Optimal Finger Search Trees in the Pointer Machine ⋆

Gerth Stølting Brodal [a,1], George Lagogiannis [b], Christos Makris [b], Athanasios Tsakalidis [b], Kostas Tsichlas [b,*]

[a] *BRICS - funded by the Danish National Research Foundation, Department of Computer Science, University of Aarhus, Ny Munkegade, DK-8000 Aarhus C, Denmark.*

[b] *Computer Engineering & Informatics Dept. of University of Patras and Computer Technology Institute (CTI). Rio, Greece, P.O. BOX 26500.*

Abstract

We develop a new finger search tree with worst case constant update time in the Pointer Machine (PM) model of computation. This was a major problem in the field of Data Structures and was tantalizingly open for over twenty years, while many attempts by researchers were made to solve it. The result comes as a consequence of the innovative mechanism that guides the rebalancing operations, combined with incremental multiple splitting and fusion techniques over nodes.

Key words: balanced search trees, data structures, complexity

1 Introduction

The *balanced search tree* is one of the most common data structures used in algorithms. Assuming that the update position is known, balanced search trees with $O(1)$ amortized update time have been presented long ago ([9,22]). It is

⋆ A preliminary version of this paper was presented in STOC 2002.

* Corresponding author.
Email addresses: gerth@brics.dk (Gerth Stølting Brodal), lagogian@ceid.upatras.gr (George Lagogiannis), makri@ceid.upatras.gr (Christos Makris), tsak@cti.gr (Athanasios Tsakalidis), tsihlas@ceid.upatras.gr (Kostas Tsichlas).

[1] Research conducted while visiting Computer Technology Institute (CTI) and University of Patras, Greece.

known ([9,24]) that updates can be performed with $O(1)$ structural changes, but the nodes to be changed have to be searched in $\Omega(\log n)$ time. Levcopoulos and Overmars ([19]) presented an algorithm achieving $O(1)$ worst case update time, by using a global splitting lemma. This lemma is based on a pebble game that is combined with the bucketing technique of Overmars ([22]). Instead of storing single keys in the leaves of the search tree, each leaf can store a list of several keys. Unfortunately, the buckets in [19] have size $O(\log^2 n)$, and a two level hierarchy of lists is necessary to guarantee $O(\log n)$ query time. Deletions are handled by means of global rebuilding. Fleischer ([10]) presented a simpler approach to the problem. The rebalancing of the tree, is distributed over the next $\log n$ insertions into the bucket which was split. Each bucket is equipped with a pointer, pointing to an ancestor or to a node near an ancestor of the specific bucket. This pointer traverses the path from the bucket to the root. In [20], some of the authors describe a worst case constant update time search tree. Some of the ideas used in the present paper are also presented in [20] in a primitive form.

Finger search trees are search trees for which the search procedure can start from any leaf of the tree. This starting element is termed a *finger*. The time complexity of the search procedure is asymptotically equal to the logarithm of the distance between the finger and the search element. Finger search trees are fundamental data structures that have been extensively studied and used. For example, finger search trees lead to optimal algorithms for the basic operations of union, intersection and difference on sets [21] and they allow for efficient list splitting [21]. In addition, they are used for the efficient implementation of priority queues [12], for sorting nearly ordered files [12] and for sorting Jordan sequences in linear time [16]. They can also be applied to many fundamental problems of computational geometry. For example, finger search trees are used in the construction of the visibility graph of a polygon [15,11], in optimal algorithms for the 3-dimensional layers-of-maxima problem [4] and in improved methods for dynamic point location [4].

Dietz and Raman ([8]) have already developed a finger search tree with constant update time in the Random Access Machine (RAM) model of computation. Furthermore, in the same model of computation, Andersson and Thorup ([3]) have surpassed the logarithmic bound in the search procedure by achieving $O(\sqrt{\frac{\log d}{\log\log d}})$ query time. These two structures are based on a global rebalancing scheme combined with the bucketing technique presented in [19]. For the Pointer Machine model of computation ([2]), steps were made towards this direction by researchers (see [6,7,12–14,18,25]), but the problem remained tantalizingly open. The best solution was given by Brodal ([6]), who proposed a finger search tree with constant insertion time, but with $O(\log^* n)$ deletion time. This time bound of the delete operation was a direct result of the inherent difficulty in handling efficiently deletions in a local rebalancing setting.

In this paper we are presenting the first constant update time finger search tree. The skeleton of this tree is an (a,b)-tree T. The rebalancing operations on the nodes of T are guided by a novel scheduling strategy. This strategy locates the position of the rebalancing operation in worst case constant time. However, it is not an efficient strategy in the sense that the subtrees of internal nodes of T can grow quite large, before they are rebalanced. As a result, the degrees of nodes of T becomes unbounded. This problem is solved by replacing the classical split and fusion operations of an (a,b)-tree, with the MultiSplit and MultiFusion operations respectively.

