Massively Parallel and Distributed Dictionaries on AVL and Brother Trees

Joaquim Gabarró*, Xavier Messeguer*

June 19, 1997

Abstract

We present a set of local rules to deal with distributed dictionaries, having as a main advantage their possible scheduling in a highly synchronized way to get massively parallel dictionaries on AVLs. Recall that, up to now trees used in massively parallel dictionaries needed to have all the leaves at the same depth, such as 2–3 trees. Therefore, it was possible (in insertions and deletions) to reconstruct the tree bottom-up in a very regular fashion, as a pipeline of *straight plane weaves* moving up. On AVL trees the situation looks different because leaves can have different depth, therefore any weave in a pipeline is highly irregular. To solve this problem we define *virtual plane waves* allowing us to develop an EREW dictionary for k keys with k processors and time $O(\log n + \log k)$. Later on we generalize the sequential algorithms on brother trees presented by T. Ottmann and D. Wood in the same way.

Keywords: *Distributed dictionaries, Massive parallel dictionaries, AVL trees, Brother trees, Parallel pipelines.*

1 AVL and Brother Trees

Since their introduction by Adel'son-Velskiĭ and Landis [1, 5], AVL trees have been recognized as an important basic data structure. However it left open how to design massively parallel dictionaries on them. This problem was first solved for other classes of trees (2–3 trees) in 1983 by W. Paul, U. Vishkin and H. Wagener [8]. They gave us effective parallel algorithms to maintain dynamically a parallel dictionary working on "batches" of k keys simultaneously. Let us recall the approach undertaken in their work. In 2-3 trees all the internal nodes have 2 or 3 sons and all the leaves have the same depth. We assume that any leaf has two *nil* nodes as left and right sons. As usual, the *height* of a node n can be defined bottom-up. We take $\textbf{height}(nil) = 0$, therefore all the leaves have height 1 and so on. The *depth* can be defined top-down starting with $\textbf{depth}(root) = 0$. We have the following invariant for any node n belonging to a 2–3 tree:

$$\textbf{depth}(n) + \textbf{height}(n) = \textbf{height}(root) = \textbf{depth}(nil).$$

Based on it W. Paul, U. Viskins and H. Wagener [8] designed pipelines to deal with massively parallel insertions and deletions. These

*LSI, Universitat Politécnica de Catalunya, FIB/FME, C/ Pau Gargallo, 5, E-Barcelona 08028, Spain. Partially supported by the ESPRIT Long Term Research Project no. 2044 (ALCOM IT).

†Contact author: Xavier Messeguer. Internet: messeguerlsi.upc.es

pipelines can be explained intuitively in terms of traveling plane waves. Recall, for instance, the basic insertion case in which every leaf incorporates at most one new key, something like a *straight wave* (or *plane wave*) is generated at the bottom of the tree and is sent up in further steps. The wave goes to the root step by step increasing the height and decreasing the depth; it looks like a straight line going up in a picture of the tree. Formally, every pair of nodes p and q belonging to the wave front verify: $\mathbf{height}(p) = \mathbf{height}(q)$ and $\mathbf{depth}(p) = \mathbf{depth}(q)$. In the general insertion case, a pipeline of waves is generated to get something like harmonic traveling waves. These ideas were applied on B trees by L. Higham and E. Schenk [3] in 1994 and on Skip lists by J. Gabarró, C. Martínez and X. Messeguer [2] in 1995.

AVL trees are highly balanced search trees, every node n stores a key, written $\mathbf{key}(n)$, the $\mathbf{left}(n)$ and $\mathbf{right}(n)$ pointers to their sons and a pointer $\mathbf{father}(n)$ to the father. In a top down search a key k pointing to node n moves to the left son $\mathbf{left}(n)$ when $k \leq \mathbf{key}(n)$ and moves to the right son $\mathbf{right}(n)$ otherwise. To compute the height we assume $\mathbf{height}(\mathit{nil}) = 0$ and

$$\mathbf{height}(n) = 1 + \max(\mathbf{height}(\mathbf{left}(n)), \mathbf{height}(\mathbf{right}(n)))$$

therefore all the leaves have height 1. The *depth* of nodes is defined as $\mathbf{depth}(\mathit{root}) = 0$ and $\mathbf{depth}(n) = 1 + \mathbf{depth}(\mathbf{father}(n))$. Observe that, in general only the following holds

$$\mathbf{depth}(n) + \mathbf{height}(n) \leq \mathbf{height}(\mathit{root}).$$

The *balance* of a node n is given by $\mathbf{bal}(n) = \mathbf{height}(\mathbf{right}(n)) - \mathbf{height}(\mathbf{left}(n))$. AVL trees are highly balanced, $\mathbf{bal}(n) \in \{-1, 0 + 1\}$. We will say that n is *lower* if $\mathbf{lower}(n) \equiv (\mathbf{height}(n) < \mathbf{height}(\mathbf{brother}(n))$ holds (we will identify true with 1 and false with 0). Finally, let us define a new depth so called *virtual depth* such that $\mathbf{virtual\text{-}depth}(\mathit{root}) = 0$ and $\mathbf{virtual\text{-}depth}(n) = 1 + \mathbf{lower}(n) + \mathbf{virtual\text{-}depth}(\mathbf{father}(n))$. This new depth give us the invariant:

$$\mathbf{virtual\text{-}depth}(n) + \mathbf{height}(n) = \mathbf{height}(\mathit{root}) = \mathbf{virtual\text{-}depth}(\mathit{nil}).$$

Therefore it is possible design *virtually plane waves* such that every pair of nodes p and q in to the wave front verify: $\mathbf{height}(p) = \mathbf{height}(q)$ and $\mathbf{virtual\text{-}depth}(p) = \mathbf{virtual\text{-}depth}(q)$ as in 2-3 trees. Based on the preceding ideas we can design a pipeline flowing bottom-up the tree. If we assume that information's flow does not modifies the AVL, we will call this tree *static* (in relation to the information) and we derive the following sheme:

PIPELINE SCHEME FOR STATIC AVLs: using *virtual depth*, the leaves become aligned at the bottom of the AVL. It is possible to start and move up a *virtual plane wave*. When a wave moves up the *height* increases and *virtual depth* decreases. We can chain virtual plane waves to get a pipeline in a static AVL.

