhere.

The price to pay for this "fine-grained approach" is that $\Theta(n^2)$ steps are needed to rebalance an arbitrary binary tree in the worst case, instead of Larsen's $O(n.\log(n))$ for an empty tree filled by n successive insertions. Note however that a single atomic step of Larsen corresponds to several steps here which makes the comparison slightly more balanced. Also, we provide the user with a better degree of concurrency. Finally, there is good experimental evidence that the convergence is obtained in $O(n)$ steps in the average.

The rest of the paper is organized as follows. Section 2.4 describes the model and the two daemons. Correctness is easy: once no daemon can work any more, the tree is an balanced. On

the other hand, convergence turns out to be much more difficult, as shown in Section 3. Finally, Section 4 reports on experimental measures of the average behavior.

2 A Concurrent AVL Rebalancing Scheme

2.1 Goal

Our goal is to design a general rebalancing strategy based on sets of local atomic actions applied concurrently. To ensure good concurrency, each action should lock as few nodes as possible for a time as short as possible. Thus, no reliable knowledge on the current global shape of the tree can be assumed. Each node stores in local registers its best local knowledge on the tree. The only reliable information is that the nodes with empty sons are aware of it. Structural knowledge has thus to be flowed explicitly through the tree from the leaves to the root.

Since insertions and deletions are unpredictable and actions have only a local scope, no global termination may be expected from this basic scheme. Instead, one has to consider a *distributed* form of *termination*: if no perturbation occurs anymore (insertion or deletion of keys), then eventually no action applies. The algorithm blocks waiting to detect new perturbations. Standard techniques can be used to superimpose a distributed termination detection algorithm to this scheme, so as to enforce global termination [Fra80] if wanted.

Observe that no extra assumption is made on the original shape of the tree besides the one mentioned above. In particular, we do not assume that the tree was actually yielded by a sequence of insertions and deletions from a balanced tree. Our scheme just takes any search tree at any time with arbitrary (even worst) information at each node and eventually rebalances it. In contrast with the previous solutions, it is thus *highly fault tolerant*. Also, it is compatible with any method for inserting and deleting keys.

The life of a daemon runs as follow: it wakes up at some point, selects a set of nodes satisfying one of its guards and locks it while it applies the appropriate action. The selection step may be roughly implemented by a random draw among all the nodes or more efficiently by a problem queue as suggests Larsen in [Lar94].

Finally observe that our approach supports any number of asynchronous daemons. A possible extreme approach is to allocate one daemon to each node of the tree: this leads to a sort of (rather unrealistic) self-balancing intelligent tree. The other extreme is to consider that the tree is a large database managed by a multiprocessor server: the system uses the cycles left idle by the clients to reorganize the structure optimally.

2.2 General Description

Let u be a node of the search tree. We denote respectively by $u{\rightarrow}\mathsf{p}, u{\rightarrow}\mathsf{ls}, u{\rightarrow}\mathsf{rs}$ the parent, the left son and the right son of u in the tree. The empty tree is denoted 'nil' and the root of the tree 'root'. The *real height* $\mathsf{realh}(u)$ is defined as usual:

$$\begin{cases} \mathsf{realh}(\mathsf{nil}) = 0 \\ \mathsf{realh}(u \neq \mathsf{nil}) = 1 + \max\left(\mathsf{realh}(u{\rightarrow}\mathsf{ls}), \mathsf{realh}(u{\rightarrow}\mathsf{rs})\right) \end{cases}$$

As concurrent modifications in the tree prevent from maintaining realh on each node, each node $u \neq \mathsf{nil}$ encodes its *local* knowledge of the state of the structure in two *private* registers in addition to the key register:

- $\mathsf{lefth}(u)$ and $\mathsf{righth}(u)$ are respectively the *apparent heights* of the left and right sons of u, at the best of the knowledge of u.

Definition 1 We call *height-relaxed search tree (HRS-tree)* a search tree whose nodes are equipped with the two private registers lefth and righth satisfying the following consistency condition: $\mathsf{lefth}(u) = 0$ (resp. $\mathsf{righth}(u) = 0$) for any node u with an empty left (resp. right) son.

The following auxiliary functions on the nodes of HRS-trees will be useful.

- $\mathsf{localh}(u)$ is the *apparent local height* of u, as computed from the two previous registers :
$$\mathsf{localh}(u) = 1 + \max(\mathsf{lefth}(u), \mathsf{righth}(u))$$
- $\mathsf{car}(u)$, the *carry* of u, is the gap of knowledge between u and its parent:
$$\mathsf{car}(u) = \begin{cases} \mathsf{lefth}(u{\rightarrow}p) - \mathsf{localh}(u) & \text{if } u \text{ is the left son of its parent} \\ \mathsf{righth}(u{\rightarrow}p) - \mathsf{localh}(u) & \text{otherwise} \end{cases}$$
The car function measures the inconsistency of local information on the structure of the tree. A node u is said *reliable* if $\mathsf{car}(u) = 0$. By convention $\mathsf{car}(\mathsf{root}) = 0$.
- $\mathsf{bal}(u)$ of u is the *apparent balance* of u, defined as follow:
$$\mathsf{bal}(u) = \mathsf{lefth}(u) - \mathsf{righth}(u)$$
A node u is said *apparently balanced* if $|\mathsf{bal}(u)| \leqslant 1$.

The following fact holds:

Lemma 1 *If each node of an HRS-tree T is reliable and apparently balanced, then T is an AVL.*

2.3 Description of the Daemons

Propagation Rule. It propagates information upwards from a son to its parent. As a convention the final state of a node u after application of a rule is denoted u'. We only presents the variations of lefth and righth from which the registers localh, car and bal are computed.

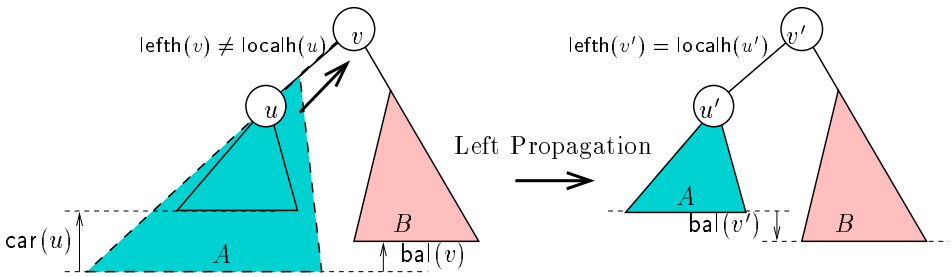


Figure 1: Propagation rules: Rule (LP) left propagation if $\mathsf{car}(u) \neq 0$

Rule (LP) – Left Propagation
Guard : Node u is the left son of node v and u isn't reliable: $\mathsf{car}(u) \neq 0$
Action : the apparent left height of v is updated:
$$\mathsf{lefth}(v') = \mathsf{localh}(u)$$

Note that : $\mathsf{bal}(v') = \mathsf{bal}(v) - \mathsf{car}(u)$.
Spatial scope : u and its parent $v = u{\rightarrow}p$.