A MultiSplit produces a group of new nodes from a single node, while a MultiFusion fuses a group of nodes into one node. These aggressive rebalancing operations guarantee that the degree of a node is lower and upper bounded by a function of the level of a node. Assuming that operations MultiSplit and MultiFusion can be performed in worst case constant time, a constant update finger search tree is devised. This tree T is called the *virtual tree*, because of the aforementioned assumption. After proving that the degree of the internal nodes of the virtual tree is bounded, we give the implementation of MultiSplit and MultiFusion. The resulting tree is called the *real tree* T'. In particular, MultiSplit and MultiFusion operations are implemented by introducing multiple levels of indirection and by applying incremental scheduling techniques. Finally, the description of the structure is completed with the implementation of the finger searching. The internal nodes of the real tree have gigantic degrees, and as a result, they are represented by using standard constant update time search trees. Consequently, the search operation can be performed in logarithmic time.

In Section 2, we introduce the reader to the basic notions used throughout the paper. In Section 3, the scheduling strategy for the rebalancing operations is given. In Section 4, we describe the virtual tree that incorporates the new rebalancing mechanism in its definition. In Section 5, we explain how to simulate the virtual tree while in Section 6 we discuss the implementation of the finger search procedure. Finally, in Section 7 the space complexity of the data structure is considered, and we conclude in Section 8 with some final remarks.

2 Preliminaries

Let T be a rooted tree with root r. Let P be a list $[v_1, v_2, \ldots, v_t]$ of distinct nodes of T so that v_i is the father of v_{i+1} for $i \in \{1, \ldots, t-1\}$. In this case, P is a *path* of T and the *head* of the path is node v_1. The *depth* of a node v in

T is defined recursively as follows:

$$depth(v) = \begin{cases} 0, \text{ if } v \text{ is the root} \\ 1 + depth(father(v)), \text{ otherwise} \end{cases}$$

,where *father(v)* denotes the father node of v. Similarly, the *height* of a node v is defined as follows: $height(v) = 0$ if v is a leaf and $height(v) = 1 + \max\{height(w) \mid w \text{ is a child of } v\}$ otherwise. We also refer to the height of a node v, as the *level* of v. The number of children of a node v is called the *degree* of v and it is denoted as $degree(v)$. The subtree of T rooted at v is denoted as T_v, and the number of leaves stored at T_v is the *weight* of v, denoted as w_v. The values stored in the internal nodes of a leaf oriented search tree T, are called *router values*. These values are used only as guides when searching for a given element.

Balanced search trees, are search trees that incorporate in their definition a balance criterion for the weight or height of individual subtrees. They constitute an elegant solution to the *dictionary* problem. In this problem, one needs to efficiently maintain a dynamic set of elements subject to the operations of insertion, deletion and search. In general, known balanced search trees like AVL-trees ([1]), red-black trees ([24]) and (a,b)-trees ([17]) support searches and update operations in $O(\log n)$ time, when storing n elements. In the PM model of computation, the time complexity of the search procedure cannot be reduced as the lower bound of $\Omega(n \log n)$ for sorting n elements would be violated. Since update operations are preceded by a search operation, their time complexity is also $\Omega(\log n)$. We can surpass this lower bound, if we decouple the search step and the update step of an update operation. More specifically, an update operation consists of the following three steps:

(1) Search for the position of the update
(2) Perform the update
(3) Perform rebalancing operations

The second step is performed in worst case constant time, since it involves the manipulation of a constant number of pointers and records, while the third step is usually performed in $O(\log n)$ time. The step of rebalancing re-establishes the balance criterion, which was possibly spoiled by an insertion or deletion, and guarantees that the time cost of a search operation is always $O(\log n)$. In the following, the first step of the update operation will not be considered when analyzing its time complexity. In this case, the update operation is *position given*.

There are two reasons for disregarding the first step. The first reason, is that in practice the rebalancing operations (eg. rotations, splits, fusions etc.) on a

path of nodes, are much more expensive than the traversal of this path. The second reason, is related to a more elegant analysis of the time complexity of the update operations. We would like to specify the complexity of the update operation without considering the search operation. This idea is externalized in a variant of balanced search trees, called *finger search trees*.

A finger search tree supports the following operations:

(1) *Createlist()*: creates a new list containing a single dummy element. A finger to this element is returned.
(2) *Search(f,x)*: searches for element x starting the search at the element of the list pointed by the finger f. If x exists, then a pointer to x is returned, otherwise a pointer to the largest element in the list, smaller than x is returned.
(3) *Insert(f,x)*: inserts the element x immediately to the right of finger f. It returns a pointer to x.
(4) *Delete(f)*: deletes the element pointed by the finger f.

Finger search trees are built on classical balanced search trees by introducing *level links*. A level link, is a pointer to an adjacent node. Each node has two such pointers, one to the left adjacent node and one to the right adjacent node. The search procedure starts from the given finger f and traverses the path from f towards the root, checking whether the search element x is in the subtree of a node in the path or in its adjacent nodes (using level links). When such a node is located, the search continues in the usual top-down manner.

Throughout this paper we implicitly assume that each node is associated to the interval of elements spanned by the node. This is a standard assumption for all search tree implementations and as a result we do not consider it in the remaining of this paper.