Later on, we will consider another approach based on Brother trees introduced by Ottmann, Six and Wood [6]. They give us another way to think about AVL trees close to 2–3 trees. In brother trees all the leaves have the same depth, internal nodes can have one or two sons, but each node with only one son has a brother with two sons. To transform an AVL tree to a Brother tree we need to add a *white node* with no information between n and $\mathbf{father}(n)$ every time $\mathbf{lower}(n)$ holds [6]. Note that $\mathbf{virtual\text{-}depth}(n)$ in an AVL tree becomes the *real* depth on the transformed Brother tree. Therefore the preceding pipeline scheme can also be applied to Brother trees.

In this paper we address the following two points:

(1) The preceding scheme shows us how to pipeline information across AVL (or

Brother) trees in a very regular fashion. This can be done chaining virtual plane waves. The preceding scheme assumes the tree does not change while information flows up. A major problem occurs when the wave front dynamically perturbes the structure of the tree. This happens in insertions and deletions. When a wave front modifies the tree, succesive waves will be highly perturbed and collisions become possible. Therefore, to design massive parallel algorithms to deal with insertions or deletions is not straightforward.

(2) We would like to consider parallel and distributed algorithms in a very general and common setting. In fact, we were looking for PRAM algorithms in AVLs for a long time, but we were only successful in obtaining them after an in-depth study of the properties in distributed algorithms. General invariants in distributed algorithms give us enough freedom to go on. In our opinion, this is a good work methodology. First try to solve the problem for a distributed environment with a set of local rules and adequate global invariants. Later try to get the massive parallel algorithm coupling the rules of the distributed algorithm with a tightly synchronized scheduler.

2 Massively parallel and distributed insertions on AVLs

Let us deal with parallel insertions. Following the idea of J.L.W. Kessels [4] we will assume that a newly inserted node does not count to compute the height. A way to do it consists to color it. This color will be stored into a new register $\mathbf{color}(n)$ and the algorithm takes this fact into account.

2.1 Red relaxed AVLs

We assume that the old nodes on AVL are white and the new ones to be inserted are red. Viewing nil nodes as nodes to by filled with new information, it makes sense to colour them red: $\mathbf{color}(nil) = \mathbf{red}$. We define the predicates $\mathbf{red}(n) \equiv (\mathbf{color}(n) = \mathbf{red})$ and $\mathbf{white}(n) \equiv (\mathbf{color}(n) = \mathbf{white})$. Intuitively, red nodes mean two things, first they do not count to compute the height [4] and second, they represent an unstable perturbation to be propagated up or erased as soon as possible. To develop the first idea we recall from [4] the definitions of the dynamic magnitude height, balance and lower of a node. The *dynamic height*, written $\mathbf{dheight}(n)$, is defined such that if n is a red leaf $\mathbf{dheight}(n) = 0$, if n is a white leaf $\mathbf{dheight}(n) = 1$, otherwise

$$\mathbf{dheight}(n) = \mathbf{white}(n) + \max(\mathbf{dheight}(\mathbf{left}(n)), \mathbf{dheight}(\mathbf{right}(n))).$$

To keep consistency we assume $\mathbf{dheight}(nil) = 0$. Based on this height, the *dynamic balance* of a node is defined as 0 when n is a leaf or *nil*, otherwise

$$\mathbf{dbal}(n) = \mathbf{dheight}(\mathbf{right}(n)) - \mathbf{dheight}(\mathbf{left}(n))$$

and the predicate lower can be redefined calling it *dynamic lower*, such that $\mathbf{dlower}(n) \equiv (\mathbf{dheight}(n) < \mathbf{dheight}(\mathbf{brother}(n)))$. A tree is called a *relaxed AVL with red nodes* (or a *red relaxed AVL*) when any node n verifies $\mathbf{dbal}(n) \in \{-1, 0, 1\}$, look [4]. Observe that when all the nodes are white, a red relaxed AVL is an AVL.

Let us modify the virtual depth in order to take care of the color. If a node does not count to compute the dynamic height, it seems obvious that it cannot count to compute the depth. We define the *virtual dynamic depth* of a node n, written as $\mathbf{virtual\text{-}ddepth}(n)$, in such a way that $\mathbf{virtual\text{-}ddepth}(root) = 0$

