Concurrent Rebalancing on HyperRed-Black Trees *

Joaquim Gabarró Xavier Messeguer Daniel Riu
Universitat Politècnica de Catalunya
Departament de Llenguatges i Sistemes Informàtics
Campus Nord–Mòdul C6
C/ Jordi Girona Salgado, 1–3
08034 Barcelona, Spain
Contact author: messeguer@lsi.upc.es

Abstract

The HyperRed-Black *trees are a relaxed version of Red-Black trees accepting high degree of concurrency. In the Red-Black trees consecutive red nodes are forbidden. This restriction has been withdrawn in the* Chromatic *trees. They have been introduced by O. Nurmi and E. Soisalon-Soininen to work in a concurrent environment. A* Chromatic *tree can have big clusters of red nodes surrounded by black nodes. Nevertheless, concurrent rebalancing of* Chromatic *trees into* Red-Black *trees has a serious drawback: in big cluster of red nodes only the top node can be updated. Direct updating inside the cluster is forbidden. This approach gives us limited degree of concurrency. The* HyperRed-Black *trees has been designed to solve this problem. It is possible to update red nodes in the inside of a red cluster. In a* HyperRed-Black *tree nodes can have a multiplicity of colors; they can be red, black or hyper-red.*

1 Introduction

Red-Black trees have been recognized as an important data structure [2]. They are highly balanced search trees. Each node n stores a key, denoted **key**(n), and each internal node n stores three pointers **left**(n), **right**(n) and **parent**(n) pointing respectively to its sons and parent. The trees satisfy the following red-black properties:

P_1 : Every node is either red or black.

P_2 : Every leaf (NIL) is black.

P_3 : If a node is red then both its children are black. This is equivalent to, no path from the root to a leaf contains two consecutive red nodes.

P_4 : Every simple path from a node to a leaf contains the same number of black nodes.

The last condition allow us to define the function *blackness* (called black-height in [2]):

blackness(n) = the number of black nodes on any path from, but not including, a node n to a leaf.

As the **blackness** is computed by counting the black nodes, therefore it makes sense to encode the color as a flag **color**$(n) \in \{1, 0\}$ such that:

1. A node n is *black* iff **color**$(n) = 1$.
2. A node n is *red* iff **color**$(n) = 0$.

Then we can write **blackness**$(n) = \sum$ **color**(u), being the sum on any path from, but not including, a node n to a leaf. And the fourth property P_4 can be rewritten:

P_4 : Every simple path from a node to a leaf has the same sum of colors. Therefore, for any node n the **blackness**(n) is well defined.

In the following sections, we deal with the concurrent rebalancing problem for Red-Black trees. First, let us consider the rebalancing problem in a sequential environment. The *sequential insertion algorithm* has two parts.

*This work has been partially supported by ESPRIT LTR Project no. 20244 — ALCOM-IT and DGICYT under grant PB95-0787 (project KOALA) and ACI with Universidad de Chile DOG 2320–30.1.1997.

1. In a *percolation* part, the key to be inserted falls until it is attached to a new red node n at the bottom of the tree. As n is red, the property P_4 is maintained.

2. If the *parent* of n is black all the red-black properties are maintained and the insertion is over. Otherwise, n and **parent**(n) are red and (P_3) is false. In this case a bottom-up *reconstruction* part starts. The redness of consecutive nodes rises up (see later the *Red Propagation* and *Red Rotation* rules). Finally, if the root becomes red it is colore black (see later the *Blacking the Root* rule).

Let us describe the preceding rules. Always, n and **parent**(n) are red but (if it exists) the grandparent of n is black. If **uncle**(n) is also red, it is possible to move up the redness "one step up" changing the color of the uncle. The "shape" of the tree is kept constant. Formally, we have the following rule:

Rule : Red Propagation

Guard: Both, **parent**(n) and **uncle**(n) are red (recall that *grandparent* of n is black).

Behavior: The **parent**(n) and **uncle**(n) become black and the *grandparent* becomes red.

Spatial Scope: Nodes **parent**(n) , **uncle**(n) and the grandparent of n.

If the *uncle* is black the preceding rule does not apply. In this case, a rotation or a double rotation allow us to move down the blackness of the *uncle*. The shape of the tree changes. This rule does not allow us propagate redness up.

Rule : Red Rotation

Guard: The **parent**(n) is red but **uncle**(n) is black (as before, the *grandparent* of n is black).

Behavior: A rotation (single o double) takes place around the *grandparent* of n.

Spatial Scope: Nodes involved in the rotation.

Finally, if the pair n and **parent**(n) of red nodes reaches the root of the tree (we mean **parent**$(n) = root$), the color of the root must be changed. Formally:

Rule : Blacking Root

Guard: The root is red.

Behavior: Color the root black.

Spatial Scope: The root.

The *on-the-fly* concurrent approach, designs algorithms as a set of local evolution rules. The control is kept as nondeterministic as possible. Any rule can be selected and applied to a node in any order as soon as its guards are satisfied. The rules assume *temporal atomicity* (they are composed by a small and fixed number of assignments and tests) and *spatial atomicity* (they need exclusive access to a fixed and small number of neighboring nodes). This approach was first undertaken by H.T. Kung and P.L. Lehman [6]. Later on, inspired by on-the-fly garbage collection algorithms [3], J.L.W. Kessels [5] found the first safe and live algorithm for AVL trees.