3 The Mechanism of Components

The mechanism of *components*, is based on an idea previously used in the work of Brodal for designing worst case efficient partially persistent data structures ([5]). Components define a logical partition over the set of internal nodes of an arbitrary tree into connected subtrees. These subtrees are used to dictate the positions of the rebalancing operations. A component containing a single node is called *singleton*. In particular, consider a dynamic search tree T and a set of components $\mathbb{C}$ over the internal nodes of T. A component $A \in \mathbb{C}$ is a subtree of T. The *handle* of the component A is the root of its subtree and it is represented by $A.root$. The maximum and the minimum degrees of a node at level i are functions of i with the exception of the root, for which the minimum

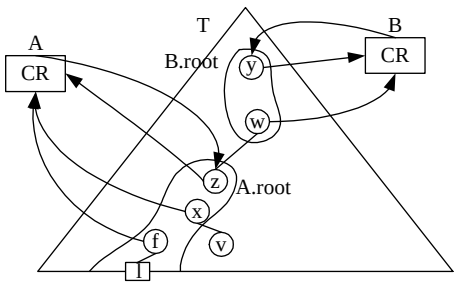

Fig. 1. The structure of a component A in a tree T. By CR we represent the component record.

degree is 2. The tree structure T, maintains its structural properties, when insertions and deletions of elements are performed, by means of rebalancing operations. When the degree of a node at level i is larger than a *split threshold*, it is split into two nodes. When its degree is less than a *fuse threshold*, it is fused with an adjacent node. The former rebalancing operation is called *split* while the latter is called *fusion*. The *sharing* operation is the combination of fusion and split. In particular, a sharing operation is a fusion followed by a split, as the fused node violates the *split threshold*.

Initially, all nodes of T constitute singleton components. The root of a singleton component is the unique node contained in it. All nodes $v \in A$ that have a child u so that $u \notin A$, constitute the *border* of the component A, which is represented by $border(A)$. It will become obvious in the sections to follow, that the nodes in the border of a component absorb all the disturbances caused by rebalancing operations in their children. In Figure 1, the structure of a component A on a tree T with root $z = A.root$ is depicted.

Components are subject to the following operations:

(1) *Break(z):* the component with handle z is invalidated and all the nodes that belonged to it become singleton components.
(2) *Add(v,z):* adds node v, which is a singleton component, in the component with handle z. It is assumed, that z is the handle of the valid component of $father(v)$.
(3) *Find(u):* returns the handle of the component in which u belongs (in Figure 1, *Find(f)* returns $A.root$).

The problem of efficiently maintaining components, is similar to the problem of efficiently maintaining disjoint sets subject to the operations of *union* and *find*. This problem is well known that cannot be implemented in worst case constant time in the Pointer Machine (see [23]). However, by exploiting the special structure of the components, a worst case constant time implementation for these operations is possible.

We represent each component by a *component record*. The component record

for component A has a pointer to $A.root$ as well as a bit called *valid*, that indicates whether A is a valid component (*valid*=TRUE) or an invalid one (*valid*=FALSE). When a component is invalid, then each node that points to the respective component record is a singleton component. Each node $u \in A$ maintains a pointer to the component record of A.

Operation *Find(u)* is implemented by traversing the pointer of u to the component record. Let A be the component represented by this component record. If A is a valid component, then a pointer to the root of A is returned, otherwise u belongs to a singleton component. In this case, a new component record for this singleton component is constructed and a pointer to u is returned. Operation *Add(v,z)* is implemented by setting the respective pointer of node v, to point to the component record of the component with handle z. Finally, operation *Break(z)* is implemented by setting the *valid* field of the respective component record to FALSE. It is clear that all operations can be performed in worst case constant time.

We will now sketch how the mechanism of components is integrated in the tree T to guide rebalancing operations. Consider a leaf l, its father f and its grandfather ff. Assume that an element is to be inserted immediately to the right of l or that l is to be deleted. Assume also, that we are provided with a finger to l. Firstly, a call to *Find(f)* is made and the root z of the component where f belongs is located. Then, this component is invalidated by a call to *Break(z)*. Finally, a rebalancing operation on z is performed, and all new nodes are added to the component of the father of z.

The mechanism of components, is a rather slow mechanism when used to schedule rebalancing operations. For example, if we consider a node x and an arbitrary node v among its children, then during any time period in which x is not rebalanced, v is rebalanced at most twice. This essential characteristic of the component mechanism is reflected in Lemma 1. Before stating the lemma, we must note that in a fusion or a sharing operation, we have to distinguish between the *active* node causing the rebalance operation and its *passive* neighbour that gets involved. In this way, Lemma 1 concerns only actively rebalanced children of a node.