The right propagation rule (RP) where u is the right son of v, can be deduced symmetrically from (LP). It is easy to see that applying these rules repeatedly will eventually set the apparent local height of each node to its real height.

Rotation Rules. These rules are inspired from the original AVL rules [AL62] but extended to the case where the balances of the nodes may exceed 2. These relaxed preconditions allow to rebalance any tree with any initial local knowledge. The rotation rules tend to reduce the apparent balance, but of course, can worsen not only the consistency of the local heights but also the real balance if the apparent balance was wrong.

Rule (RR*) – Right Rotation, Unbalanced case
Guard : Node u is the left son of node v, u is reliable, $\mathsf{bal}(u) > 0$ and $\mathsf{bal}(v) \geqslant 2$
Action : u and v execute a right rotation [FIG. 2(a)] with the obvious updating :

$$\begin{array}{ll} \mathsf{lefth}(u') = \mathsf{lefth}(u) & \mathsf{righth}(u') = \mathsf{localh}(v') \\ \mathsf{lefth}(v') = \mathsf{righth}(u) & \mathsf{righth}(v') = \mathsf{righth}(v) \end{array}$$

Note that : $\mathsf{localh}(u') = \mathsf{localh}(v) - 1$, so $\mathsf{car}(u') = \mathsf{car}(v) + 1$.
Spatial scope : u and its parent $v = u{\rightarrow}\mathsf{p}$.

The rule (LR^*), where u is the left son of v, and u and v execute a left rotation when $\mathsf{bal}(u) < 0$ and $\mathsf{bal}(v) \leqslant -2$, is obtained symmetrically from (RR^*).

Rule ($\mathbf{RR_=}$) – Right Rotation, Balanced case
Guard : Node u is the left son of node v, u is reliable, $\mathsf{bal}(u) = 0$ et $\mathsf{bal}(v) \geqslant 2$
Action : u and v execute a right rotation [FIG. 2(b)] with the obvious updating :

$$\begin{array}{ll} \mathsf{lefth}(u') = \mathsf{lefth}(u) & \mathsf{righth}(u') = \mathsf{localh}(v') \\ \mathsf{lefth}(v') = \mathsf{righth}(u) & \mathsf{righth}(v') = \mathsf{righth}(v) \end{array}$$

Note that : $\mathsf{localh}(u') = \mathsf{localh}(v)$, so $\mathsf{car}(u') = \mathsf{car}(v)$.
Spatial scope : u and its parent $v = u{\rightarrow}\mathsf{p}$.

The rule $(LR_=)$, where u is the left son of v and, u and v execute a left rotation when $\mathsf{bal}(u) = 0$ and $\mathsf{bal}(v) \leqslant -2$, is obtained as before symmetrically from $(RR_=)$.

Rule (LRR) – Left-Right double Rotation
Guard : Node w is the right son of the left son u of node v, w and u are reliable, $\mathsf{bal}(u) < 0$ et $\mathsf{bal}(v) \geqslant 2$
Action : u, v and w execute a left-right double rotation [FIG. 2(c)] with the obvious updating :

$$\begin{array}{ll} \mathsf{lefth}(u') = \mathsf{lefth}(u) & \mathsf{righth}(u') = \mathsf{lefth}(w) \\ \mathsf{lefth}(v') = \mathsf{righth}(w) & \mathsf{righth}(v') = \mathsf{righth}(v) \\ \mathsf{lefth}(w') = \mathsf{localh}(u') & \mathsf{righth}(w') = \mathsf{localh}(v') \end{array}$$

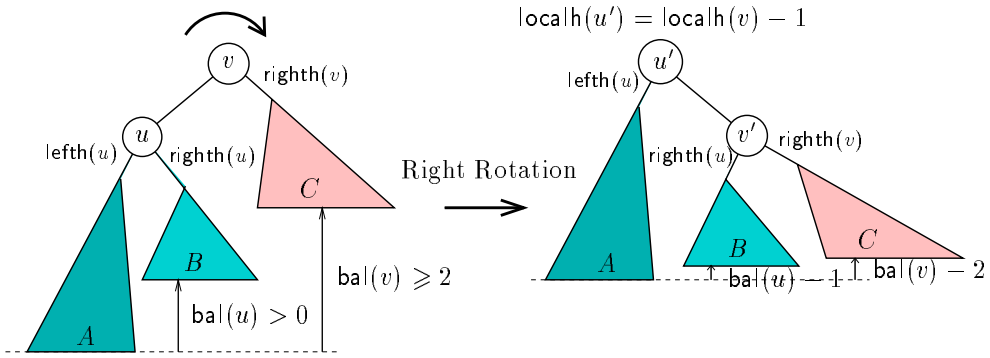
Note that : $\mathsf{localh}(w') = \mathsf{localh}(v) - 1$, so $\mathsf{car}(w') = \mathsf{car}(v) + 1$.
Spatial scope : u, its parent $v = u{\rightarrow}\mathsf{p}$ and its right son $w = u{\rightarrow}\mathsf{rs}$.

The symmetrical rule (RLR) where w is the left son of the right son u of v and u, v and w execute a right-left double rotation, applies when u and w are reliable, $\mathsf{bal}(u) > 0$ and $\mathsf{bal}(v) \leqslant -2$.

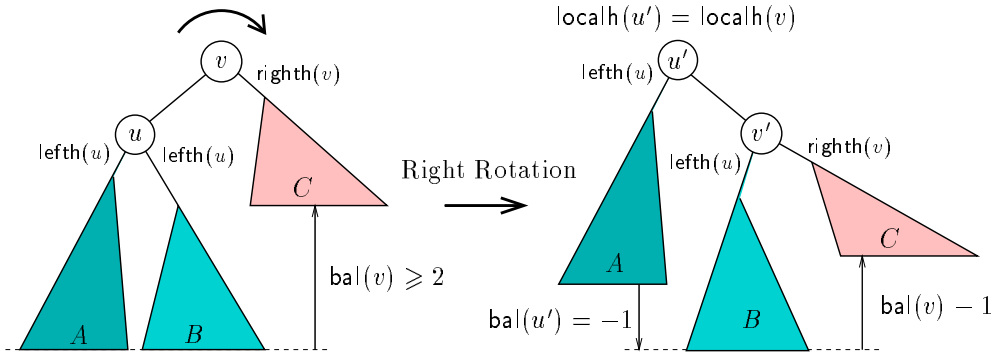
2.4 Invariant Properties

The following lemma ensures the *safety* of the algorithm: "nothing bad can happen: if the algorithm blocks, then we hold the right result".