In a recent paper, O. Nurmi and E. Soisalon-Soininen [7] introduce an extension of Red-Black trees called Chromatic trees. Chromatic trees support concurrent rebalacing after a set of insertions or deletions. The authors are mainly interested in the concurrent rebalancing after a set of deletions. To deal adequately with this problem Chromatic trees verify:

1. The property (P_4) of the Red-Black is maintained.

2. Condition (P_3) is relaxed accepting overweighted nodes. In a Chromatic tree, a red node n has **color**$(n) = 0$ as before. The situation changes with black nodes which can be *overweighted* such that **color**$(n) \in \{1, 2, \ldots\}$.

As in the case of Red-Black trees the **blackness**(n) is a well defined function. To maintain this function after a set of deletions, the remaining nodes need to "charge with the redness" of missing black sons. This is the cause of **color**$(n) \in \{1, 2, \ldots\}$.

In the Chromatic trees, *many new red nodes can be attached before some rebalancing process starts*. Therefore, big clusters of red nodes can be generated. As O. Nurmi and E. Soisalon-Soininen [7] are mainly interested in deletions, their concurrent rebalancing algorithm has the following drawback:

GRANDPARENT BOTTLENECK: Only nodes having a black grandparent can be updated. Therefore, only the top nodes of a red cluster can be updated. It is impossible to update these clusters *in the middle*. This constraint highly decreases the degree of concurrency because almost all nodes in a red cluster are "frozen".

In our paper we overcome the above drawback with the following improvement:

IMPROVEMENT: The redness of a node can be propagated even though the parent was red. Therefore, it is possible to work concurrently "in the middle" of red chains. Therefore it exists a better degree of concurrency.

This means that we deal with a new sort of *hyper-red* nodes, and a new sort of rules and trees. In our case, nodes cannot be deleted, therefore we need just *one* black color (in the case that a node is black **color**$(n) = 1$). However nodes can be inserted, and we accept *many red colors* (if a node is red **color**$(n) \in \{0, -1, \ldots\}$). These two assumptions are at the base of the HyperRed-Black trees introduced in this paper.

The rest of paper is organized as follows. In section 2 we introduce HyperRed-Black trees. Section 3 is devoted to explain the set of rules defined to overcome the grandparent bottleneck. In section 4 we consider the correctness of the concurrent rebalancing. Sections 5 and 6 are devoted to the experimental evaluation and comparison with the Chromatic approach. Finally we conclude our work introducing some open questions and raising a conjecture.

2 HyperRed-Black trees

Our improvement consists in that the redness can be accumulated in nodes, then we accept nodes having a powerful range of red tones. We define:

1. A node n is *black* iff **color**$(n) = 1$.
2. A node n is *red* iff **color**$(n) = 0$.
3. A node n is *hyper-red* iff

$$\textbf{color}(n) \in \{-1, -2, -3, \cdots\}.$$

Two reasonable consequences of redness accumulation are the following:

- The condition P_3, which forces red nodes to have black sons, must be ignored. Notice that two hyper red sons may propagate some redness units up and remain hyper-red.
- The blackness function also takes into account the hyper-red nodes, **blackness**$(n) = \sum$ **color**(u) being the sum on any path from, but not including, a node n to a leaf.

Therefore we obtain the following relaxed version of Red-Black trees so called HyperRed-Black trees:

Definition 1 *An* HyperRed-Black *tree is a binary search tree satisfying the following conditions:*

P'_1 *: Every node is either red, black or hyper-red.*

P'_2 *: Every leaf (NIL) is black.*

P'_3 *: Every simple path from a node to a leaf has the same sum of colors.*

Note that a Red-Black tree is an HyperRed-Black tree with the following two restrictions which can be locally tested:

R_1 : all the colors are 0 or 1

R_2 : if a node is red then both its children are black.

In the following, we are interested in the *concurrent rebalancing* of (an arbitrary) binary search tree into a Red-Black tree. Dealing with this problem we associate to any binary search tree its *basic* HyperRed-Black tree, and we design a set of local rules such that

- they translate HyperRed-Black trees into HyperRed-Black trees, and
- they coul be applied as soon as the restrictions R_1 and R_2 of Red-Black trees are not accomplished.

We start with the following definition of *basic* HyperRed-Black tree:

Definition 2 *Given a binary search tree T (having NIL nodes as a leaves) its* basic *HyperRed-Black tree is obtained coloring it as follows. The root is black* **color**$(root) = 1$. *All the internal nodes n (different from the root) are red,* **color**$(n) = 0$. *All the NIL nodes are black,* **color**$(NIL) = 1$.

For any basic HyperRed-Black tree it holds for any interior node n, **blackness**$(n) = 1$, and for all leaves, **blackness**$(leaf) = 0$.

3 Rules

We design a set of rules dealing with red clusters. To do this we focus the following problems:

1. The interior of a red cluster does not contain black nodes. Brothers are colored red or hyper-red, then their redness must be moved up. The following *Propagation rules* solve this problem.
2. In the frontier of a red cluster there are red or hyper red nodes having black brothers. A way to solve the problem consists to rotate black nodes down. The *rotation rules* are given to deal with this situation.
3. Finally, if the root of the tree becomes red or hyper red there is no problem to black it. The rule *blacking the root* allows us to make the root black.