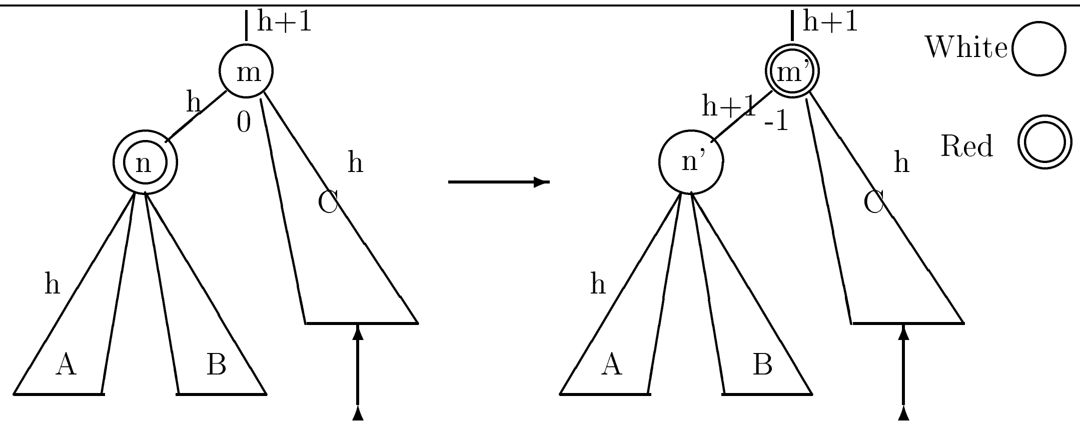

Figure 1: Red Propagation

and for any node, different from the root:

$$\textbf{virtual-ddepth}(n) = \textbf{white}(n) + \textbf{lower}(n) + \textbf{virtual-ddepth}(\textbf{father}(n)).$$

The following easy lemma gives us a precise statement of the preceding intuitions.

Lemma 1 *In a* red relaxed AVL *any node n verifies*

$$\textbf{dheight}(n) + \textbf{virtual-ddepth}(n) + \textbf{red}(n) = \textbf{dheight}(root) + \textbf{red}(root).$$

All the nil nodes have the same virtual dynamic depth *because*

$$\textbf{dheight}(root) + \textbf{red}(root) = 1 + \textbf{virtual-ddepth}(nil).$$

Based on the preceding lemma we come to our second design scheme. It will allow us to deal with waves perturbing the data structure dynamically. The red color and the dynamic magnitudes will allow us to maintain the virtual plane character of the waves.

RED DYNAMIC PIPELINE SCHEME: Using *virtual dynamic depth*, the leaves become aligned at the bottom of a red relaxed AVL. Thus it is possible to start and move up a *virtual dynamic plane wave* (also called *virtual plane waves* or *plane waves* or just *waves*). When the wave moves up the *dynamic height* increases and the *virtual dynamic depth* decreases. We can chain waves to get a pipeline in red relaxed AVL.

2.2 Local rules for insertions

We write the local rules obtained from the well known sequential insertion algorithm and we extend this set of rules to take care of parallelism and of prevention of deadlock. The rules are an extension (induced by the parallel case) of those given by J.L.W. Kessels [4]. Essentially, the rules move red nodes up (maintaining the dynamic height and balance) until they come to the root and disappear transforming a red relaxed AVL into an AVL. We only give complete versions of the most important rules, the others will be sketched. We adopt the following notation: $n(A, B)$ denotes the (sub)tree with root n, left son A and right son B. By convention, the final state of a node n is denoted as n'.

First, let us consider the rules issues from the sequential case. In this case *just one* red node moves up the color. There are two cases (propagations or rotations) depending on the dynamic balance of the father. Figure 1 deals with one of the cases of the following propagation rule.

Rule 1 (Red Propagation)

Guard: *A subtree $m(n(A,B),C)$ such that m is white, n is red with a white son and* $\mathbf{dbal}(m) \in \{0,1\}$.

Behavior: *The tree evolves new tree to $m'(n'(A,B),C)$. If* $\mathbf{dbal}(m) = 0$, *we have*

$$\begin{aligned}
\mathbf{color}(m') &:= \mathbf{red} \\
\mathbf{color}(n') &:= \mathbf{white} \\
\mathbf{dbal}(m') &:= -1 \\
\mathbf{dbal}(n') &:= \mathbf{dbal}(n)
\end{aligned}$$

Otherwise $\mathbf{dbal}(m) = 1$ *nodes m' and n' become white and*

$$\begin{aligned}
\mathbf{dbal}(m') &:= 0 \\
\mathbf{dbal}(n') &:= \mathbf{dbal}(n)
\end{aligned}$$

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

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

We will need a rule to deal with a red root, therefore we assume a rule:

Rule 2 (Root's whitening)

This rule colours the root white (increasing the dynamic height by one). There is another class of rules, so called *colored rotations*, coupling the propagation of a red color with a usual rotation in order to keep the relaxed nodes balanced. As usual we have a single and double version.

Rule 3 (Red Rotation)

Guard: *A subtree $n(p(A,B),C)$ such that* $\mathbf{dbal}(n) = \mathbf{dbal}(p) = -1$ *the node n is white and p is red.*

Behavior: *Restructure the tree into $p'(A,n'(B,C))$ with the usual updating for keys and left and right pointers. The color of n' and p' is white and its dynamic balance is* 0.

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

Note: *Symmetrically for the right-right case: $n(A,q(B,C))$,* $\mathbf{dbal}(n) = \mathbf{dbal}(q) = -1$, *$n$ is white and q is red. Colored double rotations will also appear.*

Let us consider now the rules issues from the parallel case. These rules extend those given by J.L.W. Kessels [4]. The first case happens in a subtree $n(p(A,B),q(C,D))$ when both sons of n are red (this case cannot appear with only one key to be inserted).

