Laboratoire de l'Informatique du Parallélisme

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

Height-relaxed AVL rebalancing: A unified, fine-grained approach to concurrent dictionaries

Luc Bougé, Joaquim Gabarró, Xavier Messeguer and Nicolas Schabanel

March 1998

Research Report N° RR98-18

École 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@ens-lyon.fr

Height-relaxed AVL rebalancing: A unified, fine-grained approach to concurrent dictionaries

Luc Bougé, Joaquim Gabarró,
Xavier Messeguer and
Nicolas Schabanel

March 1998

Abstract

We address the concurrent rebalancing of almost balanced binary search trees (AVL trees). Such a rebalancing 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. Based on the basic rebalancing framework, we describe algorithms to manage concurrent insertion and deletion of keys. Finally, this approach is used to emulate other well known concurrent AVL algorithms.
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?

This paper has been submitted for publication in Acta Informatica.

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

Résumé

Ce rapport présente un algorithme concurrent pour la gestion dynamique d'un arbre binaire de recherche équilibré (arbres 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 noeuds, dirigée par quelques règles d'évolutions locales. Cette approche mène à un algorithme bien plus simple que les solutions précédentes connues. Des règles complémentaires permettent de plus de gérer les insertions et suppressions concurrentes.
Ce résultat permet en particulier 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".

Ce papier a été soumis pour publication à Acta Informatica.

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

Height-relaxed AVL rebalancing: A unified, fine-grained approach to concurrent dictionaries

Luc Bougé* Joaquim Gabarró† Xavier Messeguer† Nicolas Schabanel*

March 9, 1998

Abstract

We address the concurrent rebalancing of almost balanced binary search trees (AVL trees). Such a rebalancing 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. Based on the basic rebalancing framework, we describe algorithms to manage concurrent insertion and deletion of keys. Finally, this approach is used to emulate other well known concurrent AVL algorithms.

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, Emulation.*

1 Introduction

Search trees are the core 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-Velskiĭ 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

This work has been partly supported by the French CNRS Coordinated Research Program on Parallelism, Networks and Systems PRS, the EU HCM Program under Contract ERBCHGECT 920009 and the EU Esprit BRA Program ALCOM II 7141 and ESPRIT LTR Project no. 20244 —ALCOM-IT, the Spanish DGICYT under grant PB95-0787 (project KOALA), the Spanish CICIT TIC97-1475-CE

*LIP, ENS Lyon, 46 Allée d'Italie, F-69364 Lyon Cedex 07, France. Contact: {Luc.Bouge,Nicolas.Schabanel}@ens-lyon.fr

†LSI, UPC, Campus Nord–Mòdul C6, C/ Jordi Girona Salgado 1–3, E-08034 Barcelona, Spain. Contact: {gabarro,peypoch}@lsi.upc.es

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 (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 [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". This approach has been used by Nurmi and Soisalon-Soininen [NSS96a] to define *Chromatic Trees* by relaxing the rules of Red-Black Trees. Two surveys on the topic are [SSW97, GM97].

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.

The crux of our contribution is the following amazing remark.

> *Applying the original AVL rebalancing rules to an arbitrary tree (even very unbalanced!) in an arbitrary order, does eventually reshape it into an AVL tree, even in the presence of incomplete information on the heights of the subtrees.*

This is the reason why we have coined the term *AVL rebalancing* applied to a *Height-Relaxed Search tree* (HRS-tree, for short) which generalizes the original AVL scheme. The Rotation Rules are exactly the same, except that they are applied to arbitrarily unbalanced trees.

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. 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, in any order!

The price to pay for this "fine-grained approach" is that $O(n^2)$ 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 describes the basic data structure we use, so called *Height-Relaxed Search Trees* (HRS-trees). Section 3 describes the daemons. Section 4 proves the (partial) correctness. It is easy: once no daemon can work any longer, the tree is balanced. On the other hand, convergence turns out to be much more difficult, as shown in Section 5. Section 6 reports on experimental measures of the average behavior. Section 7 describes an algorithm to manage concurrent insertions and deletions. In Section 8 shows that our scheme can emulate other concurrent extension of the original AVL scheme by enforcing specialized schedules on the behavior of the daemons. Finally, all results are summarized in Section 9.

Remark: A preliminary version of this work has appeared in [BGMS97]. It includes the results presented up to Section 5. The remaining material is new.

2 HRS-trees: a data structure for concurrent AVL rebalancing

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 explicitly flowed 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 any longer (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 excepted 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 incorrect!) 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 suggested 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 kind 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 steals the cycles left idle by the clients to reorganize the structure optimally.

Let u be a node of the search tree. We respectively denote by $u{\to}\mathit{p}$, $u{\to}\mathit{ls}$, $u{\to}\mathit{rs}$ 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:

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

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:

lefth(u) **and 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*:

$$\mathsf{lefth}(u) = 0 \text{ (resp. } \mathsf{righth}(u) = 0)$$

for any node u with an empty left (resp. right) son. In other words, all values may be arbitrary except the ones at the leaves: If I have no son, then I know it.

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

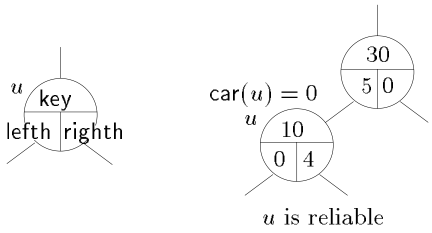
localh(u) is the *apparent local height* of u, as computed from the two previous registers:

$$\mathsf{localh}(u) = 1 + \max\left(\mathsf{lefth}(u), \mathsf{righth}(u)\right)$$

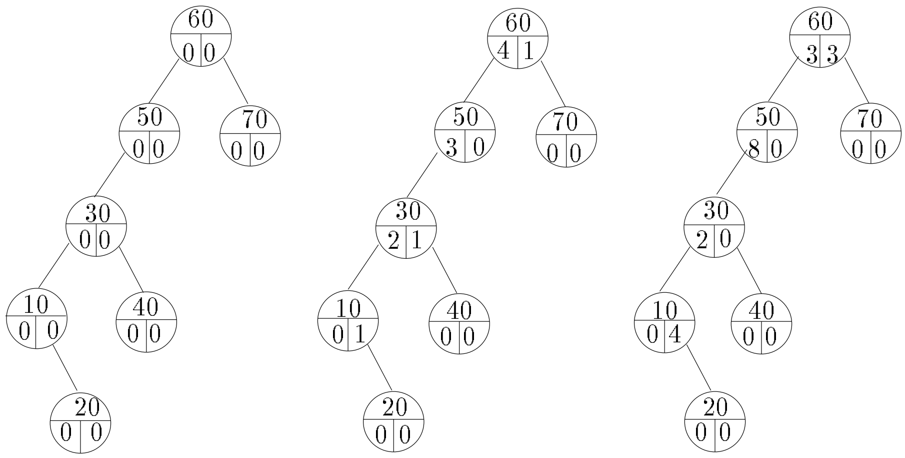
car(u)**,** the *carry* of u, is the gap of knowledge between u and its parent:

$$\mathsf{car}(u) = \begin{cases} \mathsf{lefth}(u{\to}\mathit{p}) - \mathsf{localh}(u) & \text{if } u \text{ is the left son of its parent} \\ \mathsf{righth}(u{\to}\mathit{p}) - \mathsf{localh}(u) & \text{if } u \text{ is the right son of its parent} \end{cases}$$

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 $\mathsf{car}(\mathsf{root}) = 0$.



(a) HRS-tree nodes.



(b) All the nodes "believe" they are leaves.

(c) All the nodes have faithful information.

(d) An HRS-tree with no special structure.

Figure 1: Examples of HRS-trees.

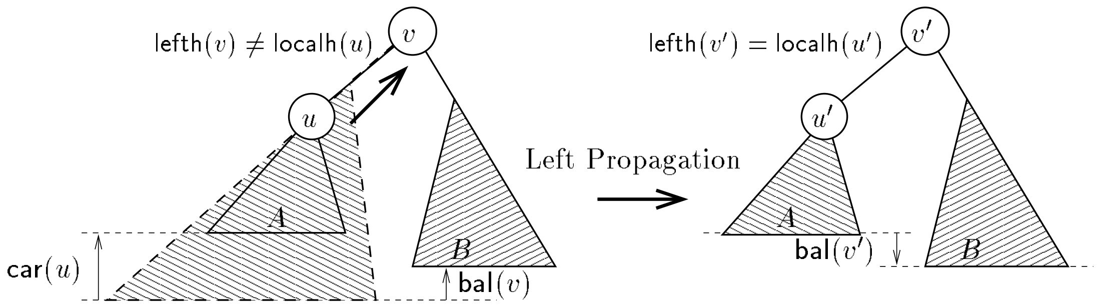


Figure 2: Propagation Rules: Rule (LP), left propagation if $\mathsf{car}(u) \neq 0$

$\mathsf{bal}(u)$ 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 $|\mathsf{bal}(u)| \leqslant 1$.

On [Fig. 1], some examples of HRS-trees are displayed. On [Fig. 1(a)], a graphical notation for lefth, righth and key registers is given, and examples of *reliability* and *apparent balance* are shown. On [Fig. 1(b)], a tree with really bad local information is given: all nodes "believe" they are leaves, nodes are unreliable but apparently balanced ($\mathsf{car}(u) = -1$ and $\mathsf{bal}(u) = 0$, for any node). This case appears when the tree is built by a series of consecutive insertions with no intermediate update. Each node "remembers" the instant when it was attached at the tree as a new leave. The [Fig. 1(c)] shows a tree with reliable local information ($\mathsf{car}(u) = 0$, for any u). In this case, quantity localh coincides with realh. Finally, [Fig. 1(d)] displays an HRS-tree with no special structure. It could appear at some intermediate step of the rebalancing algorithm.