For the sake of clarity, we adopt the following notation: $n(A, B)$ denotes the (sub)tree with root n, left son A and right son B. The final state of a node n once a rule has been applied is denoted n'. Unless specified, it is identical to the initial state. The figures are drawn with the same convention. We have only added the color of a node n as the superscript c_n.

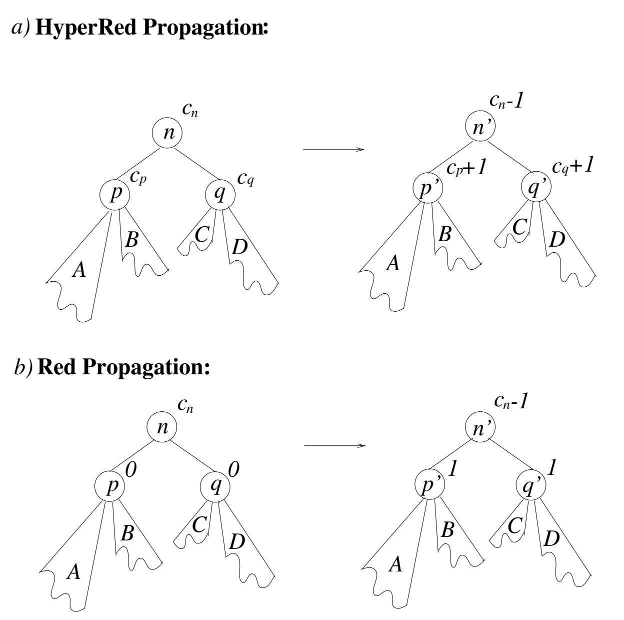

Figure 1. Propagation rule

3.1 Propagation rules

They can be applied when both brothers are red or hyper-red, and move one unit of redness up from the sons to the parent. Suppose we have a subtree $n(p(A,B),q(C,D))$, propagation can appears in two different cases:

1. If **color**$(p) < 0$ and **color**$(q) \le 0$ then there is, at least, one hyper red node p whose redness excess must be propagated up. The following *hyper red propagation* looks after this first case. We have also the symmetrical case.

2. If **color**$(p) =$ **color**$(q) = 0$ then both nodes are red. If one of the roots of A, B, C, D is red or hyper-red we have a chaining of two red nodes. The *red propagation* rule looks after this second case.

Rule : HyperRed Propagation

Guard: A subtree $n(p(A,B),q(C,D))$ such that p is hyper-red and q red or hyper-red (see Figure 1a).

Behavior: Update colors with
color$(n') =$ **color**$(n) - 1$
color$(p') =$ **color**$(p') + 1$
color$(q') =$ **color**$(q) + 1$

Spatial Scope: Nodes n, p, q.

Note: If **color**$(p) < 1$ and **color**$(q) < 0$ we have the symmetric case.

Rule : Red Propagation

Guard: A subtree $n(p(A,B),q(C,D))$ such that **color**$(p) =$ **color**$(q) = 0$ and, at least, one of the roots of the subtrees A, B, C and D is not black (see Figure 1b).

Behavior: Update colors with
color$(n') =$ **color**$(n) - 1$
color$(p') =$ **color**$(q') = 1$.

Spatial Scope: Nodes n, p, q.

Due to the local character of propagations, we have the following easy lemma:

Lemma 1 *The two rules* HyperRed Propagation *and* Red Propagation *transform any HyperRed-Black subtree into another HyperRed-Black subtree.*

3.2 Rotation rules

They are applied when one son cannot propagate the redness up because its brother is black. This obstruction is weakened by rotating down the black brother. For instance, given $n(p(\ldots),q(\ldots))$ with p red or hyper-red and q black, a single right rotation around node n gives the new root p' and moves down the black node q. But the main drawback appears with the maintenance of the property P_3' which says that the sum of colors on any path from the root to any leaf must remain unchanged. Note that the color of p only adds to paths which traverse across its sons, then it cannot carry its color when it is rotated. Therefore node p is forced to be red, and this fact is only accomplished by restoring the hyper redness of p down to its sons: *rotations undo previous propagations!*

We consider two cases depending on the redness of p. In both cases we *do not need* n black, node n can have any color.

1. If **color**$(p) < 0$ we deal with *hyper-red rotation rules*. These rules propagate the excess of redness of p down and rotate around node n.

2. If **color**$(p) = 0$, node p is red. The problem appears when p has a red or hyper red son. In this case we have a chain of red nodes. Therefore *red rotation rules* appear as a generalization of the sequential case because they deal with a chaining of two nodes.

HyperRed rotations: They occur when p is an hyper-red node. After the color of p is propagated down, a single or double rotation should be done. We perform the rotation which moves more redness up. For instance, if r is "more red" than s (formally **color**$(r) \le$ **color**(s)) a single rotation moves the redness of r up. Otherwise we need a double rotation. Let us formally define both cases.