Lemma 1 *Each child of a node x, will be rebalanced at most twice during any time period in which x is not rebalanced.*

Proof. First, note that only roots of components can be rebalanced. Let A be the component of x and let v be an arbitrary child of x. Assume v is rebalanced, after x was rebalanced the last time. The first time this happens, v joins the component A of x. In particular, x is not a component root and as a result it can only be rebalanced after the break of A. Since breaking A may occur at

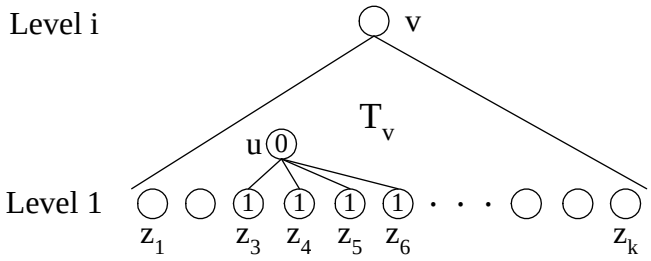

Fig. 2. Starting from z_1 we make an insertion at each node until we reach z_k in a pass. Then we start a new pass from z_1. For simplicity, the new nodes introduced by split operations are not considered. In this case, a pass has been performed and nodes z_{3-6} have their bits set to 1. In the next pass, when an insertion is performed at z_3, nodes z_{3-6} have their bits set to 0, while u's bit is set to 1.

some ancestor of x, we can rebalance v a second time without having rebalanced x. But, since the break of A made x a singleton component, x will be the root of the component of x and v after the second rebalance at v. Therefore, a third rebalance at v can only occur after x has been rebalanced. □

Lemma 1 implies that the component mechanism permits the existence of nodes with gigantic degree. In fact, the following remark shows that an aggressive version of fusion and split rebalancing operations is necessary to bound the degree of internal nodes. The following remark presents a rough estimate of the speed of the component mechanism, that is how often a node is rebalanced during update operations in its subtree.

Remark 2 *There is a sequence of update operations, so that an arbitrary node $v \in T$ at level h_v with initial weight w_v, will be rebalanced after its weight becomes $\Omega(2^{\frac{2^{h_v}}{c}} \cdot w_v)$, where c is an arbitrary parameter.*

Proof. For the sake of the proof, we attach to each internal node in T a field *bit*. This field is a single bit, indicating whether a node is a root of a component ($bit = 0$) or not ($bit = 1$). Consider a node v at level h_v with weight w_v. For a descendant z of v at level 1, the path from v to z defines a counter $c_{z,v}$ consisting of the *bit* fields for each node on this path apart from v. In this counter, the Least Significant Bit (LSB) is $z.bit$.

Initially, all internal nodes $z \in T_v$ have $z.bit = 0$. Apparently, node v is obliged to be rebalanced only when for all internal nodes $z \in T_v$, $z.bit = 1$ and $v.bit = 0$. Assume that the number of leaves of each internal node at level 1 is c. As a result, when an internal node z at level 1 is split then it will have at most $\lceil \frac{c+1}{2} \rceil$ leaves. Thus, after at most $\lceil \frac{c+1}{2} \rceil + 1$ insertions below z, z is going to be split again. We describe a scenario, where the bound stated in the remark is attained.

Insertions are performed in every node $z \in T_v$ lying at level 1 in a cyclic way. In particular, we sequentially perform one insertion after another, starting from the leftmost node to the rightmost node at level 1 as shown in Figure 2. After $\lceil \frac{c+1}{2} \rceil + 1$ insertions beneath each node, the nodes at level 1 are split. At this time, the weight of v has been doubled and all counters $c_{z,v}$ have their value increased by $\lceil \frac{c+1}{2} \rceil + 1$. The rebalancing of v will be performed when all possible counters $c_{z,v}$ have value $2^{h_v} - 1$. As a result, the weight of v will double roughly $\Omega(\frac{2^{h_v}-1}{c})$ times. Thus, the total weight of v just before rebalancing is $\Omega(2^{\frac{2^{h_v}-1}{c}} \cdot w_v)$. $\square$

4 The Virtual Tree

In this section, we describe a search tree T with worst case constant update time. We call this search structure the *virtual tree*, because in its construction, certain assumptions are made about the implementation of the rebalancing operations. In Section 5, we will see how to overcome these assumptions by simulating the virtual tree with an implementable search structure.

The skeleton tree structure is similar to an (a,b)-tree, with the difference that the lower and the upper bounds on the degree of internal nodes are not constant. In particular, the degree of an arbitrary node at level i is lower bounded by a_i and upper bounded by b_i, where a_i and b_i are functions of i. In the discussion to follow we prove that both a_i and b_i are double exponential functions of i. Additionally, the scheduling of the rebalancing operations on T is realized by the mechanism of components.

For simplicity, we will consider separately the case where only insertions are performed in the virtual tree and the case where only deletions are allowed. This separation will make the exposition of the algorithms far simpler. Finally, both solutions are combined to a constant update time virtual tree.

4.1 The Case of Insertions

In this case, the only permissible update operation is the insertion of an element. As a result, the only useful rebalancing operation is the split of an internal node. The upper bound on the degree of internal nodes lying at an arbitrary level i, is denoted as b_i. To guarantee this upper bound, the nodes of T are rebalanced by applying a *MultiSplit* operation.