The following fact expresses that this extension of the classical notion of an AVL tree behaves properly.

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

3 Ruling the daemons' behavior

3.1 Propagation rule

This rule 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 present the variations of the lefth and righth registers, from which the registers localh, car and bal are computed.

Rule (LP) – Left Propagation

Guard: Node u is the left son of node v and u is not reliable: $\mathsf{car}(u) \neq 0$

Action: the apparent left height of v is updated [Fig. 2]:

$$\mathsf{lefth}(v') = \mathsf{localh}(u)$$

Note that: $\mathsf{bal}(v') = \mathsf{bal}(v) - \mathsf{car}(u)$.

Spatial scope: Node u and its parent $v = u \to p$.

The Right Propagation Rule (RP) where u is the right son of v, can be deduced symmetrically from Rule (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.

3.2 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, node u is reliable, $\mathsf{bal}(u) > 0$ and $\mathsf{bal}(v) \geqslant 2$

Action: Nodes u and v execute a right rotation [Fig. 3(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: Node u and its parent $v = u {\rightarrow} p$.

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 (RR$_=$) – Right Rotation, Balanced case

Guard: Node u is the left son of node v, node u is reliable, $\mathsf{bal}(u) = 0$ et $\mathsf{bal}(v) \geqslant 2$.

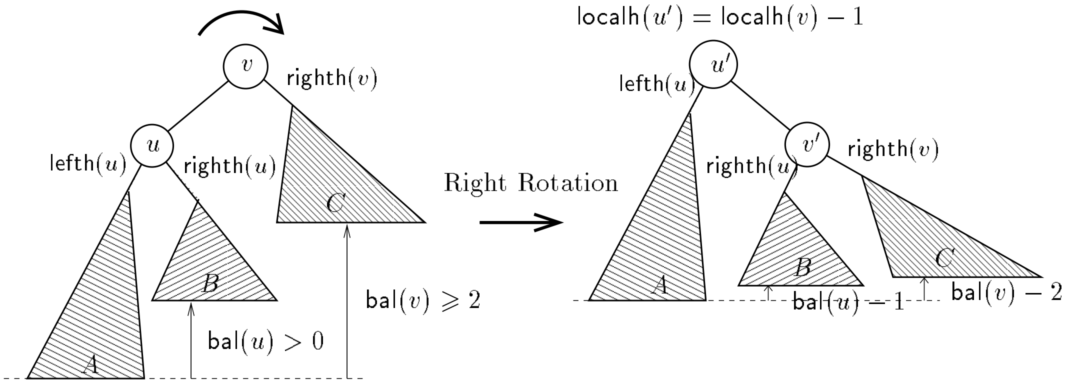
Action: Nodes u and v execute a right rotation [Fig. 3(b)] with the obvious updating:

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

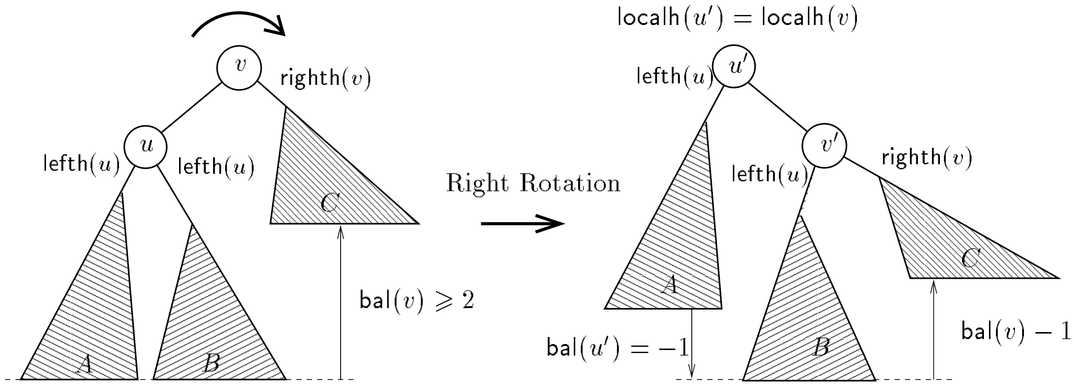
Note that: $\mathsf{localh}(u') = \mathsf{localh}(v)$, so $\mathsf{car}(u') = \mathsf{car}(v)$.

Spatial scope: Node u and its parent $v = u {\rightarrow} p$.

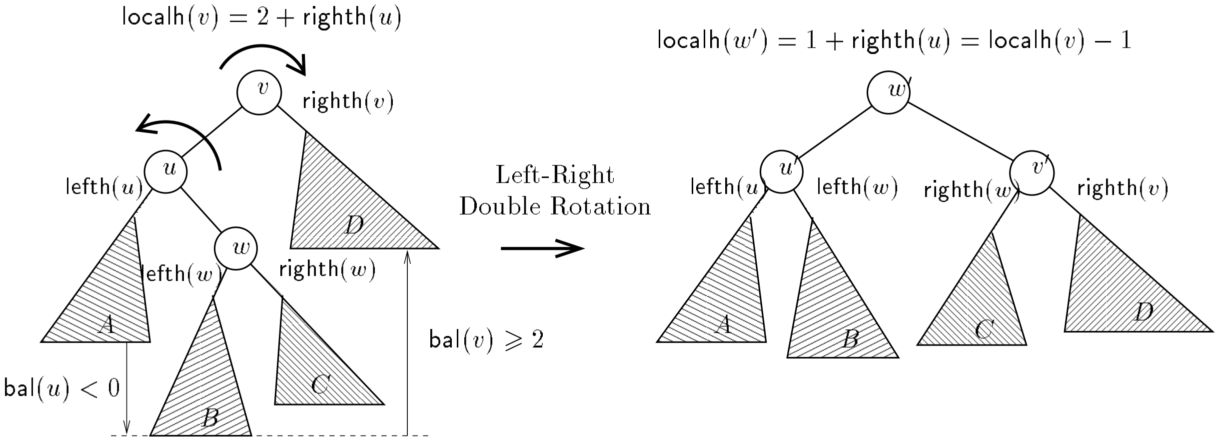
The rule (LR$_=$), where u is the left son of v and, u and v execute a left rotation when $\mathsf{bal}(u) = 0$ and $\mathsf{bal}(v) \leqslant -2$, is obtained as before, symmetrically from (RR$_=$).



(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 3: The rotation rules

Rule (LRR) – Left-Right double Rotation

Guard: Node w is the right son of the left son u of node v, nodes w and u are reliable, $\mathsf{bal}(u) < 0$ et $\mathsf{bal}(v) \geqslant 2$.

Action: Nodes u, v and w execute a left-right double rotation [Fig. 3(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: Node 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 $\mathsf{bal}(v) \leqslant -2$.

On [Fig. 4], a possible behavior of daemons is shown. They start acting over tree given in [Fig. 1(d)]. We sketch two possible evolutions. In the first case, applying only propagation rules, we get an AVL tree. In the second case (the last tree in the figure) we can get a really unbalanced tree with bad local information. It is not so clear that this tree is "closer" to an AVL tree than the initial one: it could even happen that daemons lead to a never ending reshaping process! The following sections address these questions.

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, then T is an AVL.*

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 cannot increase and v is thus even more overestimated by its parent. The same remark also applies to the rotation rules because a rotation rule cannot increase the height of the root of the rotation, and it leaves the other nodes reliable. □