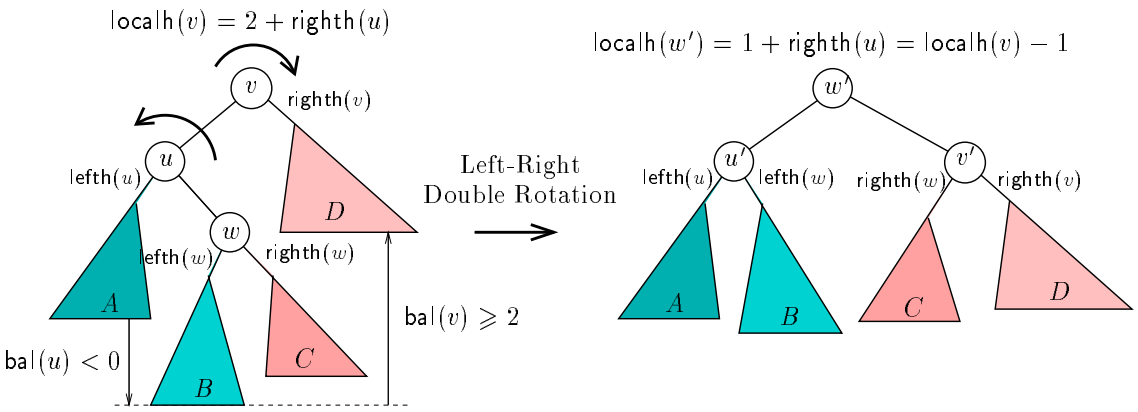
Lemma 2 (Safety property) *Let T be an HRS-tree. If T' is obtained by applying on T any one of the rules described above, then T' is an HRS-tree holding the same keys than T. Moreover if no rule applies on T, T is an AVL.*



(a) Rule (RR*) right rotation if car$(u) = 0$, bal$(u) > 0$ and bal$(v) \geqslant 2$.



(b) Rule (RR$_=$) right rotation if car$(u) = 0$, bal$(u) = 0$ and bal$(v) \geqslant 2$.



(c) Rule (LRR) left-right double-rotation if car$(u) = $ car$(w) = 0$, bal$(u) < 0$ and bal$(v) \geqslant 2$.

Figure 2: The rotation rules

Proof. Thanks to Lemma 1, the proof just consists in noticing that rotations preserve the depth-first traversal order. □

A closer look at the rules reveals the following fact which is actually the key to the proof of convergence below.

Lemma 3 (Stable state) *Let T be an HRS-tree, so that $\forall u \in T$ $\mathsf{car}(u) \geqslant 0$. If T' is obtained by applying on T any one of the rules described above, then $\forall u' \in T'$ $\mathsf{car}(u') \geqslant 0$.*

Proof. This claim is easy to prove as soon as the predicate $\forall u$ $\mathsf{car}(u) \geqslant 0$ is rewritten as "each parent overestimates the height of its sons". It is clear that if v overestimates the height of its son u and if u updates the height of v, then the height of v at worst decreases and v is thus even more overestimated by its parent. The same remark also applies to the rotation rules because a rotation rule at worst decreases the height of the root of the rotation, and leaves reliable the other nodes. □

3 Convergence

As long as perturbations occur in the tree (insertion or deletion of keys), the daemons just compete with the mutators. The resulting behavior essentially depends on their relative speeds. For the convergence analysis, we hence assume that no insertion or deletion occurs anymore and prove then that at most $\Theta(n^2)$, where n is the number of nodes of the tree, rules may be applied. By Lemma 2, the resulting tree is an AVL: the algorithm rebalances thus any arbitrary tree using at most $\Theta(n^2)$ rules.

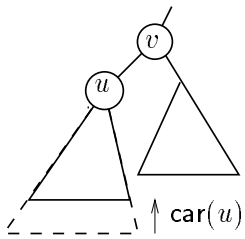
To prove the convergence, observe that if some rules are applied in parallel, their spatial scope are necessary disjoint and then the effect is exactly the same as if they would have been executed sequentially in any order: we will then assume that the rules are applied sequentially. We need to appreciate the global effect of each local rule. This is usually done by extracting a variant which strictly decreases on each rule application. Basically the propagation rule improves the global consistency of the local heights by erasing the carries whereas the rotation rules weaken the apparent imbalance of each node. Unfortunately these two families of rules fight one against the other: three cases in point are shown Figure 3.

On [Fig. 3(a)], the application of the rule (LP) erases the carry of u and not only modifies the local height of v (and then its carry) but also decreases the imbalance of v (by $|\mathsf{car}(u)|$). Conversely on [Fig. 3(b)], the application of the rule (LP) increases the imbalance of v (by $|\mathsf{car}(u)|$). On [Fig. 3(c)] the effect of the propagation on the balance of v depends on the relative apparent heights of the two subtrees.

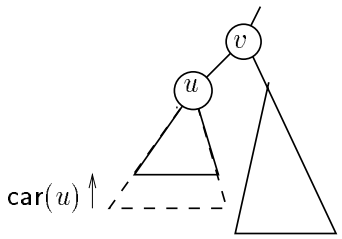
Similarly, the application of a rotation rule [Fig. 2(a)] reduces the imbalance of the root of the rotation but modifies its local height and may increase its carry.

There is thus a subtle interaction between car and bal: they cannot be considered independently to prove the convergence. A tedious exhaustive case analysis is necessary which is summed up in tables [Tab. 1 and 2]. A sketch of the analysis is presented in the sections bellow.

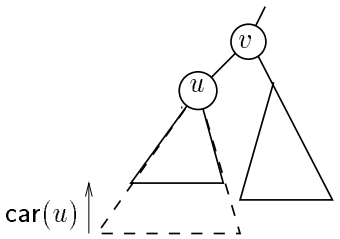
In the following sections, we only consider the rules (LP), (RR^*), $(RR_=)$, and (LRR), the same results holds trivially for the symmetrical rules (RP), (LR^*), $(LR_=)$, and (RLR).



(a) Left subtree predominates. $|\mathsf{bal}(v)|$ decreases.



(b) Right subtree predominates. $|\mathsf{bal}(v)|$ increases.



(c) Left and right subtrees are comparable. $|\mathsf{bal}(v)|$ changes.

Figure 3: The three possible cases for a left propagation with $\mathsf{car}(u) > 0$. The dotted subtree is the state of knowledge of v *before* the propagation. The symmetrical cases where $\mathsf{car}(u) < 0$ are obtained by exchanging the dotted and continuous lines.

3.1 Taking Care of Negative Carries

Lemma 3 ensures that no negative carry can appear in a subtree bearing only positive carries. Negative carries *flow upward* to the root where they vanish. To catch this phenomenon, we shall introduce $Out(u)$, the *number of nodes* of the tree which are *not in the subtree* rooted in u, as proposed by Kessels in [Kes83]. $Out(u)$ is a kind of distance from node u to the root of the tree whose advantage is that it is left unchanged outside the spatial of any rule. Let us denote by NEG:

$$\mathsf{NEG} = \sum_{\mathsf{car}(u)<0} Out(u) \cdot |\mathsf{car}(u)|$$

On a propagation from a node u to its parent v, $\mathsf{car}(u)$ is erased, $\mathsf{car}(v)$ may increase or decrease depending on the value of $\mathsf{car}(u)$, whereas all the other nodes remain unchanged.