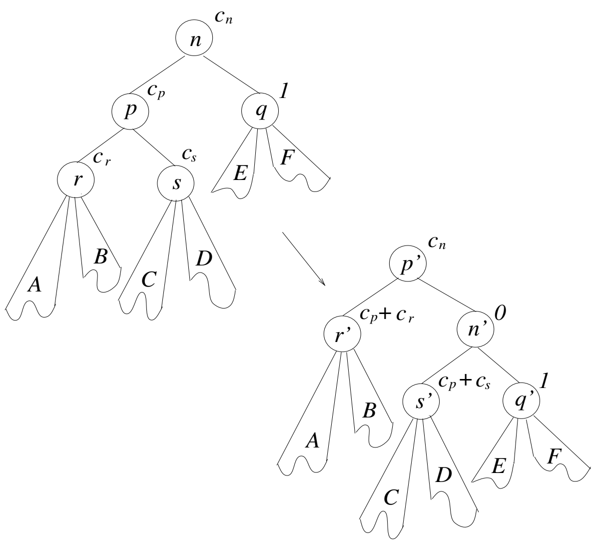

Figure 2. Single HyperRed Rotation rule

Rule : Single HyperRed Rotation

Guard: $n(p(r(A, B), s(C, D)), q(E, F))$ is a subtree such that p is hyper-red, q is black and **color**$(r) \leq$ **color**(s).

Behavior: It performs a single right rotation around n getting $p'(r'(A, B), n'(s'(C, D), q'(E, F)))$. Colors are updated as (see Figure 2)

color$(p') =$ **color**(n),
color$(n') = 0$,
color$(r') =$ **color**$(p) +$ **color**(r),
color$(s') =$ **color**$(p) +$ **color**(s).

Spatial Scope: Nodes n, p, q, r, s.

Note: We have the symmetrical case: p is black, q is hyper-red and the right son of q is more red than its brother.

Recall that a double hyper-red rotations is needed in $n(p(r(\ldots), s(\ldots)), q(\ldots))$ if s is more red than r. In this case, in addition to the down propagation of p redness, the redness of s must be propagated down. Formally:

Rule : Double HyperRed Rotation

Guard: $n(p(r(A, B), s(t(C, D), u(E, F)), q(G, H))$ is a subtree with p hyper-red, q black and **color**$(r) >$ **color**(s)

Behavior: Perform a double right rotation around node n and update colors (see Figure 3).

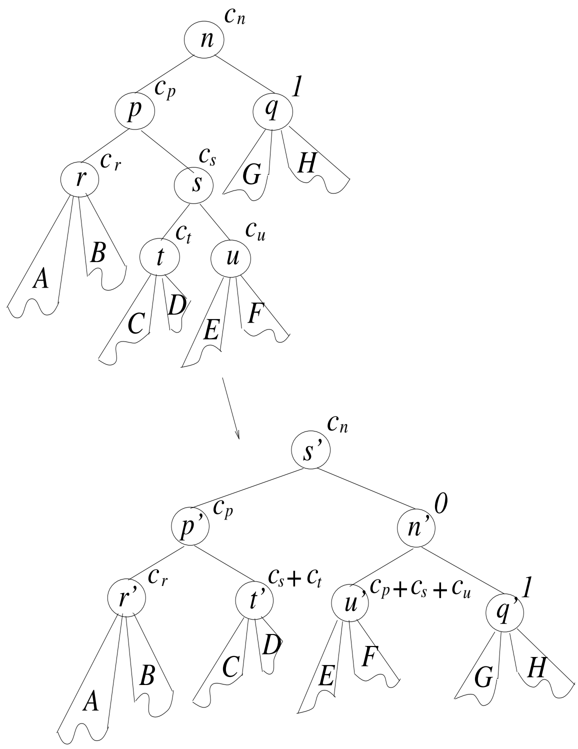

Figure 3. Double HyperRed Rotation rule

color$(s') =$ **color**(n),
color$(n') = 0$,
color$(t') =$ **color**$(s) +$ **color**(t),
color$(u') =$ **color**$(p) +$ **color**$(s) +$ **color**(u),
The colors of p, r and q do not have changed.

Spatial Scope: Nodes n, p, q, r, s, t, u, q.

Note: We have the symmetric case.

Red rotations: A subtree $n(p(r(\ldots), s(\ldots)), q(\ldots))$ needs a red rotation when p is red, q is black and one of the sons of p is red or hyper-red. Therefore we have a problem of chaining of red red nodes. As before we consider two rules. We give only the guards. To have behavior it is enough to take $c_p = 0$ in the hyper red rotation rules. It is necessary to emphasize that red rotations are not an special case of hyper-red rotations because they have different guards.

Rule : Single Red Rotation

Guard: $n(p(r(A, B), s(C, D)), q(E, F))$ is a subtree such that p red, q is black, one of the nodes r or s is not black and **color**$(r) \leq$ **color**(s).

Behavior: (see Figure 4).

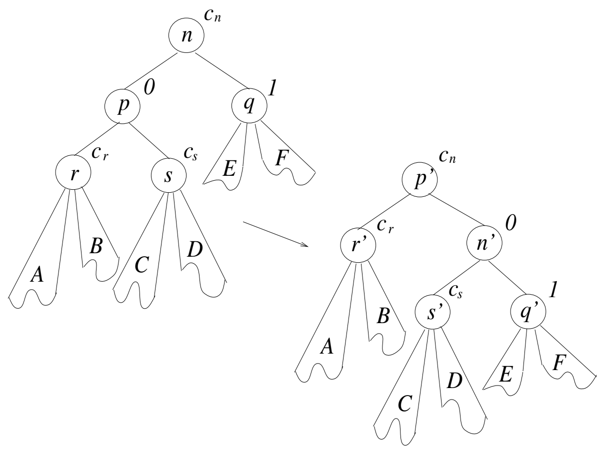

Figure 4. Single Red Rotation rule

Rule : Double Red Rotation

Guard: $n(p(r(A, B), s(C, D)), q(E, F))$ is a subtree such that p red, q is black, one of the nodes r or s is not black and **color**(s) < **color**(r).