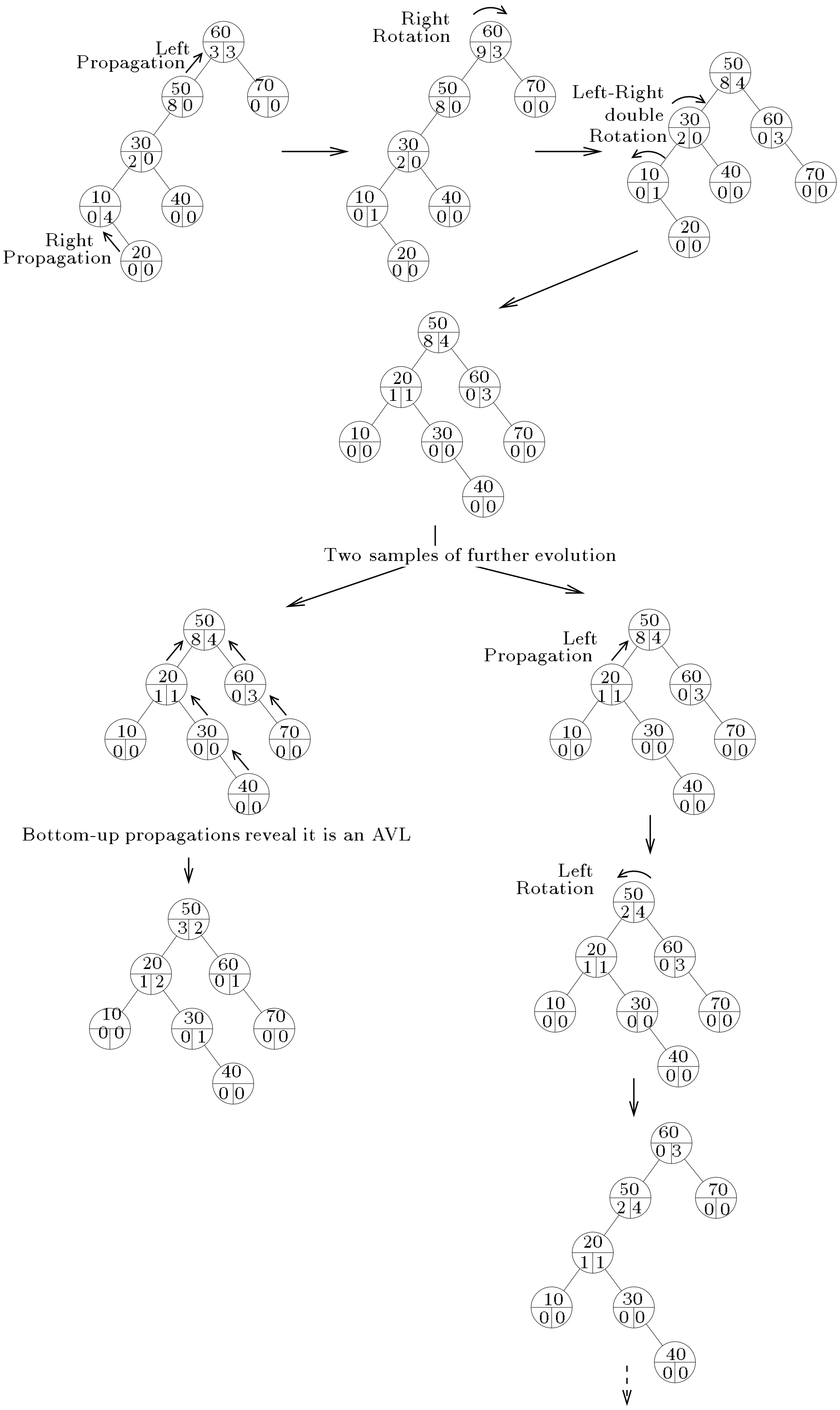
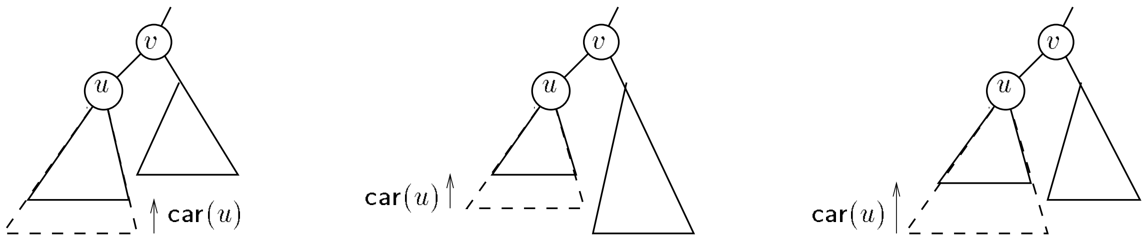


Figure 4: Some examples of rule applications.



(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 5: 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.

5 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 any longer and prove then that at most $\Theta(n^2)$ successive rules may be applied, where n is the number of nodes of the tree. By Lemma 2, the resulting tree is an AVL: the algorithm rebalances thus any arbitrary tree in at most $\Theta(n^2)$ steps.

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. Essentially, 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 on [Fig. 5].

On [Fig. 5(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. 5(b)], the application of the rule (LP) increases the imbalance of v (by $|\mathsf{car}(u)|$). On [Fig. 5(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. 3(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 independently considered 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 hold trivially for the symmetrical rules (RP), (LR_*), ($LR_=$), and (RLR).

5.1 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]. Quantity $Out(u)$ is a kind of distance from node u to the root of the tree, which is left unchanged outside the spatial scope of any rule. We introduce the NEG quantity, which measures the total negative carry of a tree.

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

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 cannot increase, and NEG *cannot increase*.
- If u bears a negative carry, then the carry of v decreases by $|\mathsf{car}(u)|$; but since the carries are weighted by Out and, since Out strictly decreases from sons to parents, then NEG *strictly decreases*.

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 cannot increase, its carry cannot decrease. Thus, negative carries cannot increase in absolute value, and NEG *cannot increase*.

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

5.2 Taking care of positive carries

The three cases in point presented on [Fig. 5] 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 addresses the case of a propagation from a node bearing a negative carry, we shall study here the three last cases which are displayed [Fig. 5]:

Case 5(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 5(b). The local height of v is left 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 5(c). A careful analysis shows that:

$$\Delta\mathsf{POS} \leqslant -\mathsf{car}(u) + \mathsf{bal}(v) < 0 \quad \text{and} \quad \Delta\mathsf{BAL} = \mathsf{car}(u) - 2.\mathsf{bal}(v)$$

> *Therefore, 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 the following effects.

Rules (RR$_*$) and (LRR). On applying these rules, $\Delta\mathsf{POS} \leqslant 1$ and $\Delta\mathsf{BAL} \leqslant -3$.

Rule (RR$_=$). In this case, $\Delta\mathsf{POS} = \Delta\mathsf{BAL} = 0$.

> *Thus, the right choice is $\alpha = 2$: $(2.\mathsf{POS} + \mathsf{BAL})$ strictly decreases on applying the rules (RR$_*$) and (LRR) at a node bearing a positive carry, and it left unchanged by on applying rule (RR$_=$).*

To fix the remaining case, namely 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$_=$): $\Delta\mathsf{RBAL} = -1$. Thus, RBAL strictly decreases on applying the rule (RR$_=$).

5.3 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 5 (Liveness property – first variant) *The integer quantity $\langle \mathsf{NEG}, 2.\mathsf{POS}{+}\mathsf{BAL}, \mathsf{RBAL}\rangle$ 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. This provides a rough bound on the convergence time of the algorithm for the HRS-trees with n nodes where initially $\forall u\ \mathsf{localh}(u) \leqslant \mathsf{realh}(u)$ (say, $\forall u\ \mathsf{localh}(u) = 1$), as $\forall u\ \mathsf{realh}(u) \leqslant n$. 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.

5.4 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 6 *On any rule application, the variations of NEG, POS and BAL satisfy:*

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

Proof. The inequality is obtained by a long chain of cases summarized in [TAB. 1 and 2]. We shall study one of the worst cases 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 7 $\langle 2.(\mathsf{NEG} + \mathsf{POS}) + \mathsf{BAL}, \mathsf{RBAL}\rangle$ *is a valid variant for the algorithm.*

Moreover the variations of RBAL and BAL are correlated:

Lemma 8 *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, $\Delta\mathsf{RBAL} \leqslant \Delta\mathsf{BAL} + 3$ holds on each rule application. Let us study the variations of $\vartheta = 3 \times \{2.(\mathsf{NEG} + \mathsf{POS}) + \mathsf{BAL}\} + \{2.(\mathsf{NEG} + \mathsf{POS}) + \mathsf{RBAL}\}$ on a rule application:

- If the rule is not the right rotation ($\mathrm{RR}_=$), then the proof of Corollary 7 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 ($\mathrm{RR}_=$), then $\Delta\mathsf{NEG} = \Delta\mathsf{POS} = \Delta\mathsf{BAL} = 0$ and $\Delta\mathsf{RBAL} = -1$, thus $\Delta\vartheta < 0$.

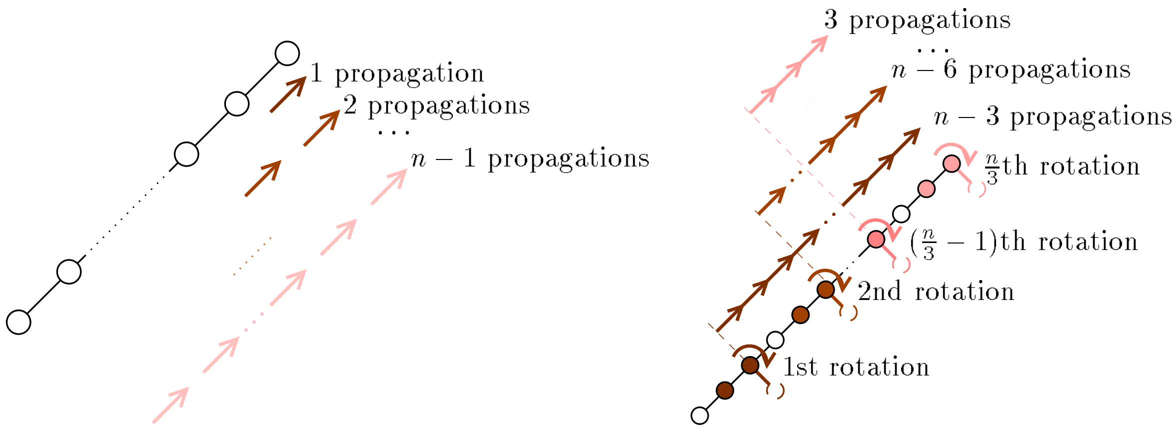
Thus ϑ is a valid variant. A careful analysis of RBAL shows that $\Delta\mathsf{RBAL} \leqslant \mathsf{BAL} + 2$ on any rule application. The same analysis finally leads to:

Theorem 9 (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 10 (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 from any initial shape.*

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



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

5.5 Examples for worst convergence times

We describe two executions of the algorithm with $\Omega(n^2)$ rule applications:

[Fig. 6(a)] shows a scheme of $\Omega(n^2)$ propagations in a linear tree where initially each node believes to be a leaf: each node, starting by the son of the root and ending by the leaf, informs in turn the root of its presence.