If u bears a positive carry, then the carry of v at worst increases, and NEG *at worst decreases.*

If u bears a negative carry, then the carry of v at worst decreases by $|\mathsf{car}(u)|$; but since the carries are weighted by Out and, since Out decreases strictly from sons to parents, then NEG *decreases strictly.*

On a rotation, the only carry to be modified is the one at the root of the rotation. Since Out remains unchanged in each exchanged subtree and since their roots are reliable, then the only term that varies in NEG is the one of the root of the rotation. Since the local height of this root at worst decreases, its carry at worst increases and NEG *at worst decreases.* We conclude then:

Lemma 4 (Negative carries) NEG *does not increase on any rule application and it decreases strictly on any propagation from a node bearing a negative carry.*

3.2 Taking Care of Positive Carries

The three cases in point presented [FIG. 3] show that car and bal seem to be correlated: their respective variations appear to have close magnitudes. We introduce the POS and BAL quantities which respectively measure the positive inconsistency of the local heights and the apparent global imbalance of the tree:

$$\mathsf{POS} = \sum_{\mathsf{car}(u)>0} \mathsf{car}(u) \quad \text{and} \quad \mathsf{BAL} = \sum_{u} |\mathsf{bal}(u)|$$

Propagation Rule. Since Lemma 4 addressed the case of a propagation from a node bearing a negative carry, we shall study here the three last cases which are displayed [FIG. 3]:

- Case 3(a): The carry of v increases by $\mathsf{car}(u)$ and the balance of v decreases to $\mathsf{bal}(v') \geqslant 0$, so:
$$-\mathsf{car}(u) \leqslant \Delta\mathsf{POS} \leqslant 0 \text{ depending on } \mathsf{car}(v), \quad \text{and} \quad \Delta\mathsf{BAL} = -\mathsf{car}(u) < 0$$
- Case 3(b): The local height of v is unchanged and $|\mathsf{bal}(v')| = |\mathsf{bal}(v)| + \mathsf{car}(u)$, so :
$$\Delta\mathsf{POS} = -\mathsf{car}(u) < 0 \quad \text{and} \quad \Delta\mathsf{BAL} = \mathsf{car}(u) > 0$$
- Case 3(c): A careful analysis shows that :
$$\Delta\mathsf{POS} \leqslant -\mathsf{car}(u) + \mathsf{bal}(v) < 0 \quad \text{and} \quad \Delta\mathsf{BAL} = \mathsf{car}(u) - 2\ \mathsf{bal}(v)$$

We conclude then that *for any* $\alpha \geqslant 2$, $(\alpha\ \mathsf{POS} + \mathsf{BAL})$ *decreases strictly on any propagation rule application from a node bearing a positive carry.*

Rotation Rules. A quick look at the rotation rules shows that:

- on applying the rules (RR^*) and (LRR): $\Delta\mathsf{POS} \leqslant 1$ and $\Delta\mathsf{BAL} \leqslant -3$
- on applying the rule $(RR_=)$: $\Delta\mathsf{POS} = \Delta\mathsf{BAL} = 0$

We conclude that *for* $\alpha = 2$, $(2\ \mathsf{POS} + \mathsf{BAL})$ *strictly decreases on* (RR^*) *and* (LRR) *rule application.* To take care of the rule $(RR_=)$ we introduce an ad-hoc quantity RBAL which disregards the apparently balanced nodes:
$$\mathsf{RBAL} = \sum_{|\mathsf{bal}(u)| \geqslant 2} |\mathsf{bal}(u)| - 1$$
It is easy to check that on applying the rule $(RR_=)$: $\Delta\mathsf{RBAL} = -1$. Then RBAL *strictly decreases on* $(RR_=)$ *rule application.*

3.3 A First Variant

The two previous subsections show that NEG does not increase on any rules application. If it remains unchanged, then $2\ \mathsf{POS} + \mathsf{BAL}$ does not increase. And if this latter quantity remains unchanged, then RBAL strictly decreases. This can be summarized as follows:

Property 1 (Liveness property – first variant) $\langle \mathsf{NEG}, 2\ \mathsf{POS} + \mathsf{BAL}, \mathsf{RBAL} \rangle$ *is a* valid variant*: it strictly decreases for the lexicographic order on any rule application and it is greater than* $\langle 0, 0, 0 \rangle$*. Therefore, no infinite sequence of rule applications is possible.*

It can be moreover checked that $\forall u\ \mathsf{localh}(u) \leqslant \mathsf{realh}(u)$ *is a stable predicate through rule applications.* It provides a rough bound on the convergence time of the algorithm for the HRS-trees where initially $\forall u\ \mathsf{localh}(u) \leqslant \mathsf{realh}(u)$ (say $\forall u\ \mathsf{localh}(u) = 1$): at any step, $\mathsf{NEG} \leqslant n^3$, $\mathsf{POS} \leqslant n^2$, $\mathsf{BAL} \leqslant n^2$ and $\mathsf{RBAL} \leqslant n^2$. The algorithm converges on these trees after at most $3n^7$ rule applications.

3.4 A Closer Look at the Rules

The first variant is not precise enough to give a satisfactory bound on the execution time of the algorithm. A tedious case analysis reveals that there exists a fine interaction between NEG, POS, BAL and RBAL.

Lemma 5 *On any rule application the variations of* NEG, POS *and* BAL *satisfy:*

$$\Delta(2\ \mathsf{POS} + \mathsf{BAL}) \leqslant |\Delta\mathsf{NEG}|$$

Proof. The inegality is obtained by a long chain of cases summarized in [Tab. 1 and 2]. We shall study one worst case as an illustration. Let us consider a left propagation from a node u to a node v such that $\mathsf{car}(u) < 0$, $\mathsf{car}(v) \leqslant 0$ and $\mathsf{bal}(v) \geqslant 0$. In this case $|\mathsf{car}(v')| = |\mathsf{car}(v)| + |\mathsf{car}(u)|$, so $\Delta\mathsf{NEG} \leqslant \mathsf{car}(u) < 0$, $\Delta\mathsf{POS} = 0$, and $\Delta\mathsf{BAL} = -\mathsf{car}(u) > 0$. Thus $\Delta(2\ \mathsf{POS} + \mathsf{BAL}) \leqslant |\Delta\mathsf{NEG}|$ holds. □

Corollary 1 $\langle 2\ (\mathsf{NEG} + \mathsf{POS}) + \mathsf{BAL}, \mathsf{RBAL}\rangle$ *is a valid variant for the algorithm.*