Lemma 2 *The two* HyperRed Rotation *and* Red Rotation *rules transform an HyperRed-Black subtree into another HyperRed-Black subtree.*

Proof: Only Hyper-Red Rotation rules increase the color of some nodes, specifically node p on single rotations and nodes p, s on double rotations (s can decrease too). Node p increases from a negative value to zero, and node s, if so, from **color**(s) to **color**(n). Then none node acquire a color bigger than one, therefore property P_1' holds.
Some nodes receive redness units from their parents. But the guard of rotation rules forces these nodes to be interior nodes. Therefore leaves are maintained blacks, and property P_2' holds.
Property P_3' holds because the sum of colors from node n to a leaf is the same as the sum from node p' to a leaf (single rotation rules), or from node s' to a leaf (double rotation rules). For instance, let l be a leaf of subtree E of Figure 3. The nodes of E do not have changed, so **blackness**(u) = **blackness**(u'). As **color**(n) = **color**(s') we only take into account the color of nodes nodes p, s, u and n', u'. But **color**(p) + **color**(s) + **color**(u) = **color**(n') + **color**(u'). []

3.3 Blacking rule

As in the sequential algorithm the redness of the root can be decreased

Rule : Blacking the Root

Guard: The root is red or hyper-red.

Behavior: Take one unit of redness off and updates **color**$(root')$ = **color**$(root) + 1$.

Spatial Scope: The root.

Lemma 3 *The rule* B*lacking the Root transform an HyperRed-Black tree into another HyperRed-Black tree*

4 Correctness

The correctness is ensured by the *safety* and the *liveness* properties. The *safety* property guarantees that, whenever we start from a basic HyperRed-Black tree, any tree obtained through the rules is fine: nothing bad may happen. The *liveness* property ensures that the sequence of rules is finite.

To deal with correctness we need to consider a restricted version of HyperRed-Black trees.

Definition 3 *We call an HyperRed-Black tree* standard *if any node n verifies:*

- *The blackness cannot be negative, then we have* **blackness**$(n) \geq 0$.
- *There is enough blackness to take care of the color. More formally,* **blackness**(n) + **color**$(n) \geq 0$.

Note that any basic tree and any Red-Black tree are standard. By induction on the set of rules we obtain the following lemma:

Lemma 4 [safety]
Any HyperRed-Black tree obtained starting from a basic tree and applying a set of rules is standard. Moreover, if no rule applies, the tree is a Red-Black tree.

The *liveness* property is more difficult to prove because the hyper-red rotation rules undo previous propagations. Recall that, intuitively "hyper-red rotations move redness down and further rotate". Then we need to analyze more carefully the relationship between the blackness and the color. Given a red or hyper-red node n having color $-c$ (with $c \geq 0$) we split c into $c + 1$ *units of red* having *labels*

$$C(n) = \{c, c-1, \ldots, 0\},$$

and we determine the set of *levels*

$$\begin{aligned} L(n) \quad = \quad & \{\, \textbf{blackness}(n) - c, \\ & \textbf{blackness}(n) - (c-1),\ \ldots, \\ & \textbf{blackness}(n) - 0\}. \end{aligned}$$

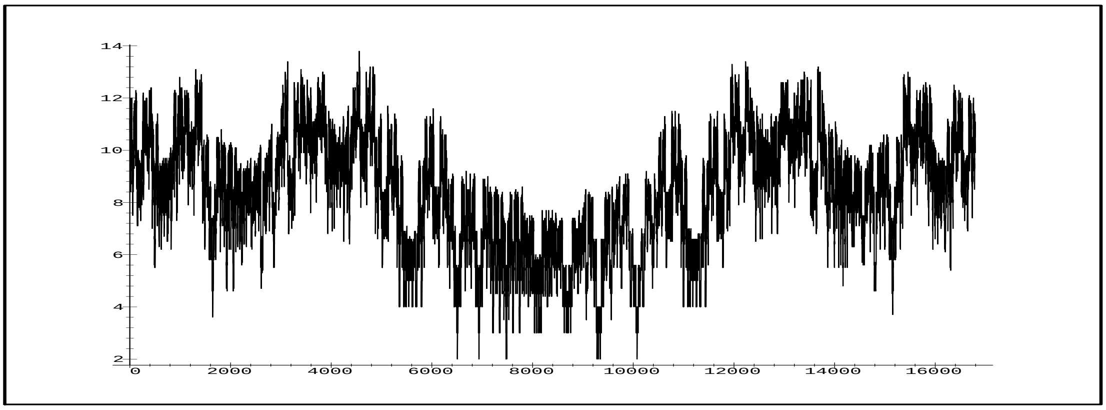

Figure 5. Number of rules applied to balance binary trees of size 10 with hyper-red rules.