Rule 4 (Parallel Red Propagation)

Guard: *A subtree $n(p(A,B),q(C,D))$ such that n is white and its sons p and q are red. There are no conditions on the dynamic balances.*

Behavior: *The dynamic balance of the nodes remain unchanged and:*

$$\begin{aligned}
\mathbf{color}(n') &:= \mathbf{red} \\
\mathbf{color}(p') &:= \mathbf{white} \\
\mathbf{color}(q') &:= \mathbf{white}
\end{aligned}$$

Spatial scope: *Nodes n, p and q.*

As the preceding propagation allows us to have red balanced nodes, the set of rotations have to be extended outside of J.L.W. Kessels colored rotations. We do this with the so called (we give only the name and the number)

Rule 5 (Parallel Red Rotation)

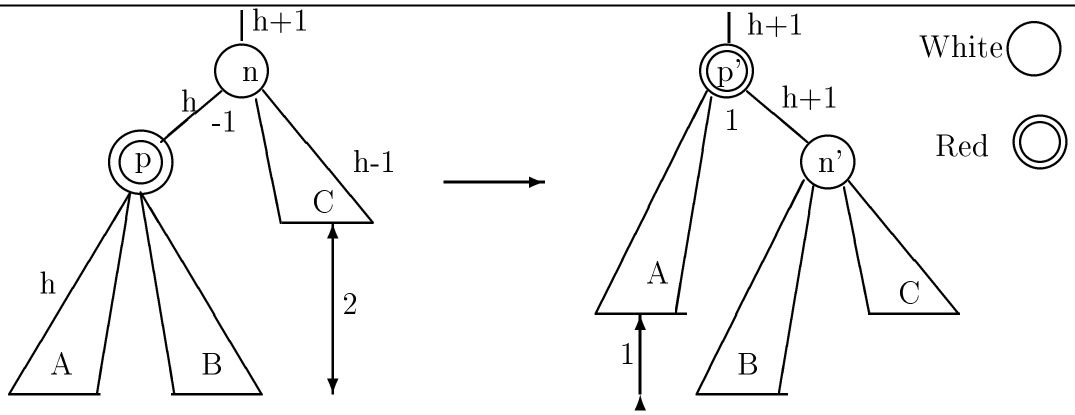

Figure 2: Red Rotation

Figure 2 explains one of the two symmetric cases: $\mathbf{dbal}(n) = -1$ and $\mathbf{dbal}(p) = 0$. Finally we need to consider the rules issues from the distributed case. To avoid deadlock problems we need, following with J.L.W. Kessels in [4], a rule (we give only the name and the number):

Rule 6 (Consecutive Red Nodes)

All the preceding rules have been designed to obtain the following lemmas. If all the rules were given explicitly, both could be proved by (lenghtly) straightforward case analysis.

Lemma 2 *Any application of a rule, keeps the tree dynamically balanced and*

(1) a propagation rule verifies

$$\mathbf{dheight}(m') = \mathbf{dheight}(m),$$

(2) a colored rotation maintains the dynamic height of the subtree, for instance in single right rotation

$$\mathbf{dheight}(p') = \mathbf{dheight}(n),$$

(3) when the root changes from red to white it holds

$$\mathbf{dheight}(root') = 1 + \mathbf{dheight}(root).$$

White nodes can be seen as stable nodes while red nodes are like bubbles going up. It seems to bre a good idea to "give the control" to the white nodes and call a node n *active* if it is white but has at least a red son.

Lemma 3 *Let n be an active node in a red relaxed AVL.*

(1) Any propagation rule keeps constant the dynamic balanced depth of any node different from n and

$$\mathbf{virtual\text{-}ddepth}(n') + \mathbf{red}(n') \\ = \mathbf{virtual\text{-}ddepth}(n).$$

(2) Any colored rotation can only modify the dynamic balanced depth of nodes in its scope. If r' is the new root of the rotated subtree we have

$$\mathbf{virtual\text{-}ddepth}(r') + \mathbf{red}(r') \\ = \mathbf{virtual\text{-}ddepth}(n).$$

2.3 Algorithms for insertions

$\boxed{\textbf{Massive parallel algorithm}}$ We will take the usual set of rules and apply them in a synchronized way giving a PRAM approach close to that of W. Paul, U. Vishkin and H. Wagener [8] for 2–3 trees. It is based on our second design scheme. We consider three cases.

(1) Assume all the keys to be inserted are attached to a white node. More formally, any red node (different from nil) with dynamic height 0 has a white father. The massive parallel algorithm can be sketched as follows. While there are active nodes,

- all active nodes with dynamic height 1, applies to each node the corresponding rule,
- all active nodes with dynamic height 2, applies to each one the corresponding rule,
- iterate, increasing the dynamic height until it becomes the dynamic height of the tree,

if the root becomes red, white it and increase the dynamic height of the tree. In this algorithm, we can see the set of active nodes as a unique wave going bottom-up the tree. The following lemma considers if this wave is "plane" going up.

Lemma 4 *The nodes that belong to the front wave have the same* virtual dynamic depth.

Proof *Observe that in the preceding algorithm two active nodes p and q belonging to the front wave verify* $\mathbf{dheight}(p) = \mathbf{dheight}(q)$. *Since throughout the whole algorithm the tree is a red relaxed AVL, for any node n it follows that*

$$\mathbf{dheight}(n) + \mathbf{virtual\text{-}ddepth}(n) + \mathbf{red}(n) = \mathbf{dheight}(root) + \mathbf{red}(root).$$

Using that any active node n in the wave is white and also is the root (except perhaps at the very last step), any node in the front wave verifies

$$\mathbf{dheight}(n) + \mathbf{virtual\text{-}ddepth}(n) = \mathbf{dheight}(root).$$

Therefore, the expression *virtual red plane wave* makes sense because the wave behaves as a plane wave using the dynamic height and the virtual dynamic depth. As all the active white nodes have the same dynamic height and dynamic balanced depth it makes sense to assign a dynamic height and a dynamic balanced depth to the virtual plane wave w written $\mathbf{dheight}(w)$ and $\mathbf{virtual\text{-}ddepth}(w)$.