Moreover the variations of RBAL and BAL are correlated:

Lemma 6 *If the absolute values of the balances of q nodes are modified on a rule application, the variations of* BAL *and* RBAL *satisfy:*

$$\Delta\mathsf{RBAL} \leqslant \Delta\mathsf{BAL} + q$$

Proof. We can assume $q = 1$ w.l.o.g.. Let b and b' be the initial and final absolute values of the modified balance. Three cases have to be distinguished: (1) if $b, b' \geqslant 1$, then $\Delta\mathsf{RBAL} = \Delta\mathsf{BAL}$; (2) if $b \geqslant 1$ and $b' = 0$, then $\Delta\mathsf{RBAL} = \Delta\mathsf{BAL} + 1$; (3) if $b = 0$ and $b' \geqslant 1$, then $\Delta\mathsf{RBAL} = \Delta\mathsf{BAL} - 1$. □

As each rule modifies at most 3 nodes, on each rule application $\Delta\mathsf{RBAL} \leqslant \Delta\mathsf{BAL} + 3$ holds. Let us study the variations of $\vartheta = 3\ \{2\ (\mathsf{NEG} + \mathsf{POS}) + \mathsf{BAL}\} + \{2\ (\mathsf{NEG} + \mathsf{POS}) + \mathsf{RBAL}\}$ on a rule application:

- If the rule is not the right rotation ($RR_{=}$), then the proof of Corollary 1 ensures that $\Delta(2\ (\mathsf{NEG} + \mathsf{POS}) + \mathsf{BAL}) < 0$. Then:
  $$\Delta\vartheta \leqslant 3 \times -1 + \Delta\{2\ (\mathsf{NEG} + \mathsf{POS}) + \mathsf{BAL} + (\mathsf{RBAL} - \mathsf{BAL})\} \leqslant -3 - 1 + 3 < 0$$
- If the rule is ($RR_{=}$), then $\Delta\mathsf{NEG} = \Delta\mathsf{POS} = \Delta\mathsf{BAL} = 0$ and $\Delta\mathsf{RBAL} = -1$, thus $\Delta\vartheta < 0$.

Thus ϑ is a valid variant. A careful analysis of RBAL shows that $\Delta\mathsf{RBAL} \leqslant \mathsf{BAL} + 2$ on any rule application. The same analysis leads finally to:

Theorem 1 (Variant) $6\ (\mathsf{NEG} + \mathsf{POS}) + 2\ \mathsf{BAL} + \mathsf{RBAL}$ *is a valid variant for the algorithm.*

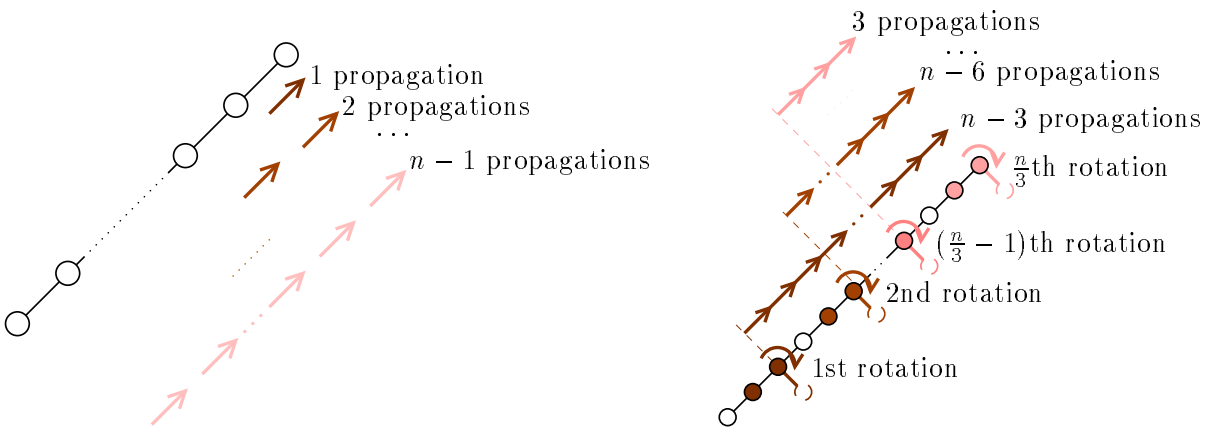
Let c_{max} and b_{max} respectively denote the maximum absolute values of car and bal initially. We get the following worst convergence time bound:

Corollary 2 (Worst convergence time) *The algorithm applies at most* $6\ \mathsf{c}_{max} n(n+1) + 3\ \mathsf{b}_{max} n$ *rules to rebalance any arbitrary HRS-tree with any initial state.*

For instance at most $6(n^2 + n)$ rules are necessary to rebalance a tree of size n whose node have their initial local height set to 1.

3.5 Examples for Worst Convergence Times

We propose two executions with $\Omega(n^2)$ rule applications :



(a) Initially each node has its local height set to 1: $\mathsf{c}_{max} = 1$ and $\mathsf{b}_{max} = 0$

(b) Initially the local heights are the real heights: $\mathsf{c}_{max} = 0$ and $\mathsf{b}_{max} = n - 1$

Figure 4: Two examples of $\Theta(n^2)$ rules executions highlighting the importance of the two terms $6\ \mathsf{c}_{max} n(n+1)$ and $3\ \mathsf{b}_{max} n$.

- [Fig. 4(a)] shows a scheme of $\Omega(n^2)$ propagations in a linear tree where initially each node believes to be a leaf: each node, beginning by the son of the root and finishing by the leaf, informs in turn the root of its presence.
- [Fig. 4(b)] shows a scheme of $\Omega(n^2)$ propagations and $\Omega(n)$ rotations in a linear tree where initially each node knows its own real height: In turn, one node out of three, beginning by the leaf and finishing by the root, executes a rotation and informs the root of the modifications of its local height.

An amazing fact is that we could not find any execution scheme involving more than $O(n)$ rotations. It is tempting to relate this to the two parts of the variant: $6\ \mathsf{c}_{max} n(n+1) = O(n^2)$ may be related to the number of propagations, and $3\ \mathsf{b}_{max} n = O(n)$ to the number of rotations. We therefore conjecture that *at most $O(n)$ rotations may be applied*. It is likely that such a bound would certainly *shed a new light on the intimate structure of AVL trees.*

4 Experimental Evaluation

The goal of this section is to present some experimental results on the pratical behavior of our rebalancing scheme. Observe that the rules have exclusive guards with respect to the node u according to the notations of Section 2.3. Therefore, our simulation repeatedly picks up a node u in the tree at random and applies to it the appropriate rule if any until no rule applies anywhere. As the number of binary trees of size n and the number of possible executions per tree grow exponentially, it is hopeless to simulate every possible behavior (the number of trees of size $n = 15$ is $9,694,845$, and $6,564,120,420$ for $n = 20$).