We define the set of red or hyper-red nodes (with a red or hyper-red brother) such that accept a unit of redness propagation of level l:

$$P_l = \{n \mid \textbf{color}(n) \leq 0,\ l \in L(n)\ \textbf{color}(\textbf{brother}(n)) \leq 0\}.$$

Therefore, if the color of n is $-c$ and the blackness is b, then node n belongs to the following sets

$$P_{b-c}, P_{b-c+1}, \ldots P_b$$

Note that $b - c = \textbf{blackness}(n) + \textbf{color}(n)$ is the smallest index. An analog definition is given to split nodes accepting rotations:

$$R_l = \{n \mid \textbf{color}(n) \leq 0,\ l \in L(n),\ \textbf{color}(\textbf{brother}(n)) = 1\}$$

Given n, this node can be considered as the root of a subtree. We call $Inside(n)$ the number of nodes in such a tree. We define:

$$I_l = \sum_{n \in R_l} Inside(n)$$

In a given step t of the algorithm we can consider the following array (given a set S, we note as $\#S$ the number of elements):

$$\mathcal{V}(t) = \left(\begin{bmatrix} I_1 \\ \#P_1 \end{bmatrix}, \begin{bmatrix} I_2 \\ \#P_2 \end{bmatrix}, \ldots, \begin{bmatrix} I_l \\ \#P_l \end{bmatrix}, \ldots \right)$$

Given two arrays $\mathcal{V}$ and $\mathcal{V}'$, we say that $\mathcal{V} > \mathcal{V}'$ if exists an index $l > 0$ such that for all $j < l$ we have $I_j = I'_j$ and $\#P_j = \#P'_j$, and

- $I'_l < I_l$
- or $I_l = I'_l$ and $\#P'_l < \#P_l$.

The function $\mathcal{V}(t)$ can be used as a variant function in HyperRed-Black trees as we see in the following lemma.

Lemma 5 [liveness]
Any application of a propagation or a rotation rule strictly decreases the function $\mathcal{V}(t)$.

Proof: In a propagation rule from $n(p(\cdots), q(\cdots))$ to $n'(p'(\cdots), q'(\cdots))$ take $l = \textbf{blackness}(p) + \textbf{color}(p)$. Then l is the smallest index such that $p, q \in P_l$. The propagation rules increase the color of p and q. Therefore $p, q \notin P_l$, and $P'_l = P_l - 2$. The quantity I_l does not increase because R_l remains constant. It could happen that R_{l+1} increases.
In a single rotation rule $n(p(r(\cdots), s(\cdots)), q(\cdots))$ take $l = \textbf{blackness}(s) + \textbf{color}(s)$ and $b = l + \textbf{color}(p)$. Then l is the smallest index such that $r, s \in P_l$, and b the smallest index such that $p \in R_b$. As rotation rules undo previous $|\textbf{color}(p)|$ propagations, the nodes r, s will belong to the following sets with smaller indexes than l: $P_l, P_{l-1}, \ldots, P_b$. But, when node p is rotated down, the inside of R_b strictly decreases, therefore $I'_b < I_b$. []

The Blacking the Root rule does not change the variant function $\mathcal{V}(\sqcup)$. However, this variant function can be easily modified to take care of this case. Therefore the proposed algorithm is totally correct. In the following sections we will consider their experimental behavior.

5 Experimental evaluation

In this section we analyze the performance of the concurrent rebalancing algorithm. We generate the sequence of

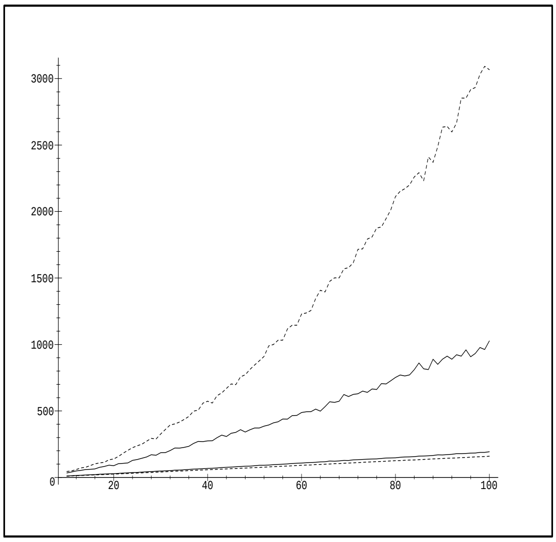

Figure 6. Number of failures (two upper lines) and successes (two down lines) needed to rebalance a linked list of $0 \leq n \leq 100$ nodes into a Red-Black tree. Solid lines depict the HyperRed-Black case. Dotted lines depict the Chromatic case.