(2) Let us consider the case of an AVL with a "red barbe on the bottom". By this we mean that all the red nodes, containing a new key to be inserted, are at the bottom of the tree and all the other nodes are white. To solve this case we would like to pipeline different virtual plane waves. To do this we need to prove that two different waves do not collide if they are initially separated from one another. Let us prove that, any virtual plane wave will not be affected by the behaviour of other waves higher in the tree. It holds:

Theorem 1 *Take a relaxed AVL having red nodes at the bottom. If we start from the bottom a virtual plane wave w moving up and λ (take for instance $\lambda = 10$) steps later we start another one w' moving up at the same rate, the wave w' will remain virtually plane and moreover*

$$\begin{aligned}\mathbf{dheight}(w) - \mathbf{dheight}(w') &= \mathbf{virtual\text{-}ddepth}(w') \\ &\qquad - \mathbf{virtual\text{-}ddepth}(w) \\ &= \lambda\end{aligned}$$

while w and w' are moving up.

(3) Finally let us consider the case where we have an AVL with n keys and we have to insert an ordered set of k new keys stored in the array $a[1..k]$. We recall the search algorithm by packet routing [8, 3, 2]. Moreover, as this search algorithm in a binary search tree is formally identical to a search by packet routing on Skip lists, the results about read conflicts [2] apply directly and we have that at most three subpackets can remain on a node.

When the subpackets are located at the bottom of the AVL, the divide and conquer approach given in [8] allows us to start a pipeline and the results of (2) assure that

Theorem 2 *The massively parallel insertion of k keys into a red relaxed AVL tree with n keys takes time $O(\log n + \log k)$ using k processors on an EREW PRAM.*

Distributed algorithm In this case we take the set of rules and keep the control as non-deterministic as possible, that means that any rule can be selected and applied to the structure in any order as soon as its guard is satisfied. We consider that all keys have been percolated through the tree, left at the bottom and colored red. Thus the initial tree is composed of an AVL tree with white nodes up and red nodes at the bottom.

Lemma 5 *The insertion distributed algorithm is deadlock free. Let T be an AVL tree with red nodes at the bottom, when no rule applies all nodes are white. Define $outside(n)$ as the number of nodes not contained into the subtree rooted at n, $NUMBER = \sum_{red(n)} 1$ and $RED = \sum_{red(n)} outside(n)$, for any application of a rule, the variant $(NUMBER, RED)$ strictly decreases.*

3 Massively parallel and distributed deletions on AVLs

When some nodes are deleted the tree can become lower. In order to maintain the dynamic magnitudes we introduce *green nodes* which count twice (green nodes are a complementary notion of red nodes). We assume that the set of keys to be deleted does not contain repetitions and any key to be deleted really appears in the tree. Let us sketch the version of the sequential deletion algorithm we will use.

- Request phase: A *request to delete* the key k is introduced at the root and percolated down through the tree until it stops at a node n at the bottom of the tree. If $k \leq \mathbf{key}(n)$ the value k is stored in the so called *left request register* otherwise it is stored in the *right request register*. A *request manager* of n (its father or grandfather) will take care of this request. Depending on the cases, the node n will be deleted, a *green node* will be generated (to maintain the AVL character), an *update set* written $\{\mathbf{key}(n) := k\}$, meaning "the value $\mathbf{key}(n)$ will substitute k on the tree" will be generated.

- Balance and update phase: a green node will move up the tree until it disappears or comes to the root. Update comes up until it arrives at a node holding the key k and substitutes k by $\mathbf{key}(n)$.

Based on these ideas, we develop green relaxed AVL. Later on we sketch a set of local rules maintaining the relaxed balance and finally we sketch massively parallel and distributed algorithms.

3.1 Green relaxed AVLs

The deletion algorithm starts by percolating down any deletion request. Therefore we assume that any node n has two new registers called **left-request**(n) and **right-request**(n). At the end of this percolation phase any deletion request will be located at one request register. The following easy property holds,

Lemma 6 *If the set of keys to be deleted does not contain any repetition and all these keys belong to the tree, any leaf n will contain at most two deletion requests: one located at* **left-request**(n) *and the second in* **right-request**(n).

If a leaf n has a request we will say that n is the *request holder* of this request. Moreover if n has a father m, this father will we charged to deal initially with this request, and we will say that m is the *request manager* of this request. As before we will color the nodes of the tree to get a dynamic balance. The strategy is just the opposite of the insertion case. For instance, when a leaf p is deleted in the subtree $n(p, nil)$, it becomes just a leaf n', therefore node n' should *count twice* in order to maintain the height. To take care of this fact, we color n' green. Therefore, nodes can be white or green, and green nodes counts twice. In deletions, the *dynamic height* of a node, is given by $\mathbf{dheight}(n) = 2$ if n is a green leaf, $\mathbf{dheight}(n) = 1$ if n is a white leaf, otherwise

$$\mathbf{dheight}(n) = 1 + \mathbf{green}(n) + \max(\mathbf{dheight}(\mathbf{left}(n)), \mathbf{dheight}(\mathbf{right}(n))).$$

We define $\mathbf{dheight}(nil) = 0$ and $\mathbf{color}(nil) = \mathbf{white}$, however in deletions nil nodes do not play an important role. Of course, all the request holders and all the request managers appear at the bottom of the tree. Formally, any *request holder* has dynamic height 1 or 2 and *request managers* have dynamic height 2 or 3.