Experimental Worst Convergence Time Analysis. First, we concentrate on small trees and record for each tree the worst convergence time measured on a large number of simulations. The results are displayed on [Fig. 5]. The diagrams are based on the following tree enumeration: we enumerate the binary trees of fixed size n simply by enumerating recursively all the possible right subtrees for all the possible left subtrees and we index each tree by its *rank in this enumeration*. The advantage of this method is that it respects the recursive structure of binary trees; in particular trees which have close indexes have close shapes. We execute between $1,000$ and $10,000$ simulations on each tree depending on their size n (a total of $16,796,000$ simulations for $n = 10$).

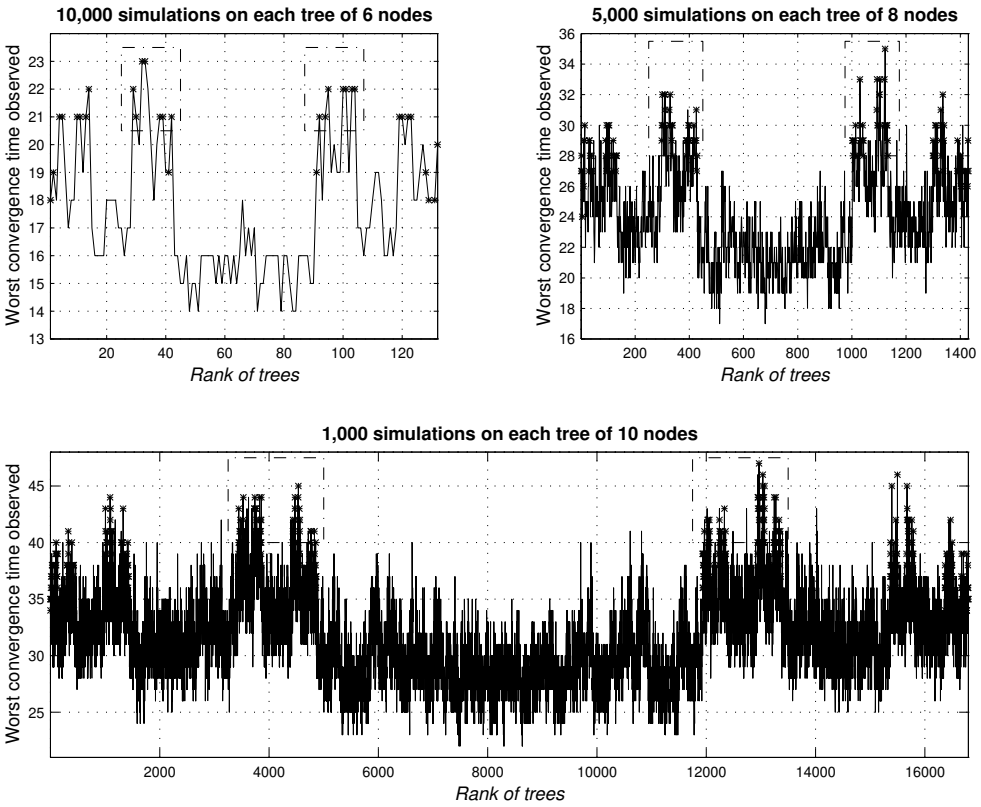


Figure 5: Worst convergence times observed on $10,000$ executions over each tree of size $n = 6$, on $5,000$ executions per tree for $n = 8$ and on $1,000$ executions per tree for $n = 10$. The '$*$' point out the linear trees. The dashed rectangles highlight the worst convergence time localisations.

It appears that these diagrams have fractal structures: the diagram for $n = 6$ appears in the diagram where $n = 8$ which appears in the diagram where $n = 10$. As the rank respects the recursive structure of trees, this means that our rebalancing algorithm is somehow continous with respect to the shape of the tree. A closer look at the enumeration shows that the central part of the diagram corresponds to the highly balanced trees and the peaks on the sides to the linear trees as shown [Fig. 5].

Thus, linear trees seem to be the most difficult ones to rebalance. More precisely, the second pair of peaks from the borders of the diagrams [Fig. 5] appears to be always the highest: the study of the ranks of those trees reveals that their shapes are really close to the *regular zigzag tree*, i.e. the linear trees where each son of a right son (resp. left son) is a left son (resp. right son) [Fig. 6].

In fact further intensive simulations (not presented here) show that the regular zigzag trees appear to be the most difficult to rebalance among the linear trees.

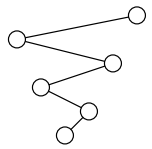


Figure 6: A regular zigzag of size $n = 6$.

Again, intensive simulations on the regular zigzag trees with upto $5,000$ nodes yield a *worst convergence time of $\gamma.n$ rules applications, where $\gamma \cong 4$* [Fig. 7]. The quadratic executions exhibited in Section 3.5 are thus likely to be extremely singular.

Experimental Average Convergence Time Analysis. A more precise analysis of the convergence time distribution confirms the above assumption. The result of the simulations is shown [Fig. 7].

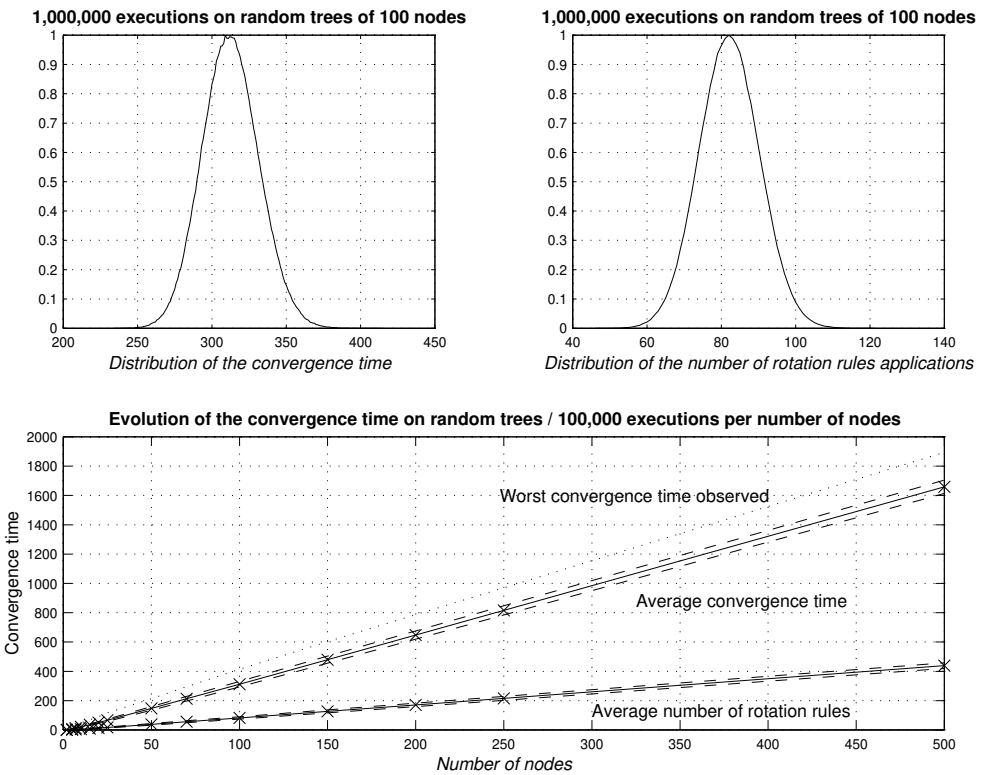


Figure 7: Average convergence time observed on $100,000$ executions over random trees of size $n = 2,\ 5,\ 7,\ 10,\ 15,\ 20,\ 25,\ 50,\ 70,\ 100,\ 150,\ 200,\ 250,\ 500$. The dotted lines represent the dispersion intervals.