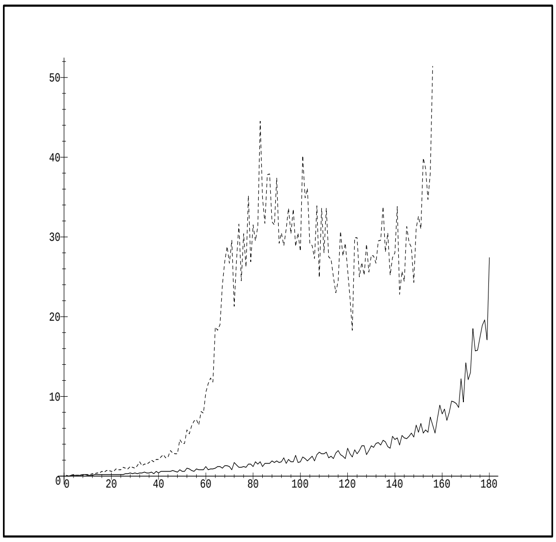

Figure 7. Number of failures between consecutive successes when a linked list (of size 100) is transformed into a Red-Black tree. Solid line depicts the a HyperRed-Black case. Dotted line depicts the Chromatic approach.

all binary trees of ten nodes, and for each one, we count the number of rules needed to balance it.

The trees have been created following M. Solomon and R. A. Finkel [9]. The first tree of size h, denoted $\mathbf{first}_h$, is the linked list such that all internal nodes are left sons, and the $\mathbf{last}_h$ tree is the symmetric one. The function **next**, which generates the sequence of trees, is defined as follows. Let $t = n(A, B)$ be a binary tree such that $size(A) = a$ and $size(B) = b$, if $B \neq \mathbf{last}_b$ then $\mathbf{next}(t) = n(A, \mathbf{next}(B))$. If $B = \mathbf{last}_b$ and $A \neq \mathbf{last}_a$ then $\mathbf{next}(t) = n(\mathbf{next}(A), \mathbf{first}_b)$. Finally, if A and B are the last trees of their size transfer one node from A to B and take $\mathbf{next}(t) = n(\mathbf{first}_{a-1}, \mathbf{first}_{b+1})$. Recall that the number of trees is determined by the *Catalan number*

$$B(n) = \binom{2n}{n} \frac{1}{n+1,}\text{, then } B(10) = 16796.$$

The function *next* locates the balanced trees in the middle of the sequence, whereas the extremes contain linear trees. Note that the firsts $B(9) = 4892$ trees of Figure 5 are those trees with $size(A) = 9$ and empty B. The following $B(8) = 1430$ trees, which start in point 4893 and end in point 6322, are those trees such that $size(A) = 8$ and $size(B) = 1$, and so on. This construction technique gives the fractal nature of the function.

For each tree we count the number of propagation and rotation rules needed to obtain a Red-Black tree. At each step, a node is selected randomly, and (if possible) updated. As the rules have exclusive guards, at most one rule applies every time. The process continues until no rule applies anywhere. The idea of random selections is commonly used in Hopfield networks [4] and it has also been used in the concurrent rebalancing of the AVL trees in [8, 1].

The figure 5 shows the *expected* number of rules for the sequence of trees. Each tree has been randomly balanced one hundred times (this quantity ensure us a relative error of data smaller than 5%). Balanced trees (located in the middle) are quickly transformed into a Red-Black trees (with less than 9 rules and in some cases only 2 rules). Unbalanced trees need more than 10 rules (and in some cases 14 rules).

6 Average behavior of Chromatic versus HyperRed-Black trees

In the following we compare HyperRed-Black trees with Chromatic trees. More specifically, we consider the average time rebalancing of trees. This time is computed by counting the number of steps needed to obtain a Red-Black tree. At each step, one node is selected randomly and (if possible) updated. It is possible to choose a *locally stable* node,

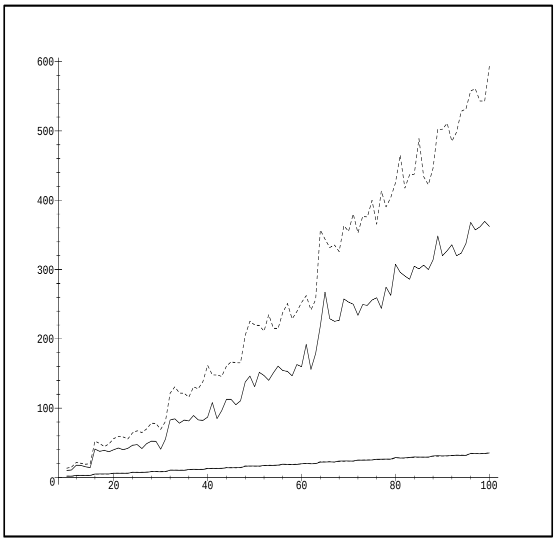

Figure 8. Number of failures (two upper lines) and successes (two almost equals down lines) needed to transform a balanced tree of $0 \leq n \leq 100$ nodes into a Red-Black tree. Solid lines depict the HyperRed-Black case. Dotted lines depict the Chromatic case.

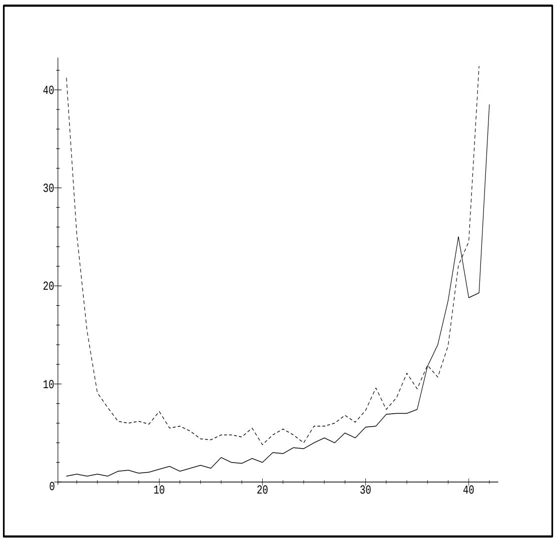

Figure 9. Number of failures between consecutive successes when a balanced tree (of size 100) is transformed into a Red-Black tree. The solid line depicts the HyperRed-Black case, the dotted line depicts the Chromatic case

where no rule applies. In this case, the rebalance of the tree does not progress at all and we count this step as a *failure*. Otherwise, the node is *locally unstable* and can be updated. We make progress and we count this step as a *success*.