The MultiSplit operation produces nodes with degree between b'_i and $4b'_i$, where b'_i is a parameter. The lower bound on the degree of nodes at level i

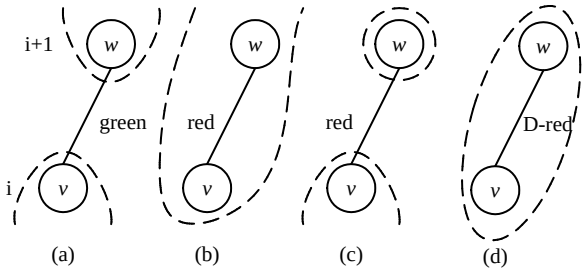

Fig. 3. The definition of colors attached to the edges.

is b'_i. This clearly holds since only insertions are allowed and the MultiSplit operation, by definition, produces nodes with degree larger or equal to b'_i. The only exception is the root of the tree that can have degree less than b'_i. As a result, a MultiSplit operation at a node at level i can create at most $r'_i = \frac{b_i}{b'_i}$ new nodes. Initially, T consists only of a single leaf storing the element $-\infty$ and the parent node of the leaf. The leaf and its parent are singleton components. For the sake of analysis, we will make the following assumption:

Assumption 3 *The MultiSplit operation can be performed in worst case constant time.*

We will estimate suitable values for the parameters b_i and b'_i based on the configuration of the nodes in the data structure with respect to components. Figure 3 depicts four possible configurations for the component relation between a node w and its arbitrary child v. Each of these configurations is represented by a color attached at the edge connecting v and w. The colors are: (a) *green*, when v is root of a component and w belongs to a non-singleton component, (b) *red*, when v and w belong to the same component and w is not the root, (c) *red*, when v and w are both roots of components and (d) *D-red* when v and w belong to the same component and w is the root.

These colors are not actually stored, but are a conceptual tool used in the analysis. They describe the potential of the child node to produce new nodes through MultiSplit operations, when its father is not rebalanced. In particular, in Figure 3 it is shown that when the edge (w, v) is green (configuration (a)) then v can be MultiSplit resulting in configuration (b). Configuration (c) is derived from configuration (b) by performing a *Break* operation on the component of w while configuration (d) is derived from configuration (c) by applying a MultiSplit to v. Configuration (c) can be also derived from configuration (a) when a break is performed on the component of w. Finally, configuration (a) follows configuration (d) when the component with handle w becomes invalid (and w is rebalanced).

Thus, a green edge may produce up to r'_i red edges, while a red edge may produce up to r'_i D-red edges. It is clear that D-red edges do not produce

Procedure Insert(l,l')
1. $f \leftarrow$ father(l)
2. InsertLeaf(l',f)
3. $z \leftarrow$ Find(f)
4. Break(z)
5. $\mathbb{Z} \leftarrow$ MultiSplit(z)
6. $y \leftarrow$ Find(father(z))
7. **for** each node $u \in \mathbb{Z}$ **do**
8. Add(u,y)

Fig. 4. The pseudocode for the *Insert(l,l′)* operation. Function *InsertLeaf(l′,f)* inserts the new leaf l' in an internal node f lying at level 1. By $\mathbb{Z}$, we represent the set of new nodes produced by the MultiSplit of z.

any new edges. These observations can be expressed by the use of a potential function Φ. This function counts the number of new edges introduced in a node at level i, due to MultiSplit operations at level $i-1$. The potential function for an arbitrary node v_i at level i is defined as follows:

$$\Phi(v_i) = r'^2_{i-1} \cdot \#\text{green} + r'_{i-1} \cdot \#\text{red} + \#\text{D-red}, \tag{1}$$

where #green denotes the number of green child edges of v_i, #red denotes the number of red child edges of v_i and #D-red denotes the number of D-red child edges of v_i.

In other words, the potential of a node represents the maximum degree that a node can get, without being rebalanced. Since we defined b_i as the upper bound in the degree of any node at level i, the following must hold:

$$\Phi(v_i) \leq b_i. \tag{2}$$

In the following, the insertion algorithm is presented and the upper bound on the degree of internal nodes is estimated.

4.1.1 The Insertion Algorithm

The procedure *Insert(l,l′)* implements the insertion operation, where l is a leaf of T and l' is the new leaf inserted next to l (recall that updates are position given). Firstly, leaf l' is inserted below node $f = father(l)$. Afterwards, the handle z of the component $A \in \mathbb{C}$ is located, in which f belongs. Assume that w is the father of z and that w belongs to the component $B \in \mathbb{C}$ with handle y (see Figure 1). Component A is invalidated by a call to *Break(z)* and then a MultiSplit operation is performed on z. The new nodes produced by the MultiSplit of z, are added to the component in which w belongs (component B). The pseudocode for this operation is depicted in Figure 4.

4.1.2 The Proof of the Upper Bound

The proof of the upper bound on the degree of internal nodes at level i, is a computation of b_i so that certain restrictions are satisfied. In particular, we will write down the restrictions that b_i and b'_i should satisfy, and then we will provide values that satisfy these restrictions. The formulation of these restrictions makes extensive use of Lemma 1.