The behavior of our algorithm appears to be very smooth : the convergence time seems to follow a "Gaussian-like" distribution as well as the number of rotation rule applications. The average convergence time appears to be $\alpha.n$ with $\alpha \cong 3.5$ with a standard deviation of $\beta\sqrt{n}$ with $\beta \cong 4.1$. This Gaussian-like distribution confirms the previous result on practical worst cases: the

probability of convergence time greater than $4.n$ tends to 0 as n grows. Thus, *our scheme rebalances in practice a arbitrary binary tree after at most $O(n)$ rule applications.*

The first theorical studies we made on linear trees revealed that propagation rules and rotation rules could not be considered independently: if we only consider propagation rules (which are preponderant as observed above), we have prooved that the average convergence time towards complete knowledge is $\Theta(n\sqrt{n})$. This is much more than the $\Theta(n)$ steps observed. In fact rotations reduce the height of the tree and then accelerate convergence towards total knowledge. Unfortunately further inquisitions did not lead to more precise theoretical estimation concerning the convergence time distribution. Note however that the analysis of the standard sequential AVL algorithm is still one of the outstanding problems in the analysis of algorithms according to [SF96].

5 Conclusion

This paper presents a fine-grained, distributed approach to the problem of managing concurrent request in AVL search trees. Our major contribution is to demonstrate that an abstract view of the problem yields an algorithm simpler than previously known ones. In our view, inserting and deleting keys in a search tree is simply considered as an external perturbation made by anonymous *mutators* on which one has no control. Rebalancing the tree back to an AVL shape is the job of an asynchronous daemon which competes with the mutators. It repeatedly selects a local part of the structure in a nondeterministic way, locks it, reshapes it if needed according to the information locally available, and unlocks it.

The key of our fine-grained approach is thus to completely uncouple the rebalancing process with respect to the perturbations. This yields a quite robust and flexible scheme. It supports multiple concurrent daemons and arbitrary perturbations. In particular, one could even imagine that the scheme is applied concurrently with other restructuring schemes on the same data structure! Faults can be tolerated, as long as they preserve the basic underlying invariants (well-founded search tree). In fact, such faults can be seen as yet another kind of perturbation.

The price to pay is that no information on the perturbation can be used to guide the rebalancing process optimally. In the worst case, rebalancing an arbitrary tree with n nodes necessitates $O(n^2)$ steps, instead of the $O(n.\log(n))$ steps of Larsen [Lar94]. This is because we have to explicitly flow information upwards in the tree, and we cannot guarantee that this is done in an efficient way in all cases. The fine-grained, distributed feature of our rebalancing scheme makes its theoretical analysis rather difficult. Yet, an extensive experimentation provides strong evidence that quadratic behaviors are extremely exceptional. A linear average convergence time seems very likely. Unfortunately, a rigorous proof of such a conjecture is out of reach of our current methods.

In fact, this fine-grained scheme yields a useful basis to design more complex algorithms by restricting the scheduling of the rules to "efficient" ones. It turns out that many existing algorithms previously proposed in the literature can be seen as such specializations (up to suitable remaning of the registers). The extended version of this paper shows that this is the case for the algorithms of Ellis [Ell80] Kessels [Kes83], Nurmi and al. [NSSW87, NSSW92], and Larsen [Lar94]. The initial sequential AVL algorithm even appears as a limit case. As our scheme has been proved correct (safety and liveness), any non-deadlocking specialization of it yields a correct algorithm, too.

These results illustrate the power gained by taking a more abstract view of such concurrent manipulations of trees. It is very likely that quite a quite similar approach can be applied to other rebalancing schemes. Some preliminary studies have been done for 2-3 trees which seem to confirm this idea, but much work is still needed in this direction.

References

[AL62] G. M. Adel'son-Vel'skiĭ and E. M. Landis. An algorithm for the organization of the information. *Soviet Mathematics Doklady*, (3):1259–1263, 1962.

[BGM95] L. Bougé, J. Gabarró, and X. Messeguer. Concurrent avl revisited : self-balancing distributed search. Research Report RR95-45, LIP, ENS Lyon, France, 1995. Available at URL http://www.ens-lyon.fr/LIP.

[BS77] R. Bayer and M. Schkolnick. Concurrency of operations on B-trees. *Acta Informatica*, 9(1):1–21, 1977.

[DLM+78] 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, 1978.

[Ell80] C. S. Ellis. Concurrent search and insertion in AVL trees. *IEEE Trans. Comp.*, C-29(9):811–817, 1980.

[Fra80] N. Francez. Distributed termination. *ACM Trans. Progr. Languages and Systems*, 2:42–55, 1980.

[Kes83] J. L. S. Kessels. On the fly optimisation of data structures. *Comm. ACM*, 26(11):895–901, 1983.

[KL80] H. T. Kung and P. L. Lehman. Concurrent manipulation of binary serch trees. *ACM Trans. Database Systems*, 5(3):354–382, 1980.

[Knu73] D. E. Knuth. *The art of computer programming, Fundamental algorithms.* Addison-Wesley, 1973.

[Lar94] K. S. Larsen. AVL trees with relaxed balance. In *Proc. Int. Parallel Processing Symp.*, pages 888–893. IEEE Comp. soc., 1994.

[NSS96] O. Nurmi and E. Soisalon-Soininen. Chromatic binary serach trees. *Acta informatica*, 33(6):547–557, 1996.

[NSSW87] O. Nurmi, E. Soisalon-Soininen, and D. Wood. Concurrency control in database structures with relaxed balance. *ACM Symp. Princ. Distributed Systems*, pages 170–176, 1987.

[NSSW92] O. Nurmi, E. Soisalon-Soininen, and D. Wood. Concurrent balancing and updationg of AVL trees. Technical Report 1992 ITKO-B76, Helsinki Univ. of Techn., Dept. Comp. Sciences, 1992.