[Fig. 6(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 among the space of all binary trees.*

6 Experimental worst convergence time analysis

The goal of this section is to present some experimental results on the practical behavior of our rebalancing scheme. Observe that the rules have exclusive guards with respect to the node u according to the notations of Section 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 $n = 20$).

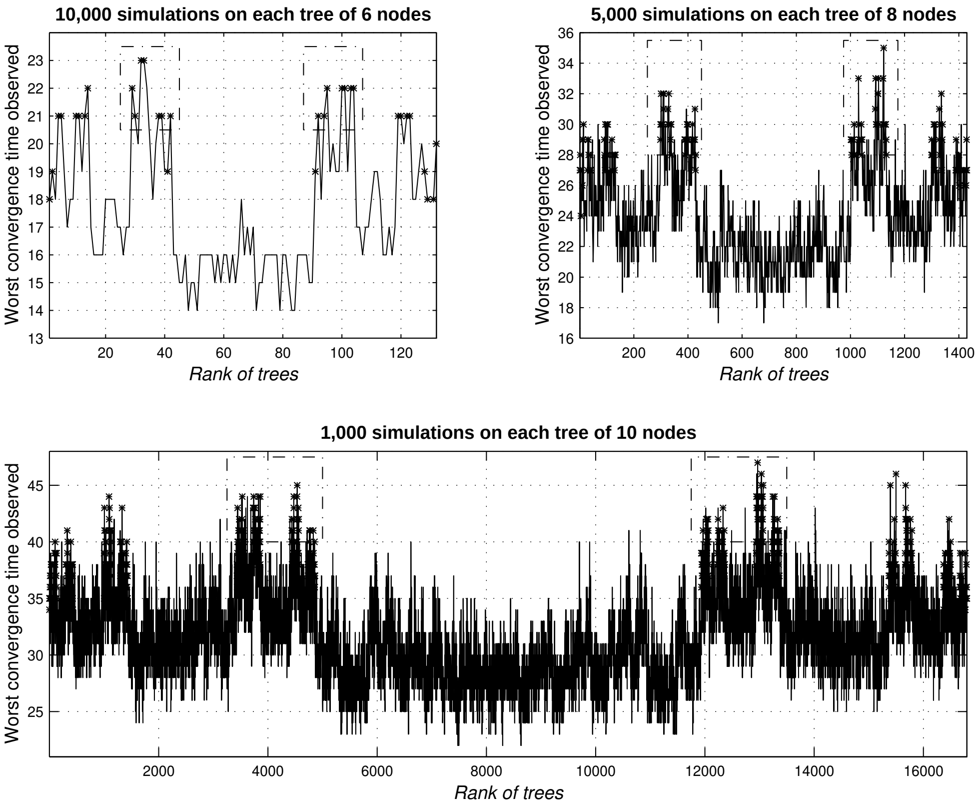


Figure 7: Worst convergence times observed on $10,000$ executions over each tree of size $n = 6$, on $5,000$ executions per tree for $n = 8$ and on $1,000$ executions per tree for $n = 10$. The '$*$' point out the linear trees. The dashed rectangles highlight the worst convergence time localizations.

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. 7]. 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 $n = 10$).

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 continuous 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. 7].

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. 7] 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. 8].

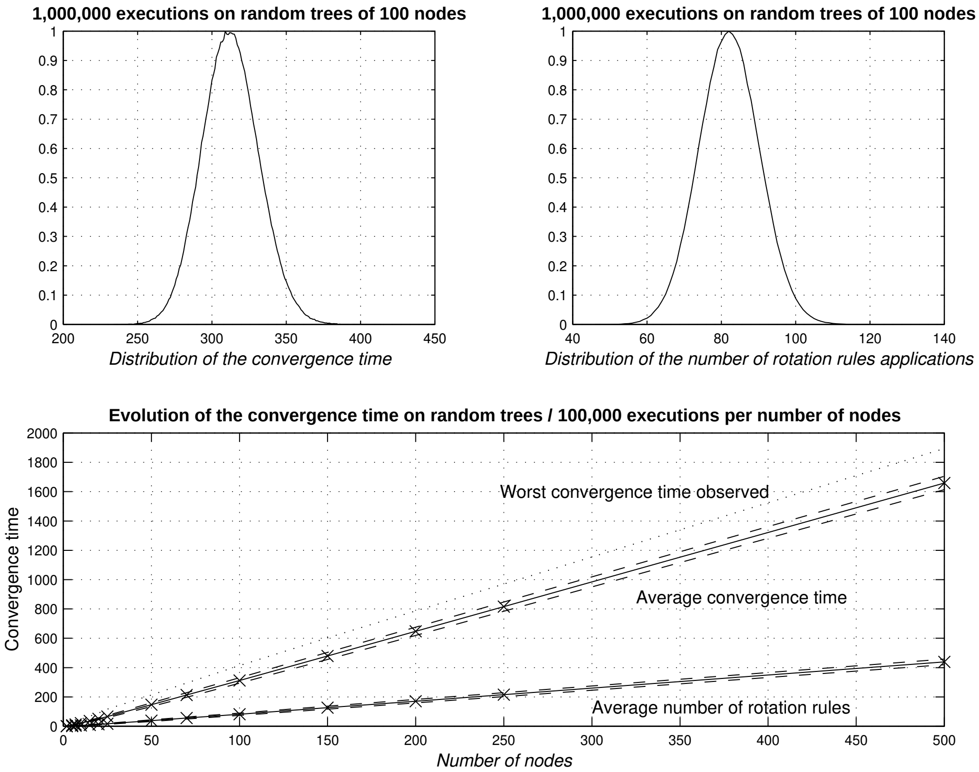


Figure 9: Average convergence time observed on $100,000$ executions over random trees of size $n = 2, 5, 7, 10, 15, 20, 25, 50, 70, 100, 150, 200, 250, 500$. The dotted lines represent the dispersion intervals.

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.

Again, intensive simulations on the regular zigzag trees with up to $5,000$ nodes yield *a worst convergence time of $\gamma.n$ rules applications, where $\gamma \cong 4$* [Fig. 9]. The quadratic executions exhibited in Section 5.5 are thus likely to be extremely singular.

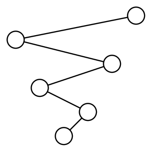


Figure 8: A regular zigzag of size $n = 6$.

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. 9].

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 an arbitrary binary tree after at most $O(n)$ rule applications.*

Unfortunately we do not have any theoretical estimation concerning the convergence time distribution. Note however that the analysis of the standard sequential AVL algorithm is still one of

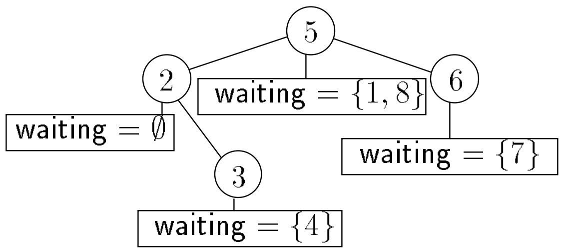


Figure 10: A strongly sorted tree with its waiting sets

the outstanding problems in the analysis of algorithms according to [SF96].

7 Concurrent insertion and deletion of keys

Refining this basic rebalancing framework, we can now design algorithms to manage the concurrent insertion and deletion of keys in AVL trees. We only sketch these algorithms, a complete development can be found in [BGM95].

7.1 Concurrent insertions

Our approach is to consider the action of inserting keys into the tree as a *percolation* of the keys along the tree, starting from the root down to the leaves. The keys percolate in accordance with the key ordering: at any moment, a key a percolating in the left (resp. right) subtree of a node v should be lower (resp. greater) than the key $\mathsf{key}(v)$ stored at v. An arbitrary number of keys may concurrently percolate along the tree, and new keys may be dropped at the root at any point. At any moment, each node u in the tree stores a number of keys waiting to percolate down in a bag $\mathsf{waiting}(u)$, as shown on [Fig. 10]. When a percolating key reaches a leaf node, then a new leaf node is created as a son of the former one. The rebalancing scheme will eventually reshape the tree into an AVL within a finite delay after the last key to be inserted has been stored.

Of course, this percolation process runs concurrently with the rebalancing scheme, and we have to guarantee that no interference may occur:

- Creating a new leaf node should respect the consistency condition of HRS-trees: a leaf node knows it is a leaf.
- Rotating a subtree should respect the consistency of the percolation process: a percolating key should not be moved away from its percolation path by a rotation.

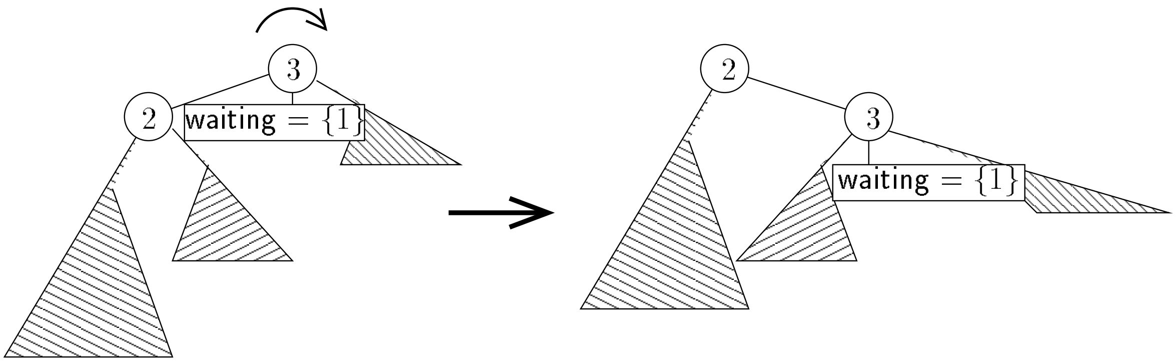
These two conditions are sufficient to ensure the correct concurrent behavior of the whole scheme.