The lower bound on the degree of nodes at level i is b'_i, with the exception of the root of T. Consider an arbitrary node v lying at level i, its children lying at level $i-1$, and the time period between two successive rebalancing operations at v. The edges connecting v and its children can be in one of four possible configurations (see Figure 3), while the potential of v is given by Equation (1). Firstly, it is imperative to prove that during this time period the potential of v is not increased. This is true, since one green edge may produce up to r'_{i-1} red edges and one red edge may produce up to r'_{i-1} D-red edges, while the only transition of edges between configurations (without causing v to rebalance) is from configuration (a) to (b), from (b) to (c), from (c) to (d) and from (a) to (c). In each of these transitions the potential, either remains the same or is reduced. As a result, the maximum degree of v, between two successive rebalancing operations, is given by its potential. The potential gets its maximum value, when v has initially $4b'_i$ children and all its corresponding child edges are green:

$$\Phi(v_i) \leq 4r'^2_{i-1} \cdot b'_i \tag{3}$$

Since values of b_i that satisfy Inequality (2) must be computed, Inequality (3) is used to compute values of b_i that satisfy the stronger condition:

$$4r'^2_{i-1} \cdot b'_i \leq b_i \tag{4}$$

Inequality (4) must hold for $i > 1$. Moreover, $b_i \geq 2$ and $b'_i \geq 2$ for $i \geq 1$, since the tree T must be at least binary. Finally, as we will justify in Section 5, there is another restriction relating b'_i and r'_{i-1}, for $i > 1$. This restriction arises from implementation details of the MultiSplit operation:

Restriction 1 $b'_i \geq r'^2_{i-1}$.

To compute suitable values for b_i and b'_i, Inequality (4) and Restriction 1 will be strengthened and given in a simpler form. As a result, using Restriction 1, Inequality (4) is strengthened as follows:

Restriction 2 $4b'^2_i \leq b_i$

Moreover, since $r'_{i-1} \leq \frac{b_{i-1}}{2}$ (recall that $r'_{i-1} = \frac{b_{i-1}}{b'_{i-1}}$), Restriction 1 is strength-

ened as follows:

Restriction 3 $b'_i \geq \frac{b^2_{i-1}}{4}$.

We will strenghten Restriction 3 by providing an upper bound to b'_i that will help in the calculations. Quantity b'_i is upper bounded by b^2_{i-1} and Restriction 3 is strengthened as follows: $\frac{b^2_{i-1}}{4} \leq b'_i \leq b^2_{i-1}$. Then, substituting in Restriction 2 the maximum value of b'_i we get:

$$4(b^2_{i-1})^2 \leq b_i \Rightarrow 4^{4^0} \cdot 4^{4^1} \cdot b^{4^2}_{i-2} \leq b_i \Rightarrow \cdots \Rightarrow 4^{4^0} \cdot 4^{4^1} \cdot \ldots \cdot 4^{4^{i-2}} \cdot b_1^{4^{i-1}} \leq b_i$$

Choosing $b_1 = 4^4$ we get:

$$\prod_{j=0}^{i} 4^{4^j} \leq b_i \Rightarrow 4^{\sum_{j=0}^{i} 4^j} \leq b_i \Rightarrow 4^{\frac{4^{i+1}-1}{3}} \leq b_i \Rightarrow 4^{\frac{4^{i+2}}{2}} \leq b_i \Rightarrow 2^{2^{2i+4}} \leq b_i \quad (5)$$

In conclusion, by choosing b'_i so that $\frac{b^2_{i-1}}{4} \leq b'_i \leq b^2_{i-1}$, any value of b_i greater than $2^{2^{2i+4}}$ is valid for our construction. Finally, the value of b'_i should be: $b'_i \geq \frac{b^2_{i-1}}{4} \geq \frac{(2^{2^{2(i-1)+4}})^2}{4} \Rightarrow b'_i \geq 2^{2^{2i+3}-2}$.

The values for b_i and b'_i must satisfy all the restrictions stated above. As a result, we conclude that when the mechanism of components is used to schedule the rebalancing operations, the degree of nodes is upper and lower bounded.

4.2 The Case of Deletions

In this subsection, the only allowable update operation is the deletion of an arbitrary element. As a result, the only useful rebalancing operations are the fusion and the sharing between adjacent nodes. In a similar manner to that of Subsection 4.1, we define lower and upper bounds on the degree of internal nodes. The lower bound on the degree of internal nodes lying at an arbitrary level i is a_i, and the upper bound is $a'_i = 4f_i$, where f_i is a chosen parameter. When the degree of nodes at level i lies in the range $[a_i, f_i]$, they are called *small* nodes, while when their degree lies in the range $[f_i, 4f_i]$, they are called *medium* nodes. Initially, all nodes in the virtual tree T are medium, except possibly the root, and each component is a singleton component.