[SF96] R. Sedgewick and Ph. Flajolet. *Analysis of algorithms.* Addison-Wesley, 1996.

A Appendix: Exhaustive Case Analysis

Table 1: Exhaustive case analysis for the rule (LP) according to the notation of Section 2.3.

Propagation rule – The cases				ΔNEG	ΔPOS	ΔBAL
$\mathsf{car}_u > 0$	$\mathsf{car}_v \geqslant 0$		$\mathsf{car}_u \leqslant \mathsf{bal}_v$	0	0	$-\mathsf{car}_u < 0$
			$\mathsf{bal}_v \leqslant 0$	0	$-\mathsf{car}_u < 0$	$\mathsf{car}_u > 0$
			$0 < \mathsf{bal}_v < \mathsf{car}_u$	0	$-\mathsf{car}_u + \mathsf{bal}_v < 0$	$\mathsf{car}_u - 2\mathsf{bal}_v \lesseqgtr 0$
	$\mathsf{car}_v < 0$	$\mathsf{car}_{v'} < 0$	$\mathsf{car}_u \leqslant \mathsf{bal}_v$	$-\mathsf{car}_u \cdot Out_v \leqslant 0$	$-\mathsf{car}_u < 0$	$-\mathsf{car}_u < 0$
			$\mathsf{bal}_v \leqslant 0$	0	$-\mathsf{car}_u < 0$	$\mathsf{car}_u > 0$
			$0 < \mathsf{bal}_v < \mathsf{car}_u$	$-\mathsf{bal}_v \cdot Out_v \leqslant 0$	$-\mathsf{car}_u < 0$	$\mathsf{car}_u - 2\mathsf{bal}_v \lesseqgtr 0$
		$\mathsf{car}_{v'} \geqslant 0$	$\mathsf{car}_u \leqslant \mathsf{bal}_v$	$\mathsf{car}_v \cdot Out_v \leqslant 0$	$-\mathsf{car}_u + \mathsf{car}_{v'} < 0$	$-\mathsf{car}_u < 0$
			$\mathsf{bal}_v \leqslant 0$	$\mathsf{car}_v \cdot Out_v \leqslant 0$	$-\mathsf{car}_u + \mathsf{car}_{v'} < 0$	$\mathsf{car}_u > 0$
			$0 < \mathsf{bal}_v < \mathsf{car}_u$	$\mathsf{car}_v \cdot Out_v \leqslant 0$	$-\mathsf{car}_u + \mathsf{car}_{v'} < 0$	$\mathsf{car}_u - 2\mathsf{bal}_v \lesseqgtr 0$
$\mathsf{car}_u < 0$	$\mathsf{car}_v \leqslant 0$		$\mathsf{bal}_v \leqslant \mathsf{car}_u$	$\mathsf{car}_u \cdot Out_u < 0$	0	$\mathsf{car}_u < 0$
			$0 \leqslant \mathsf{bal}_v$	$\mathsf{car}_u \cdot (Out_u - Out_v) < 0$	0	$-\mathsf{car}_u > 0$
			$\mathsf{car}_u < \mathsf{bal}_v < 0$	$\mathsf{car}_u \cdot (Out_u - Out_v) + \mathsf{bal}_v \cdot Out_v < 0$	0	$-\mathsf{car}_u + 2\mathsf{bal}_v \lesseqgtr 0$
	$\mathsf{car}_v > 0$	$\mathsf{car}_{v'} \geqslant 0$	$\mathsf{bal}_v \leqslant \mathsf{car}_u$	$\mathsf{car}_u \cdot Out_u < 0$	0	$\mathsf{car}_u < 0$
			$0 \leqslant \mathsf{bal}_v$	$\mathsf{car}_u \cdot Out_u < 0$	$\mathsf{car}_u < 0$	$-\mathsf{car}_u > 0$
			$\mathsf{car}_u < \mathsf{bal}_v < 0$	$\mathsf{car}_u \cdot Out_u < 0$	$\mathsf{car}_u - \mathsf{bal}_v < 0$	$-\mathsf{car}_u + 2\mathsf{bal}_v \lesseqgtr 0$
		$\mathsf{car}_{v'} < 0$	$\mathsf{bal}_v \leqslant \mathsf{car}_u$	$\mathsf{car}_u \cdot Out_u - \mathsf{car}_{v'} \cdot Out_v < 0$	$-\mathsf{car}_v < 0$	$\mathsf{car}_u < 0$
			$0 \leqslant \mathsf{bal}_v$	$\mathsf{car}_u \cdot Out_u - \mathsf{car}_{v'} \cdot Out_v < 0$	$-\mathsf{car}_v < 0$	$-\mathsf{car}_u > 0$
			$\mathsf{car}_u < \mathsf{bal}_v < 0$	$\mathsf{car}_u \cdot Out_u - \mathsf{car}_{v'} \cdot Out_v < 0$	$-\mathsf{car}_v < 0$	$-\mathsf{car}_u + 2\mathsf{bal}_v \lesseqgtr 0$

Rotations rules – The cases			ΔNEG	ΔPOS	ΔBAL	ΔRBAL
Rule (RR*)	$\mathsf{car}_v \geqslant 0$		0	1		
$\mathsf{bal}_u > 0, \mathsf{bal}_v \geqslant 2$	$\mathsf{car}_v < 0$		$-Out_v \leqslant 0$	0		
	$\mathsf{bal}_u + 1 \leqslant \mathsf{bal}_v$				$-\mathsf{bal}_u - 2 \leqslant -3$	$\leqslant -1$
	$\mathsf{bal}_u + 1 \geqslant \mathsf{bal}_v$				$-\mathsf{bal}_v - 1 \leqslant -3$	$\leqslant -1$
Rule ($\mathbf{RR_=}$)	$\mathsf{bal}_u = 0, \mathsf{bal}_v \geqslant 2$		0	0	0	-1
Rule (LRR)	$\mathsf{car}_v \geqslant 0$		0	1		
$\mathsf{bal}_u < 0, \mathsf{bal}_v \geqslant 2$	$\mathsf{car}_v < 0$		$-Out_v \leqslant 0$	0		
	$\mathsf{bal}_w \geqslant 0$	$\mathsf{bal}_{v'} \geqslant 0$			$-\mathsf{bal}_w - 3 \leqslant -3$	$\leqslant -1$
		$\mathsf{bal}_{v'} \leqslant 0$			$-\mathsf{bal}_v - 1 \leqslant -3$	$\leqslant -2$
	$\mathsf{bal}_w < 0$	$\mathsf{bal}_{u'} \geqslant 0$			$\mathsf{bal}_u - 2 \leqslant -3$	$\leqslant -2$
		$\mathsf{bal}_{u'} \leqslant 0$			$\mathsf{bal}_w - 3 \leqslant -3$	$\leqslant -1$

Table 2: Exhaustive case analysis for the rotation rules according to the notation of Section 2.3