We equip each node of the tree with a new register $\mathsf{waiting}(u)$ (the *waiting bag* at u) which holds the keys waiting at node u for downwards percolation. Operation $+$ adds a key to the set and operation $-$ removes it.

Definition 2 (Strongly sorted HRS-tree) A HRS-tree is *strongly sorted* if the following holds: If u is in the left (resp. right) subtree of v, and $a \in \mathsf{waiting}(u)$, then $a \leqslant \mathsf{key}(v)$ (resp. $a \geqslant \mathsf{key}(v)$) (see [Fig. 10]).

As a simple example of such a tree, consider a single node u and n keys $a_1, \ldots, a_n$ with

$$\mathsf{key}(u) = a_1 \qquad \mathsf{waiting}(u) = \{a_2, \ldots, a_n\} \qquad \mathsf{lefth}(u) = 0 \qquad \mathsf{righth}(u) = 0$$



Strongly sorted Not strongly sorted

Figure 11: Left-left rotations do not preserve strong sortedness

We present the *Insert Left Percolation* rule (ILP), moving a key a in $\mathsf{waiting}(u)$ with $a \leq \mathsf{key}(u)$ from u to $u{\rightarrow}\mathsf{ls}$. The Insert Right Percolation rule with $a > \mathsf{key}(u)$ is deduced symmetrically.

Rule (ILP) – Insert Left Percolation

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

Action: If u has a left son v, then

$$\mathsf{waiting}(u') = \mathsf{waiting}(u) - a \quad \mathsf{waiting}(v') = \mathsf{waiting}(v) + a$$

Otherwise, create a new node v, left son of u and:

$$\begin{array}{c} \mathsf{waiting}(u') = \mathsf{waiting}(u) - a \quad \mathsf{key}(v') = a \\ \mathsf{waiting}(v') = \emptyset \quad \mathsf{lefth}(v') = 0 \quad \mathsf{righth}(v') = 0 \end{array}$$

Spatial scope: Node u and the potential new node v.

Unfortunately, as shown on [Fig. 11], the rotation rules may invalidate the strong sortedness property. We have to refine them so as to take into account the waiting bags of the nodes involved in their spatial scopes. A first possibility is to gather all the keys waiting at the nodes in the spatial scope at the new root of the rotated subtree. An alternative choice is to allow a rotation to be applied only if no key is waiting at the nodes in its spatial scope. The choice between these two strategies depends on the relative priority one assigns to percolation with respect to rebalancing. For instance, we present the refinement of the (RR) rule according to the second alternative below. It is clear that is vacuously preserves strong sortedness.

Rule (IRR$_*$) – Insert Right Rotation, Unbalanced case

Guard: Node u is the left son of node v, node u is reliable, $\mathsf{bal}(u) > 0$ and $\mathsf{bal}(v) \geqslant 2$, $\mathsf{waiting}(u) = \mathsf{waiting}(v) = \emptyset$.

Action: Nodes u and v execute a right rotation [Fig. 3(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: Node u and its parent $v = u \rightarrow \mathsf{p}$.

The safety and the liveness of this refined scheme can be established without any difficulty.

Safety. If no rule applies, then the tree is an AVL whose all waiting bags are empty, and all keys dropped at the root have been stored into nodes.

Liveness. A variant can be designed along the idea developed in the previous sections. Let KEYS be the number of currently percolating keys:

$$\mathsf{KEYS} = \sum_u |\mathsf{waiting}(u)|$$

This quantity cannot increase after the last key to be inserted has been dropped at the root of the tree. It strictly decreases on creating a new leaf to store a key. It is left unchanged by all other rules, including reshaping rules.

Let quantity $In(u)$ be the number of nodes in the subtree rooted at node u (this is the complementary of quantity $Out(u)$). Let WAIT be the number of percolating keys weighed by their distance to the leaves, as measured by In.

$$\mathsf{WAIT} = \sum_u In(u) \cdot |\mathsf{waiting}(u)|$$

This quantity strictly decreases on moving a key downwards. It is left unchanged by all reshaping rules, at least if we restrict rotations so that the nodes in their spatial scope have empty waiting bags (second alternative). Therefore, the integer quantity $\langle \mathsf{KEYS}, \mathsf{WAIT}, \mathsf{NEG}, 2.\mathsf{POS} + \mathsf{BAL}, \mathsf{RBAL} \rangle$ is a valid variant.

Note that there is no need here to assume anything about the unicity of the inserted keys.

7.2 Concurrent insertion and deletion

We now address the deletion of keys, in concurrence with the insertion of keys and the rebalancing of the tree. The deletion of keys can be handled much in the same way as the insertion. To delete some key a, one drops a *negative* key $\bar{a}$ at the root. It percolates downwards along the tree as for an insertion until it reaches key a: at this point, both keys annihilate.

Remark *This crucially relies on the fact that key a and key $\bar{a}$ follow the same insertion path along the tree. Therefore, we must restrict ourselves to the case where all keys are unique. (Multiple keys with identical values could be handled by a suitable labeling.)*

A negative key $\bar{a}$ may vanish in two exclusive ways.

- Either it percolates down to a waiting bag $\mathsf{waiting}(u)$ where key a already sits. In this case, both keys vanish together:

$$\mathsf{waiting}(u') = \mathsf{waiting}(u) - \{a, \bar{a}\}$$

- Or it may reach the waiting bag $\mathsf{waiting}(u)$ of a node u storing key a: $\mathsf{key}(u) = a$. Then, key $\bar{a}$ vanishes, but we cannot "reset" node u in general: due to the structure of a search tree, a node may be removed only if it is a leaf! We simply mark node u as *dead*, a leave it as such. It will eventually be removed once it has been pushed down to the leaves, see below.

- There is no other possibility: if a negative key $\bar{a}$ reaches the waiting bag of a tree, then it means that no insertion of key a had occur beforehand.

The last problem remaining to be solved is the deletion of dead nodes. If a dead node is a leaf, then it can be safely removed. Otherwise, it must be pushed down to the leaves by a series of single rotations (Right and Left Rotation rules). The problem is that such rotations may temporarily increase the local unbalance of the tree, and thus compete with the basic rebalancing scheme: a dead node could be pushed down by such a rotation, and immediately pushed up by the original scheme. Therefore, we must refine the rules so as to prevent any interference between them. A straightforward fix is the following (we do not present the symmetric rules, as they as obvious).

- A copy of the single rotation rule is added to take care of the dead nodes specifically. It can be used only is the root of the rotated subtree is a dead node with an alive son, disregarding their respective balances. Applying this rule ensures that one dead node gets closer to the leaves of the tree.

- The original rules are refined so that they apply only if no node in their spatial scope is dead.

Rule (DRR) – Delete Right Rotation

Guard: Node u is the left son of node v, node v is dead and node u is alive, and $\mathsf{waiting}(u) = \mathsf{waiting}(v) = \emptyset$.

Action: Nodes u and v execute a right rotation [Fig. 3(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: Node u and its parent $v = u \rightarrow \mathsf{p}$.

Once a dead node has been pushed down to the leaves, it can be removed if its waiting bags are empty. The registers of the father have to be adjusted so as to maintain the consistency condition of the HRS-trees.

Rule (DLDL) – Delete Left Dead Leaf

Guard: Node u is the left son of node v, u is a leaf, u is dead, and $\mathsf{waiting}(u) = \emptyset$.

Action: Node u is deleted, and the left register of v is adjusted.

$$\mathsf{lefth}(v') = 0$$

Note that: There is no restriction on v: it may be also dead.

Spatial scope: Node u and its parent $v = u \rightarrow \mathsf{p}$.

It can be shown that if no rule applies, then the tree is an AVL whose all waiting bags are empty and whose all nodes are alive. The crucial observation is that keys may always percolate down, and that a lowest dead node (there is no other dead node in the subtrees rooted at it) with empty waiting bags may always be rotated down to the leaves.

We should stress that these algorithms offer many opportunity of variation. For instance, a dead left son may be removed as soon as it has no right son. Yet, the overall impact of all these variants on the efficiency is far from being clear.

8 Emulation of other AVL based algorithms

The approach given here is "fine-grained" with a high degree of concurrency and atomicity. We can use it to *emulate* other existing "medium-grained" algorithms. 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 bounded number of rules of B. We keep the discussion informal, but it could be formally rewritten with the notion of homomorphisms between models of parallel computation systems introduced by T. Kasai and R. Miller [KM82]. We will only take into account the rebalancing phase (when all the keys have been transformed into new leaves).

8.1 Sequential algorithm

In this algorithm, every node u holds a register $\mathsf{bf}(u) = \mathsf{realh}(u \rightarrow \mathit{ls}) - \mathsf{realh}(u \rightarrow \mathit{rs})$ with values in $\{-1, 0, 1\}$. Essentially, the sequential insertion phase reconstructs the tree bottom up in order to maintain the 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 rules bottom up updating the value of $\mathsf{bf}(u)$ 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 [Ell80]). On reaching the *critical node* (the last node having a non-zero balance), a single rotation is possibly applied if ever necessary. The distributed algorithm mimics a bottom-up version of the sequential one.

8.2 Another definition of relaxed height in HRS-trees

Let us consider the emulation of some distributed algorithms based on local rules [Kes83, NSSW87, NSSW92, Lar94]. They are based on various approximate notions of height. For the sake of this presentation, we introduce the following alternate definition.