We will use an aggressive version of the binary fusion operation for (a,b)-trees; we call this rebalancing operation *MultiFusion*. A MultiFusion, is a rebalancing operation that fuses a set S ($|S| \geq 2$) of adjacent nodes with common

father, into a single node. All components rooted at nodes participating in a MultiFusion are broken. For the sake of analysis, the following assumption is made:

Assumption 4 *Let S be an arbitrary set of adjacent nodes in T, with common father. Then, a MultiFusion on the nodes of S can be performed in worst case constant time.*

The lower bound on the degree of small nodes, is essential to prove the logarithmic bound of the search operation. The upper bound on the degree of medium nodes is also necessary, since consecutive MultiFusion operations may lead to a node with unbounded degree. In particular, a MultiFusion should generate only medium nodes. However, in some cases there are nodes that can not participate in a MultiFusion operation, because the generated node would have degree larger than a'_i. In normal (a,b)-trees, the operation of *sharing* is employed to handle these cases.

However, in our case, a general use of the sharing operation induces many problems to the structure's implementation. To overcome the need of a general sharing operation, we allow the existence of nodes with degree that may become less than a_i. These nodes are called *tiny*, while nodes for which the lower bound is valid are called *normal*. The degree of tiny nodes is at most $f_i - 1$. The operation of sharing is employed, only when it involves one tiny node with degree less than a_i. If the tiny node has degree larger than a_i, then it can not participate in a sharing operation. This is called a *restricted sharing*, since always one of the two nodes involved has degree less than a_i. The existence of tiny nodes requires the application of mechanisms, which guarantee that tiny nodes do not spoil either the linear space complexity or the time bounds for searching. This is accomplished by maintaining Invariant 1. This invariant, as we will see in 4.2.2, can be easily maintained by applying the restricted sharing and MultiFusion operations.

Invariant 1 *(a) A tiny node at level i has degree at least a_{i-1} and (b) The left sibling (if it exists) and the right sibling (if it exists) of a tiny node are normal.*

Consequently, we distinguish the following cases with respect to the function of the MultiFusion operation:

(1) The MultiFusion produces a node with degree in the range $[f_i, 4f_i]$, in which case the generated node is normal.
(2) The MultiFusion fails to produce such node. In this case, either we apply the sharing operation or (if no sharing is possible) the MultiFusion operation is completed, producing a tiny node. A sharing operation is not possible if the tiny node has degree larger than a_i.

In the following, we first provide a relation between the lower and the upper bound in the degree of normal nodes and then we show how to maintain the invariants concerning the tiny nodes. Afterwards, we describe the algorithm for the deletion operation and finally we provide sharp estimations for the parameters used in our construction.

4.2.1 Lower and Upper Bounds in the Degree of Normal Nodes

The relation between the upper and the lower bound in the degree of normal nodes is heavily based on Lemma 1. However, the property described in Lemma 1 of the component mechanism, will not be given in the form of a potential function as in the case of insertions, but it will be incorporated in the proof of Theorem 5. This theorem computes the *fusion factor*, that is, the required degree of a new node after a MultiFusion operation in order to guarantee the lower bound of a_i.

Theorem 5 *The parameters f_i and a_i must satisfy the following restriction:* $f_i \geq a_i \cdot \frac{8^{2^i-2} \cdot f_1^{2^{i-1}}}{a_1^{2^{i-1}}}$.

Proof. Assume a node v at level i and its children at level $i-1$. We say that an *epoch* starts at v, when v causes a MultiFusion operation. Lemma 1 states that the children of v will go through two epochs (multifused at most twice) before the transition of an epoch at v itself. As a result, if v initially has degree f_i, then after the transition to a new epoch, its number of children must be at least equal to a_i. When all the children of v go through an epoch, then the number of child edges of v are reduced by $r_{i-1} = 2 \cdot \frac{a'_{i-1}}{a_{i-1}}$.

By setting the maximum degree of a node at level i to be $a'_i = 4 \cdot f_i$, the child edges of v are reduced by $r_{i-1} = 8 \cdot \frac{f_{i-1}}{a_{i-1}}$. This follows from the requirement that the nodes at level $i-1$ have degree at least f_{i-1} and at most $a'_{i-1} = 4 \cdot f_{i-1}$ when they pass to a new epoch. Since by induction, the lower bound on their degree is a_{i-1}, a MultiFusion operation involves at most $\frac{8 \cdot f_{i-1}}{a_{i-1}}$ nodes to generate a new node at level $i-1$. By Lemma 1 and the above discussion, the following recurrence is derived:

$$\frac{f_i}{(8 \cdot \frac{f_{i-1}}{a_{i-1}})^2} \geq a_i$$

Solving this recurrence we get:

$$f_i \geq \frac{8^2 \cdot a_i}{a_{i-1}^2} \cdot f_{i-1}^2 \Rightarrow f_i \geq \frac{8^2 \cdot a_i}{a_{i-1}^2} \cdot \frac{8^4 \cdot a_{i-1}^2}{a_{i-2}^4} \cdot f_{i-2}^4 \Rightarrow$$