As we said before, we need one more register. For instance, assume we have a subtree $n(p, q)$ and at the very bottom of the tree leaves p and q. Further suppose that p, with key k, holds only a right request d, meaning "the key d has to be deleted", written $\mathbf{right\text{-}request}(q) = d$. In this case following the ideas sketched in the sequential algorithm, the key k will be used to substitute the key d upward in the tree. Therefore, the algorithm deletes node q and the tree evolves to $n'(p', nil)$ and node n' stores the set $\{d := k\}$ meaning "the key k will substitute the key d upwards in the tree". When updates go up they can be pack together, we allow a new register $\mathbf{update}(n) = \{d_1 := k_1, d_2 := k_2, \dots\}$.

The *virtual dynamic depth* of a node n is given by $\mathbf{virtual\text{-}ddepth}(root) = 0$ and for any node, different from the root:

$$\mathbf{virtual\text{-}ddepth}(n) = 1 + \mathbf{green}(n) + \mathbf{lower}(n) + \mathbf{virtual\text{-}ddepth}(\mathbf{father}(n)).$$

Lemma 7 *In a* green relaxed AVL *any node n verifies*

$$\mathbf{dheight}(n) + \mathbf{virtual\text{-}ddepth}(n) - \mathbf{green}(n) = \mathbf{dheight}(root) - \mathbf{red}(root).$$

All the nil nodes have the same dynamic balanced depth *because*

$$\mathbf{dheight}(root) - \mathbf{green}(root) = \mathbf{virtual\text{-}ddepth}(nil).$$

As in the case of red relaxed AVLs we can introduce a pipeline scheme for this kind of tree based on dynamic magnitudes.

GREEN DYNAMIC PIPELINE SCHEME: With the *virtual dynamic depth* we can pipeline *virtual waves* in green relaxed AVL trees.

3.2 Local rules for deletions

Let us *sketch* the set of rules to deal with deletions. Recall that we need rules to take into account three concepts: requests, green color and update sets. As in the insertions case rules keep the tree dynamically balanced. First, let us briefly consider the rule

Rule 7 (Request Manager)

dealing with request managers located at (dynamic) height 2 or 3. This rule processes only one request at any application. If necessary, the manager:

- colors green a node at the bottom of the tree in order to maintain the dynamic height(Figure 3.*d*),
- generates a new update set $\{d := k\}$ (Figure 3.*e*),

- rotates in order to maintain the red relaxed AVL character. (Figure 3.e).

To reduce the number of cases, if both sons have requests we always give priority to the left son over the right one. Figures 3.a), b) and c) consider in detail the case of request manager of height 2 with $\mathbf{dbal}(n) = 0$, and p and q leaves. A request manager of dynamic height 3 (Figure 3d) is considered if their sons have no request.
The following rules force the update sets and green nodes to move up. These rule does not change the "topology" of the AVL tree.

Rule 8 (Update)

> **Guard:** *A subtree $n(p(A,B),q(C,D))$ all three nodes have an update set. Nodes can have any color (white or green), can have requests and some of them can be leaves.*
>
> **Behavior:** *The tree evolves to $n'(p'(A,B),q'(C,D))$. The nodes p',q' are equal to p,q' without updates. The nodes n and n' have the same color. As the update of p has the form $\{n := k\}$, therefore $\mathbf{key}(n') = k$ and $\mathbf{update}(n') = \mathbf{update}(n) \cup \mathbf{update}(q)$.*
>
> **Spatial scope:** *Nodes n,p,q.*
>
> **Note:** *All other cases are quite similar and easy.*

Rule 9 (Green Propagation)

> **Guard:** Case: One sided. *A subtree $m(n(A,B),C)$ such that m is white, n is green with a white son. Nodes m and n does not have updates. Moreover, $\mathbf{dbal}(m) \in \{0,-1\}$.*
>
> **Behavior:** *If $\mathbf{dbal}(m) = 0$, nodes m' and n' become white and*
>
> $$\mathbf{dbal}(m') := 1$$
> $$\mathbf{dbal}(n') := \mathbf{dbal}(n)$$
>
> *Otherwise* $\mathbf{dbal}(m) = -1$
>
> $$\mathbf{color}(m') := \mathbf{green}$$
> $$\mathbf{color}(n') := \mathbf{white}$$
> $$\mathbf{dbal}(m') := 0$$
> $$\mathbf{dbal}(n') := \mathbf{dbal}(n)$$
>
> **Spatial scope:** *Node n and its father m.*
>
> **Note:** *Symmetrically if n is the right son of m. The two sided green propagation and the root whitening are easy to design.*

Finally we develop green rotations (Figure 4). As before, green rotations are the complementary notion to red rotations. Note that we do not need a rule to prevent deadlock in distributed algorithm because green rotations send consecutive green nodes down.

Rule 10 (Green rotation)

> **Guard:** Case: Single *A subtree $n(p(A,B),C)$, $\mathbf{dbal}(n) = -1$, $\mathbf{dbal}(p) \in \{0,-1\}$ the node n,p are white and the root of C is green. Neither n, p, or the root of C have updates.*
>
> **Behavior:** *Restructure the tree into $p'(A,n'(B,C))$ with the usual updating for keys and left and right pointers. If $\mathbf{dbal}(p) = 0$, the nodes n',p' and the root of C becomes white*
>
> $$\mathbf{dbal}(p') := 1 \quad \mathbf{dbal}(n') := -1$$
>
> *Otherwise $\mathbf{dbal}(p) = -1$, nodes n',p' become balanced, the root of C becomes white and*
>
> $$\mathbf{color}(p') := \mathbf{green}$$
> $$\mathbf{color}(n') := \mathbf{white}$$
>
> **Spatial scope:** *Node n and its left son p.*
>
> **Note:** *The symmetrical cases for single rotation and double green colored rotations also exit.*