We analyze two extreme cases. First, the *linked list case* in which the basic tree is a linked list. Second, the *balanced tree case* in which the basic tree is an almost complete binary tree fulfilled by levels.

- *Linked list case:* We explore the average time needed to rebalance a linear linked list into a Red-Black tree (see Figure 6). According to the experimental results the number of successes is linear in relation to the number of nodes. The HyperRed-Black approach behaves as $2.0\,n$, and the Chromatic one gives us $1.7\,n$. On the other hand, the number of failures is almost linear for the HyperRed-Black case and clearly quadratic for the Chromatic case. Specifically, we have $11.0\,n$ and $0.45\,n^2$.

  The figure 7 confirms the quadratic behavior of the Chromatic rules. Initially rotation rules can be applied on many nodes. But when the tree has become 'balanced' (after 60 successes) propagation rules must be applied, and from this time, there are $O(n)$ trials for each success.

- *Balanced tree case:* Figure 8 shows the experimental results for this case. HyperRed-Black rules behaves as $0.58\,n$ and Chromatic rules as $0.56\,n$. The number of failures seems to be almost linear, specifically $4.18\,n$ in the HyperRed-Black case and $6.36\,n$ in the Chromatic case.

  The Figure 9 shows clearly the drawback of chromatic rules. As the initial tree is balanced rotations do not take place and propagations can be made if the grandfather node is black. But the basic initial tree is a big cluster of red nodes with a black root. In this case, only the root accepts the application of a rule, and this node should be found randomly.

Of course, HyperRed-Black trees accept more rules than Chromatic trees. Therefore, it is easier to select a unstable node in a HyperRed-Black tree than in a Chromatic tree. Intuitively, HyperRed-Black trees have more freedom to evolve than Chromatic trees. However it is not clear why the increase of freedom is a good strategy. Note that, the total number of steps is minor in HyperRed-Black tree case. This fact is not obvious at all to us. The excess of freedom could generate a kind of almost-compensating effects giving very low convergence rate (these facts makes proof of safeness difficult). However, this not happens with random access.

7 Conclusions

We have designed a new type of relaxed Red-Black trees called HyperRed-Black trees. These trees overcome the *grandparent bottleneck* problem appearing in Chromatic trees because HyperRed-Black trees accept hyper-red nodes (nodes with many degrees of redness). Then HyperRed-Black trees can always propagate the redness up.

We have found a set of *propagation, rotation* and *blacking* rules. They can be applied to a node in any order as soon as their guards are satisfied. Any final tree obtained applying these rules is a Red-Black tree (*safety property*), and any sequence of applications is finite (*liveness property*).

We have implemented and compared both HyperRed-Black with Chromatic trees. The experimental conclusions are

1. HyperRed-Black trees need more updates than Chromatic trees.
2. HyperRed-Black trees fail less than Chromatic trees.
3. HyperRed-Black trees progress better than Chromatic trees.

In section 4 we have proved the convergence of the rebalancing procedure. However, the variant function suggests us bad bounds when compared with experimental results. Based on the preceding results and also in similar results obtained for AVL trees [1] we conjecture:

Conjecture 1 *The expected number of rules needed to rebalance a binary search tree is linear (in relation to the size of the tree) in Chromatic and HyperRed-Black trees.*

References

[1] L. Bougé, J. Gabarró, X. Messeguer, and N. Schabanel. Concurrent rebalancing of AVL trees: a fine-grained approach. In *European Conference in Parallel Processing (Europar'97)*, To be published by Springer-Verlag in LNCS, 1997. Also appeared as a Tech. Rep LSI-97-10R. LSI. UPC. Barcelona.

[2] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. McGraw Hill, MIT, 1990.

[3] E. Dijkstra, L. Lamport, A. Martin, C. Scholten, and E. Steffens. On-the fly garbage collection: an exercise in cooperation. *CACM*, 21:966–975, 1978.

[4] J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. In *Proc. of National Academy of Science, USA*, volume 79, pages 2554–2558, 1982.

[5] J. Kessels. On-the-fly optimization of data structures. *CACM*, 26(11):895–901, 1983.

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

[7] O. Nurmi and E. Soisalon-Soininen. Chromatic binary search trees: A structure for concurrent rebalancing. *Acta Informatica*, 33(6):547–557, 1996. Also appeared in 10th ACM PODS, 1991.

[8] N. Schabanel. Equilibrage AVL distribué d'arbres binaires de recherche. Stage de D.E.A, Printemps 1996.

[9] M. Solomon and R. Finkel. A note on enumerating binary tree. *JACM*, 27(1):3–5, 1980.