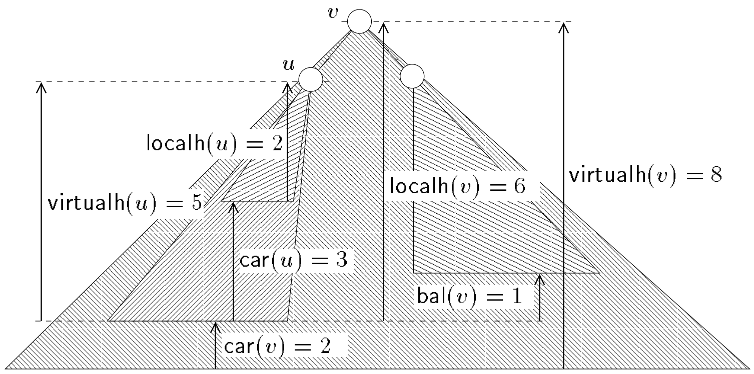


Figure 12: The **virtualh** and their relations with the **localh** (case with positive carries).

Definition 3 (Virtual height) The virtual height $\mathsf{virtualh}(u)$ of a node u in an HRS-tree is defined by

$$\mathsf{virtualh}(u) = \mathsf{localh}(u) + \mathsf{car}(u)$$

If u is a left son we have $\mathsf{car}(u) = \mathsf{lefth}(u{\to}\textsf{\textit{p}}) - \mathsf{localh}(u)$. Thus, $\mathsf{virtualh}(u) = \mathsf{lefth}(u{\to}\textsf{\textit{p}})$. In general

$$\mathsf{virtualh}(u{\to}\textsf{\textit{ls}}) = \mathsf{lefth}(u) \qquad \mathsf{virtualh}(u{\to}\textsf{\textit{rs}}) = \mathsf{righth}(u)$$

therefore

$$\mathsf{localh}(u) = 1 + \max\left(\mathsf{virtualh}(u{\to}\textsf{\textit{ls}}), \mathsf{virtualh}(u{\to}\textsf{\textit{rs}})\right)$$

and

$$\mathsf{virtualh}(u) = 1 + \mathsf{car}(u) + \max\left(\mathsf{virtualh}(u{\to}\textsf{\textit{ls}}), \mathsf{virtualh}(u{\to}\textsf{\textit{rs}})\right)$$

The virtual height approximately behaves as a height because it verifies the usual definition corrected with a $\mathsf{car}(u)$. Therefore the **virtualh** can be redefined as:

$$\begin{cases} \mathsf{virtualh}(\mathsf{nil}) = 0 \\ \mathsf{virtualh}(u \neq \mathsf{nil}) = 1 + \mathsf{car}(u) + \max\left(\mathsf{virtualh}(u{\to}\textsf{\textit{ls}}), \mathsf{virtualh}(u{\to}\textsf{\textit{rs}})\right) \end{cases}$$

As $\mathsf{car}(\mathsf{root}) = 0$,the **virtualh** of the **root** verifies:

$$\mathsf{virtualh}(\mathsf{root}) = 1 + \max\left(\mathsf{virtualh}(\mathsf{root}{\to}\textsf{\textit{ls}}), \mathsf{virtualh}(\mathsf{root}{\to}\textsf{\textit{rs}})\right)$$

Note that $\mathsf{car}(u) \in \{\ldots, -2, -1, 0, 1, 2, \ldots\}$. Moreover the *balance* can be rewritten as:

$$\mathsf{bal}(u) = \mathsf{virtualh}(u{\to}\textsf{\textit{ls}}) - \mathsf{virtualh}(u{\to}\textsf{\textit{rs}})$$

On [FIG. 12], a graphical interpretation of the **virtualh** and **localh** is given. We have detailed two nodes u and v such that $v{\to}\textsf{\textit{ls}} = u$. In the depicted case, both nodes have positive carries. The other cases, for instance $\mathsf{car}(u) > 0$ and $\mathsf{car}(v) < 0$ also admit nice graphical interpretations.

8.3 Kessels' approach

We now consider emulating the algorithm proposed by J.L.W. Kessels in [Kes83]. We start with a brief description. Each node holds two private registers: its *dynamic balance* dbal and its *dynamic carry* dcarry. The *dynamic height* quantity is defined as:

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

Note that $\mathsf{dheight}$ is not a private register. The dynamic balance is defined as

$$\mathsf{dbal}(u) = \mathsf{dheight}(u \rightarrow ls) - \mathsf{dheight}(u \rightarrow rs)$$

We borrow the following definition from Kessels:

Definition 4 A *Carry-Relaxed AVL Tree* (CarryRelaxedAVL-tree) is a binary search tree whose nodes are equipped with two private registers dbal and dcarry ssatisfying $\mathsf{dbal}(u) \in \{-1, 0, +1\}$ and $\mathsf{dcarry}(u) \in \{0, 1\}$.

To be precise, Kessels adds to the preceding definition the condition: any node u such that $\mathsf{dheight}(u) \neq 0$ and $\mathsf{dcarry}(u) = 1$ satisfies $\mathsf{dbal}(u) \neq 0$. We relax this condition.

The quantities $\mathsf{virtualh}$ and $\mathsf{dheight}$ obey similar equations:

$$\begin{aligned} \mathsf{virtualh}(u) &= 1 + \mathsf{car}(u) + \max(\mathsf{virtualh}(u \rightarrow ls), \mathsf{virtualh}(u \rightarrow rs)) \\ \mathsf{dheight}(u) &= 1 - \mathsf{dcarry}(n) + \max(\mathsf{dheight}(u \rightarrow ls), \mathsf{dheight}(u \rightarrow rs)) \end{aligned}$$

This suggests us the following identifications between CarryRelaxedAVL-trees and the HRS-trees:

$$\mathsf{dheight}(u) \equiv \mathsf{virtualh}(u) \qquad \mathsf{dcarry}(u) \equiv -\mathsf{car}(u)$$

As a consequence, we can also identify

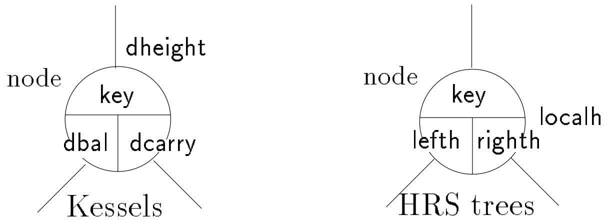
$$\mathsf{dbal}(u) \equiv \mathsf{bal}(u) \qquad \begin{cases} \mathsf{lefth}(u) & \equiv \mathsf{dheight}(u \rightarrow ls) \\ \mathsf{righth}(u) & \equiv \mathsf{dheight}(u \rightarrow rs) \end{cases}$$

Recall that a node u of a HRS-tree holds the registers $\mathsf{lefth}(u)$ and $\mathsf{righth}(u)$. In a CarryRelaxedAVL-tree, node u holds the registers $\mathsf{dbal}(u)$ and $\mathsf{dcarry}(u)$.

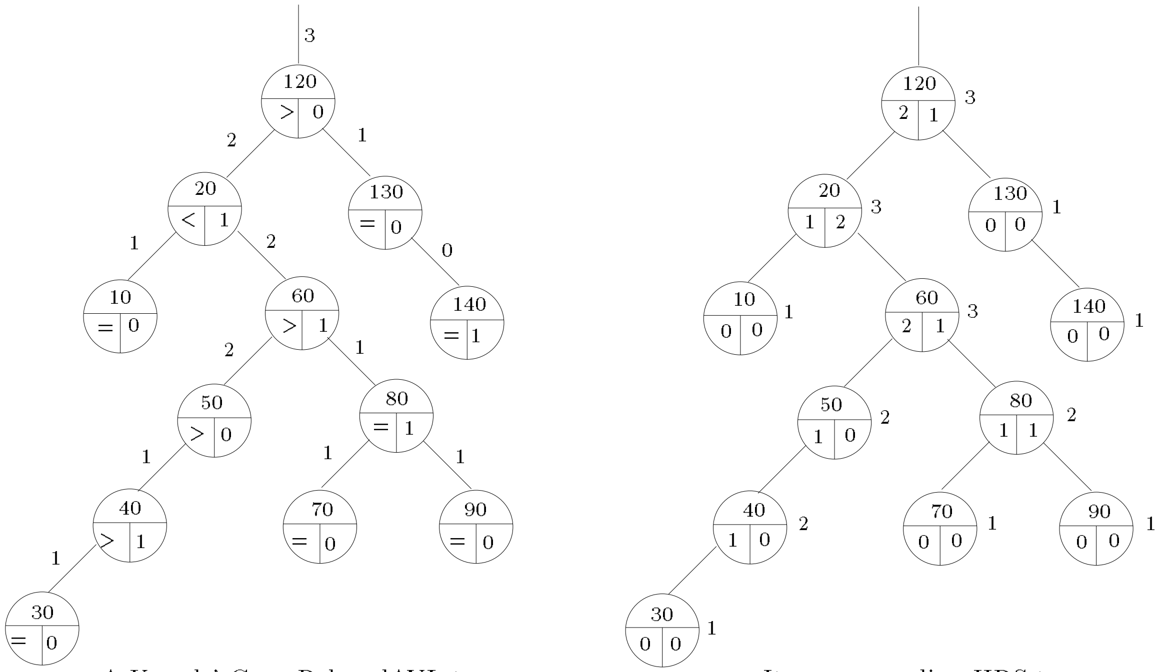
Lemma 11 (Bijection property) *There is a bijection between HRS-trees whose all nodes u satisfy $\mathsf{car}(u) = \{0, -1\}$ and $\mathsf{bal}(u) \in \{-1, 0, +1\}$ and CarryRelaxedAVL-trees.*

Proof. The shape of the trees is kept constant through the bijection. Only the contents of the local registers is changed. Given a node u in a CarryRelaxedAVL-tree, the corresponding node in the HRS-tree is computed using $\mathsf{lefth}(u) = \mathsf{dheight}(u \rightarrow ls)$ and $\mathsf{righth}(u) = \mathsf{dheight}(u \rightarrow rs)$. Given a node u in a HRS-tree, it can be transformed into an CarryRelaxedAVL node using $\mathsf{dbal}(u) = \mathsf{bal}(u)$ and $\mathsf{dcarry}(u) = \mathsf{car}(u)$. □

On [Fig. 13] we give an example of the relationships between the two models.