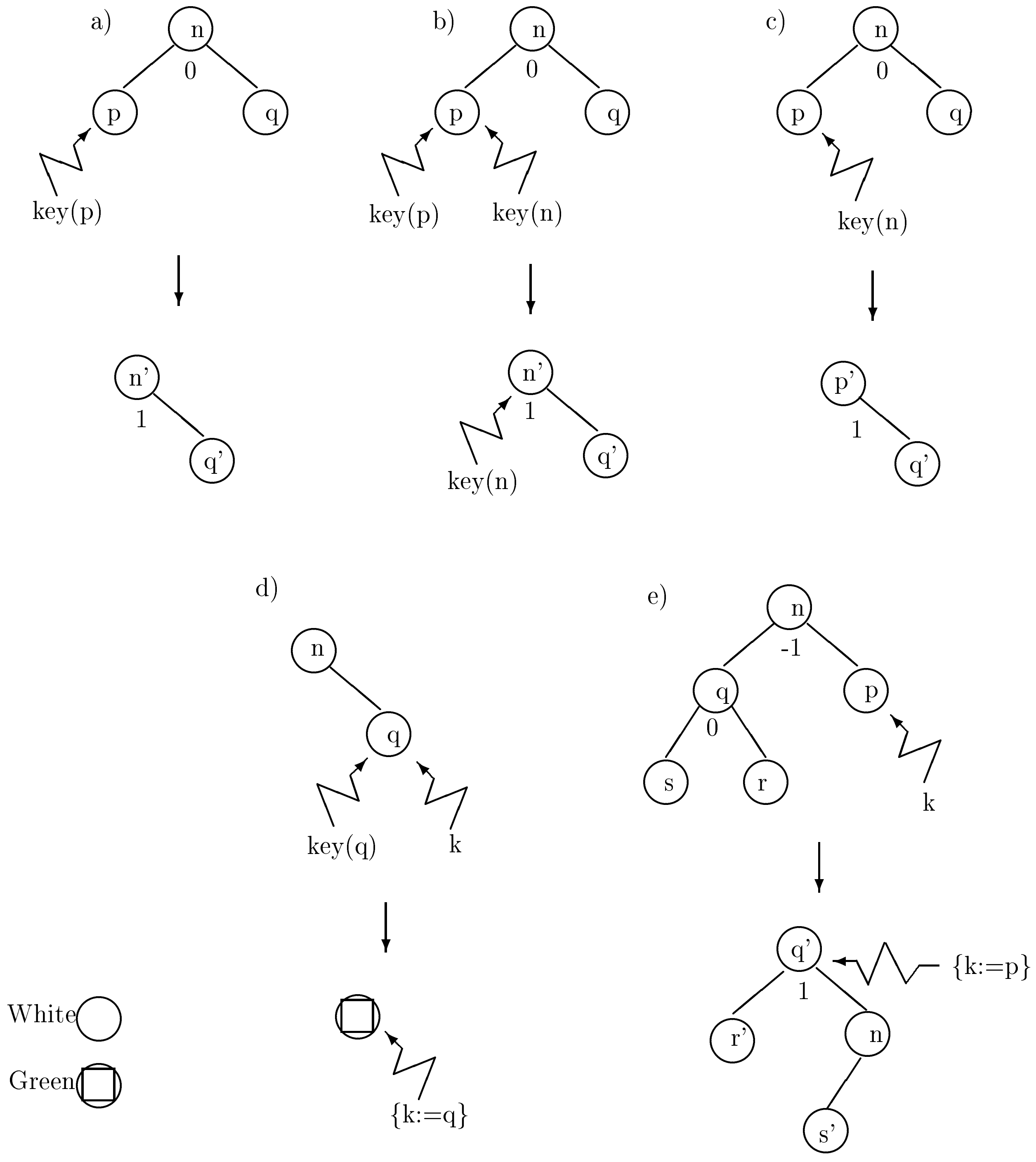

Figure 3: Request manager

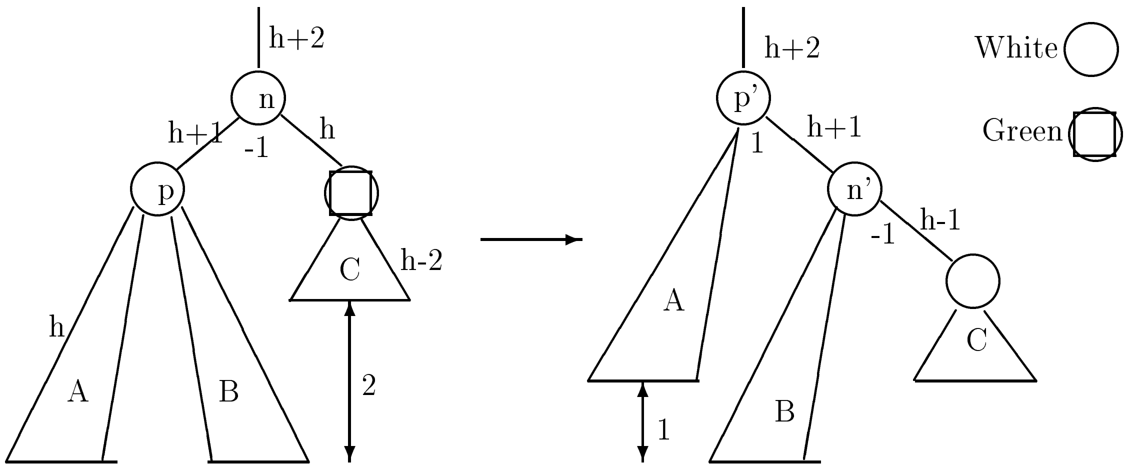

Figure 4: Green Rotation

3.3 Algorithms for deletions

Massively parallel case Assume that we have k keys to be deleted, all of them in the tree, with no repetitions. As in the insertion case we generate waves. Here we have two different types of waves:

- *Update-wave* dealing with the update sets. This wave is charged to move up the update sets and to replace the deleted keys by the new ones. This wave is generated applying the request manager rule and moves up with the update rule.