(a) Local registers.



(b) A concrete example of the bijection.

Figure 13: Relations between CarryRelaxedAVL-trees and HRS-trees.

Daemons in CarryRelaxedAVL-trees The following three local rules have been proposed by Kessels [Kes83] in order to maintain the consistency condition of CarryRelaxedAVL-trees trough reshaping: $\mathsf{dbal}(n) \in \{-1, 0, +1\}$ and $\mathsf{dcarry}(u) \in \{0, 1\}$.

Rule (Transformation A) – Left Carry Propagation

Guard: Node u is the left son of node v, $\mathsf{dcarry}(u) = 1$, $\mathsf{dcarry}(v) = 0$ and $\mathsf{dbal}(v) = \{-1, 0\}$.

Action: First, set $\mathsf{dcarry}(u') = 0$ and then:

> **Case $\mathsf{dcarry}(v) = -1$:** Set $\mathsf{dbal}(v') = 0$ and $\mathsf{dbal}(v') = 0$.
>
> **Case $\mathsf{dbal}(v) = 0$:** Set $\mathsf{dcarry}(v') = 1$ and $\mathsf{dbal}(v') = 1$.

Spatial scope: Nodes v and $u = v \rightarrow \mathit{ls}$.

Rule (Transformation B) – Right Rotation with Carry

Guard: Node u is a left son of a node v and $\mathsf{dbal}(u) = 1$, $\mathsf{dcarry}(u) = 1$, $\mathsf{dbal}(v) = 1$, $\mathsf{dcarry}(v) = 0$.

Action: A single right rotation takes place and the new values of dynamic balances and carries are $\mathsf{dbal}(u') = \mathsf{dbal}(v') = 0$ and $\mathsf{dcarry}(u') = \mathsf{dcarry}(v') = 0$.

Spatial scope: Nodes u and v.

Note that: Symmetrically if u is the right son of v.

The following rule deals with double rotations. The version given here is a slightly modified version (cases 4 and 5) from the one given in [Kes83]. With this version, the proof of liveness is easier and moreover the will get our emulation result.

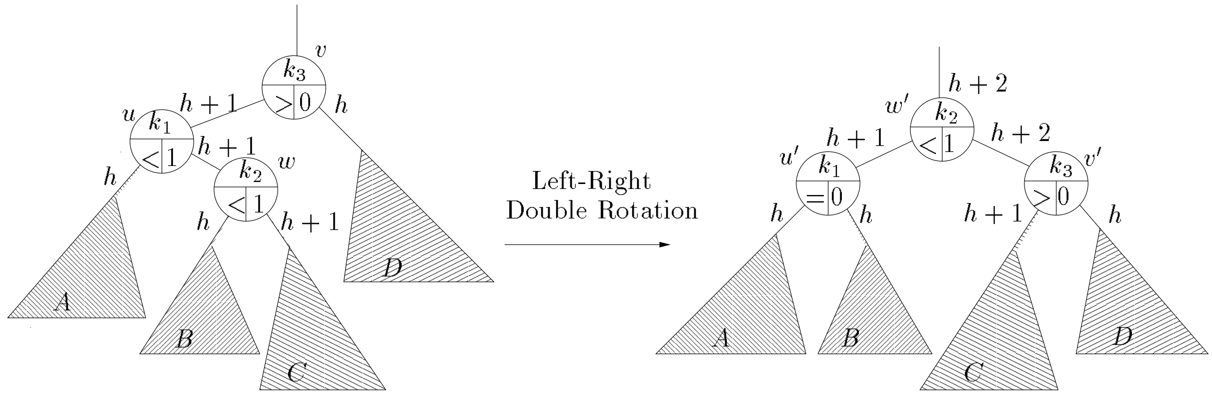


Figure 14: Transformation C, Case $\mathsf{dcarry}(w) = 1$ and $\mathsf{dbal}(w) = -1$.

Rule (Transformation C) – Double Left Right Rotation with Carries

Guard: A subtree as on [FIG. 14], left part, with $\mathsf{dcarry}(v) = 0$, $\mathsf{dbal}(v) = +1$ and $\mathsf{dcarry}(u) = 1$, $\mathsf{dbal}(u) = -1$.

Action: Restructure the tree into the subtree as on [FIG. 14], right part,with the usual updating for keys and left and right pointers. The dynamic balances and carries are updated as follows:

Case $\mathsf{dcarry}(w) = 0$ and $\mathsf{dbal}(w) = -1$: Carries are updated as $\mathsf{dcarry}(u') = \mathsf{dcarry}(v') = \mathsf{dcarry}(w') = 0$. New balances are $\mathsf{dbal}(u') = 1$ and $\mathsf{dbal}(w') = \mathsf{dbal}(v') = 0$.

Case $\mathsf{dcarry}(w) = 0$ and $\mathsf{dbal}(w) = 0$: Carries are updated as $\mathsf{dcarry}(u') = \mathsf{dcarry}(v') = \mathsf{dcarry}(w') = 0$. New balances are $\mathsf{dbal}(u') = \mathsf{dbal}(v') = \mathsf{dbal}(w') = 0$.

Case $\mathsf{dcarry}(w) = 0$ and $\mathsf{dbal}(w) = 1$: Carries are updated as $\mathsf{dcarry}(u') = \mathsf{dcarry}(v') = \mathsf{dcarry}(w') = 0$. New balances are $\mathsf{dbal}(u') = \mathsf{dbal}(w') = 0$ and $\mathsf{dbal}(v') = -1$.

Case $\mathsf{dcarry}(w) = 1$ and $\mathsf{dbal}(w) = -1$: Carries are updated as $\mathsf{dcarry}(u') = \mathsf{dcarry}(v') = 0$ and $\mathsf{dcarry}(w') = 1$. New balances are $\mathsf{dbal}(u') = 0$, $\mathsf{dbal}(w') = -1$ and $\mathsf{dbal}(v') = 1$.

Case $\mathsf{dcarry}(w) = 1$ and $\mathsf{dbal}(w) = 1$: Carries are updated as $\mathsf{dcarry}(u') = \mathsf{dcarry}(v') = 0$ and $\mathsf{dcarry}(w') = 1$. New balances are $\mathsf{dbal}(u') = -1$, $\mathsf{dbal}(w') = 1$ and $\mathsf{dbal}(v') = 0$.

Spatial scope: Nodes u, v, w.

Emulation of Kessels' algorithm

Lemma 12 (Emulation property) *Any rule of the CarryRelaxedAVL rebalancing scheme can be emulated by applying at most three rules of the HRS rebalancing scheme.*

Proof. Thanks to Lemma 11, the proof just consists on chaining propagations and rotations in the HRS model. We consider the three cases. The *Transformation A* can be emulated by the left propagation rule of the HRS-trees. The *Transformation B* can be emulated chaining a propagation with a single rotation. The *Transformation C* can be emulated coupling two propagation rules with a double left-right rotation. □

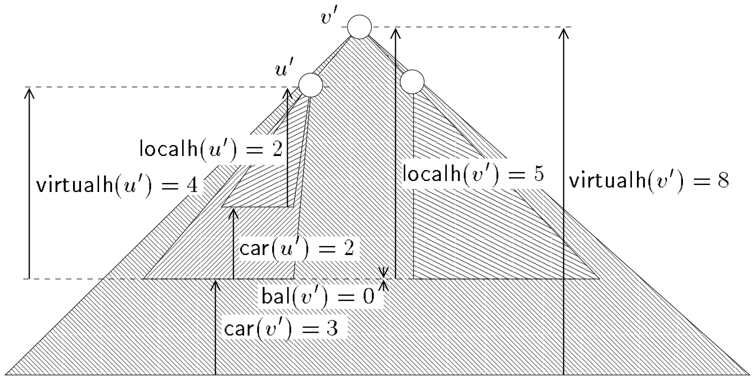


Figure 15: Example of application of Rule (OneLP). The initial configuration corresponds to [Fig. 12], this figure corresponds to the final configuration. In the initial configuration, nodes u and v have positive carries and $\mathsf{lefth}(v) = \mathsf{virtualh}(u) = 5$. Note that $\mathsf{lefth}(v') = \mathsf{lefth}(v) - 1$.

8.4 Nurmi et al.'s approach

We consider an extension of the Kessels' algorithm developed by O. Nurmi, E. Soisalon-Soininen and D. Wood in [NSSW92] (see also [NSSW87]). Each node holds two private registers, the *relaxed balance* rbal and the *tag* tag. The tag quantity denote the information on height yet to be propagated upwards by the node. A *relaxed height* for such trees is defined as:

$$\begin{cases} \mathsf{rheight}(\mathsf{nil}) = 0 \\ \mathsf{rheight}(u) = 1 + \mathsf{tag}(u) + \max(\mathsf{rheight}(u \to \mathit{ls}), \mathsf{rheight}(u \to \mathit{rs})) \end{cases}$$

and $\mathsf{rbal}(u) = \mathsf{rheight}(u \to \mathit{ls}) - \mathsf{rheight}(u \to \mathit{rs})$. As in Kessels' approach, they introduce a a relaxed version of AVL trees.

Definition 5 A *tag-relaxed AVL* (TagRelaxedAVL-tree) is a binary search tree whose nodes are equipped with two private registers tag and rbal satisfying the following consistency condition: $\mathsf{rbal}(u) \in \{-1, 0, +1\}$ and $\mathsf{tag}(u) \in \{-1, 0, 1, 2, \ldots\}$.

As before, we consider only the rebalancing process. The transformations consist in two phases. Phase 1 is designed to decrease the tag value of some node u. At the end of this phase, the value of $\mathsf{rbal}(u \to p)$ lies between -2 and 2. Phase 2 brings back the value of $\mathsf{rbal}(u \to p)$ between -1 and 1 by readjusting the tags and making a rotation if necessary. The algorithm eventually yields a tree such that $\mathsf{tag}(u) = 0$. As the local transformations has to maintain the relaxed balance, *only one-unit variations on the tag values can be correctly absorbed.* This incremental variation of the tag values induces an additional level of complexity in the description of the transformations.

Let us consider emulating this algorithm. The definitions of rheight and virtualh suggest us the identifications:

$$\mathsf{rheight}(u) \equiv \mathsf{virtualh}(u) \qquad \mathsf{rbal}(u) \equiv \mathsf{bal}(u) \qquad \mathsf{tag}(u) \equiv \mathsf{car}(u)$$

Unfortunately, this algorithm cannot be directly emulated in our HRS scheme. It specifies that tag values vary by at most one unit on each rule application, whereas our HRS propagation rules directly set carries to zero in one step. The HRS rules have thus to be recast to an incremental version. For instance, the new rule for one unit left propagation is as follows

Rule (OneLP) – One Unit Left Propagation

Guard: Node u is a left son of v and $\mathsf{car}(u) \neq 0$.

Action: The apparent left height of v is updated:

$$\mathsf{lefth}(v') = \begin{cases} \mathsf{lefth}(v) - 1 & \text{if } \mathsf{car}(u) > 0 \\ \mathsf{lefth}(v) + 1 & \text{if } \mathsf{car}(u) < 0 \end{cases}$$

Spatial scope: Nodes u and v.

On applying Rule (OneLP), the quantities are modified by at most one unit. The [FIG. 15] displays an example with positive carries ($\mathsf{car}(u) > 0$ and $\mathsf{car}(v) > 0$). In general we have the following straightforward lemma:

Lemma 13 *In one application on Rule (OneLP) nodes u' and v' satisfy:*

$$\mathsf{car}(u') = \begin{cases} \mathsf{car}(u) - 1 & \textit{if } \mathsf{car}(u) > 0 \\ \mathsf{car}(u) + 1 & \textit{if } \mathsf{car}(u) < 0 \end{cases}$$

$$\mathsf{virtualh}(u') = \begin{cases} \mathsf{virtualh}(u) - 1 & \textit{if } \mathsf{car}(u) > 0 \\ \mathsf{virtualh}(u) + 1 & \textit{if } \mathsf{car}(u) < 0 \end{cases}$$

$$\mathsf{car}(v') = \begin{cases} \mathsf{car}(v) + (\mathsf{bal}(v) > 0)?1:0 & \textit{if } \mathsf{car}(u) > 0 \\ \mathsf{car}(v) - (\mathsf{bal}(v) \geq 0)?1:0 & \textit{if } \mathsf{car}(u) < 0 \end{cases}$$

$$\mathsf{virtualh}(v') = \mathsf{virtualh}(v)$$

The $\mathsf{bal}(u)$ *quantity evolves as:*

$$\mathsf{bal}(v') = \begin{cases} \mathsf{bal}(v) - 1 & \textit{if } \mathsf{car}(u) > 0 \\ \mathsf{bal}(v) + 1 & \textit{if } \mathsf{car}(u) < 0 \end{cases}$$

We can get (with patience!) new versions for rules dealing with rotations and double rotations with one unit carry variations. Let call (UnitRules) this set of modified rules.

Lemma 14 (Emulation property) *Using the (UnitRules) version of the HRS scheme, we can mimic the restructuring transformations in TagRelaxedAVL-trees coupling unit propagations and unit rotations.*

8.5 Larsen's approach.

Finally, let us consider the approach proposed by K.L. Larsen [Lar94]. Recall that, given a node u with father $u{\rightarrow}\mathsf{p}$, the transformations given by Nurmi et al. in [NSSW92] require as a precondition $\mathsf{tag}(u) \neq 0$ and $\mathsf{tag}(u{\rightarrow}\mathsf{p}) = 0$. This precondition has been relaxed by Larsen so as to accept non-zero values for $\mathsf{tag}(u{\rightarrow}\mathsf{p})$. to avoid the accumulation of negative values in $\mathsf{tag}(u{\rightarrow}\mathsf{p})$, Larsen modifies the transformations above by coupling more tightly the propagations and the rotations. We can emulate this approach along the same lines as above.

9 Conclusion

This paper presents a fine-grained, distributed approach to the problem of managing concurrent requests 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 one 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 (the HRS consistency requirement). 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 skills.

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 remaining of the registers). We show 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.

This approach has been used in [GMR97] to extend *Chromatic trees* into *Hyper-Red-Black trees* having close daemons to increase the degree of concurrency.

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 trees. Research Report RR95-45, LIP, ENS Lyon, 1995. Appeared with the same title as: INRIA, Rapport de Recherche 2761. Available at URL http://www.ens-lyon.fr/LIP.

[BGMS97] L. Bougé, J. Gabarró, X. Messeguer, and N. Schabanel. Concurrent rebalancing of AVL trees: a fine-grained approach. In *European Conf. Parallel Proc (Euro-Par '97)*, volume 1300 of *Lect. Notes Comp. Science*, pages 421–429, 1997.

[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.

[GM96] J. Gabarró and X. Messeguer. Massively parallel and distributed dictionaries on AVL and Brother Trees. In Yetongnon K. and S. Hariri, editors, *Parallel and Distributed Computing Systems*, pages 14–17. ISCA, 1996. Look also: Parallel dictionaries with local rules on AVL and Brother trees, Report LSI-96-31-R, 1997.

[GM97] J. Gabarró and X. Messeguer. A unified approach to concurrent and parallel algorithms on balanced data structures. In IEEE, editor, *Proc. of XVII Int. Conf. Chilean Computer Society*, 1997.

[GMR97] J. Gabarró, X. Messeguer, and D Riu. Concurrent rebalancing on HyperRed-Black trees. In IEEE, editor, *Proc. of XVII Int. Conf. Chilean Computer Society*, 1997.

[GS78] L. J. Guibas and R. Sedwick. A dichromatic framework for balanced trees. In *Proc. Ann. Symp. Foundations of Computer Science*, volume 19, pages 8–21. IEEE Comp. Soc., 1978.

[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.

[KM82] T. Kasai and R. Miller. Homomorphisms between models of parallel computation. *J. Comp. Sys. Sci.*, 25:285–331, 1982.

[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.

[NSS96a] O. Nurmi and E. Soisalon-Soininen. Chromatic binary search trees. *Acta informatica*, 33(6):547–557, 1996.

[NSS96b] O. Nurmi and E. Soisalon-Soininen. Chromatic binary search trees, a structure for concurrent rebalancing. *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.

[OSW79] T. Ottmann, H. Six, and D. Wood. On the correspondence between AVL trees and brother trees. *Computing*, 23(1):43–54, 1979.

[OW79] T. Ottmann and D. Wood. 1-2 brother trees or AVL trees revisited. *The Computer Journal*, 23(3):248–255, 1979.

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

[SSW97] E. Soisalon-Soininen and P. Widmayer. Relaxed balancing in search trees. In D.Z. Du and K.I. Ko, editors, *Advances in Algorithmics, languages and Complexity: Essays in Honor of Ronad V. Book*, pages 267–283. Kluwer Academic, 1997.

A Appendix: Exhaustive Case Analysis

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	$\mathsf{car}_u < 0$
			$0 \leqslant \mathsf{bal}_v$	$\mathsf{car}_u \cdot Out_u < 0$	$\mathsf{car}_u < 0$	$-\mathsf{car}_u > 0$
			$\mathsf{car}_u < \mathsf{bal}_v < 0$	$\mathsf{car}_u \cdot Out_u < 0$	$\mathsf{car}_u - \mathsf{bal}_v < 0$	$-\mathsf{car}_u + 2\mathsf{bal}_v \lesseqgtr 0$
		$\mathsf{car}_{v'} < 0$	$0 \leqslant \mathsf{bal}_v$	$\mathsf{car}_u \cdot Out_u - \mathsf{car}_{v'} \cdot Out_v < 0$	$-\mathsf{car}_v < 0$	$-\mathsf{car}_u > 0$
			$\mathsf{car}_u < \mathsf{bal}_v < 0$	$\mathsf{car}_u \cdot Out_u - \mathsf{car}_{v'} \cdot Out_v < 0$	$-\mathsf{car}_v < 0$	$-\mathsf{car}_u + 2\mathsf{bal}_v \lesseqgtr 0$

Table 1: Exhaustive case analysis for the rule (LP) according to the notation of Section 3.

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 (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 3