- *Green-wave* moving up green nodes. It is also generated by the manager rule. Moves up by propagation and rotation rules. This green-wave moves up applying green propagation and rotation rules. A node is active in an *green-wave* if it has a green son.

The algorithm works in phases. In each one the following happens: all request managers of dynamic height 2 start up a request. Green nodes and update sets appear. An *update-wave* starts going up and a "little bit" later a green-wave goes up. A "little bit" later the entire scheme is repeated with the request managers having dynamic height 3.

Note that a request can give us a maximum of one component in an update set an at most one green node, therefore it is possible to implement the preceding phase with at most $2k$ processors, but it is really easy to decrease this to k processors. Moreover waves look like straight waves using dynamic balanced depth. As in a phase the number of requests decreases by a constant fraction; we can pipeline phases to get:

Theorem 3 *The massively parallel deletion of k keys into an AVL tree with n keys takes time $O(\log n + \log k)$ using k processors on an EREW PRAM.*

Distributed case Recall that any rule can be selected and applied to the structure in any order as soon as its guard is satisfied. We consider that leaves have all request:

Lemma 8 *The deletion distributed algorithm is deadlock free. Let T be an AVL tree with* *all requests at the bottom, when no rule applies all nodes are white and all updates have been made. Define*

$$REQUEST = \sum_{request(n)} 1,$$

$$NUMBER = \sum_{green(n)} ,$$

$$GREEN = \sum_{green(n)} outside(n)$$

$$UPDATE = \sum_{update(n)} outside(n).$$

Any application of a rule makes the variant function

$(REQUEST, NUMBER, GREEN, UPDATE)$

strictly decreasing.

4 Brother trees

Recall that in Brother trees all the leaves have the same depth, internal nodes can have one or two sons, but each node with only one son has a brother with two sons. *In our version information is stored in the leaves and internal nodes act as routers.* To transform an AVL tree to a Brother tree we need to add a *white node*, denoted $\circ$, with no information between n and $\mathbf{father}(n)$ if $\mathbf{lower}(n)$ holds. Therefore, any internal node n different from $\circ$, has one of the following forms, $n(A, B)$, $n(A, \circ(B))$ or $n(\circ(A), B)$ such that A, B are brother trees having a binary node as a root.

Relaxed brother trees for insertions are intuitively close to the 2–3 trees because we accept nodes with two or three sons. As before, binary nodes can have one of the forms $n(A, B)$, $n(A, \circ(B))$ or $n(\circ(A), B)$. Only two forms are allowed to ternary nodes, $n(A, B, C)$ or $n(A, \circ(B), C)$ (with $\circ$ in the middle). In any case all the leaves have the same depth. We can extend the set of rules given by T. Ottmann and D. Wood [7] to deal with the parallel cases. These rules allow us to move up ternary nodes. As an example consider the following rule:

Rule 11 (Brother Rotation)

Guard: *A node n having the form $n(p(A \circ (B), C), \circ(D))$. Assume $\mathbf{key}(n) = k$ and $\mathbf{keys}(p) = \{u, v\}$.*

Behavior: *Restructure the tree as $n'(p'(A \circ (B)), q'(C, D)$ with $\mathbf{key}(n') = v$, $\mathbf{key}(p') = u$, $\mathbf{key}(q') = k$*

Spatial scope: *Node n, the roots of A and C, and the node $\circ$.*

In *Relaxed brother trees for deletions* we can have things like $n(\circ(A), \circ(\circ(B)))$ where chains of two $\circ$ appear. Extending the rules given for deletion by T. Ottmann and D. Wood [7] with the parallel cases we can get another complete set of rules to deal with deletions. As an example consider the following one:

Rule 12 (Parallel White Up)

Guard: *A node n having the form $n(\circ(A), \circ(\circ(B)))$.*

Behavior: *Restructure the tree as $\circ(n'(A, \circ(B))$.*

Spatial scope: *Node n, the roots of A and the first node $\circ$.*

As in the case of AVL trees, using dynamic depth we can define *virtual straight plane weaves.* However, in brother trees, these weaves are *really* straight line because dynamic depth is the real depth. If you make a picture of a weave you get a straight line. Following the previous ideas we can get a EREW massively parallel dictionary working in time $O(\log n + \log k)$ and k processors and also a distributed dictionary.

Acknowledgement. We thank Ray Greenlaw for his help.

References

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

[2] J. Gabarró, C. Martínez, and X. Messeguer. A design of a parallel dictionary using skip lists. *Theoretical Computer Science*, 1995. Accepted for publication. A preliminary version "Parallel Update and Search in Skip Lists" appeared in Computer Science 2: Research and Applications, pages 15-26, Plenum Press, New York, 1994.

[3] L. Higham and E. Schenks. Maintaining B-trees on an EREW PRAM. *J. of Parallel and Dist. Comp.*, 22:329–335, 1994.

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

[5] D.E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, Reading, Mass., 1973.

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

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

[8] W. Paul, U. Vishkin, and H. Wagener. Parallel dictionaries on 2–3 trees. In J. Díaz, editor, *Proc. 10th International Colloquium on Automata, Programming and Languages, LNCS 154*, pages 597–609. Springer-Verlag, 1983. Also appeared as "Parallel computation on 2–3 trees" in RAIRO Informatique Théorique, pages 397-404, 1983.