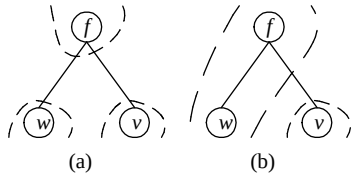

Fig. 5. The setting for the sharing operation between v and w.

$$f_i \geq \frac{8^2 \cdot a_i}{a_{i-1}^2} \cdot \prod_{j=2}^{i-1} \frac{8^{2^j} \cdot a_{i-j+1}^{2^{j-1}}}{a_{i-j}^{2^j}} \cdot f_1^{2^{i-1}} \Rightarrow$$

$$f_i \geq a_i \cdot \frac{8^{2^i-2} \cdot f_1^{2^{i-1}}}{a_1^{2^{i-1}}} \tag{6}$$

The theorem follows. □

The upper bound for internal nodes at level i is $a_i' = 4 \cdot f_i$. Theorem 5, provides a way to achieve the lower bound on the degree of nodes. As a result, a node at level i will never have degree less than a_i, when the MultiFusions create nodes with out-degree at least f_i. When a MultiFusion does not generate a node with degree at least f_i, then this node is tiny, in the sense that the lower bound does not hold for this node. Thus, tiny nodes are nodes for which the MultiFusion has failed to satisfy Theorem 5. In the following, we will discuss how to handle these nodes.

4.2.2 Maintaining Tiny Nodes

At this point, we investigate the problems generated from our decision to support *tiny* nodes. We allow the existence of these nodes, because it is difficult to implement the general operation of sharing. A restricted form of sharing is implemented, by imposing that one of the two participating nodes is tiny and that its degree is less than a_i. The restricted version of sharing is easy to implement, because (i) as Lemma 6 states, this operation does not affect the component mechanism, and (ii) the small size of the involved tiny node allows for its efficient implementation, as we will see in Section 5.

Lemma 6 *The basic property of the component mechanism given by Lemma 1 is not affected by the application of the sharing operation.*

Proof. Let v and w be the two nodes participating in the sharing operation after breaking the component rooted at v. Let G_v be the component with

handle v, and G_w be the component that contains w. The following cases for node w are distinguished with respect to component G_w (see Figure 5):

- w is the root of G_w. In this case, we break the two components, v and w are added in the component of their father and finally, a sharing operation is performed between v and w,
- w belongs to the component of its father. In this case, we break the component of v, v is added to the component of its father and finally, a sharing operation is performed between v and w.

As a result, the mechanism of components is not affected since we always break components that participate (either actively or passively) in this rebalancing operation. □

The following lemma explains how to maintain Invariant 1.

Lemma 7 *Assume that $a_i > a_{i-1}r_{i-1}^2$. By applying the rebalancing operations of MultiFusion and sharing, Invariant 1 is maintained.*

Proof. First, we prove part (a) of Invariant 1. Let v be a tiny node at level i. Let t be the last time v was rebalanced, and was found to have degree $> a_i$ but no MultiFusion or sharing was possible. After this time, when v will be rebalanced again, assume that its degree will be smaller than a_i. As a result, v will be rebalanced either by a MultiFusion or a sharing. When this rebalancing occurs, the degree of v will be at least (according to Lemma 1) equal to a_i/r_{i-1}^2. Since $a_i/r_{i-1}^2 > a_{i-1}$, part (a) of Invariant 1 follows.

We prove part (b) of Invariant 1 by contradiction. Assume a rebalancing operation at node v that violates part (b) of the invariant. In this way, assume that v becomes tiny and it has a tiny sibling w. Node v became tiny because its MultiFusion generated a node with degree less than f_i and no sharing was possible. However, the assumption that w is also tiny is impossible because v could be fused with w. Consequently, the part (b) of the invariant is proved. □

4.2.3 The Deletion Algorithm

Assume that leaf l is deleted, and assume that $f = father(l)$. Procedure *Delete(l)* implements the deletion operation. Firstly, l is removed from its father f. Then, the handle z to the component $A \in \mathbb{C}$ where f belongs is located. Assume that w is the father of z and that w belongs to the component $B \in \mathbb{C}$ with handle y (see Figure 6).

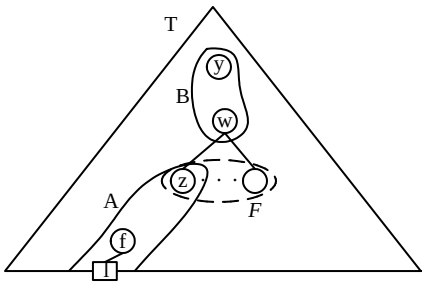

Fig. 6. The setting of the deletion pseudocode.

Procedure Delete(l)
1. $f \leftarrow$ father(l)
2. DeleteLeaf(l)
3. $z \leftarrow$ Find(f)
4. Break(z)
5. **for** each node $u \in F$ **do**
6. **if** (u is root of a component) **then** Break(u)
7. $y \leftarrow$ Find(father(z))
8. $z' \leftarrow$ MultiFusion(z)
9. Add(z',y)

Fig. 7. The pseudocode for the *Delete(l)* operation. Function *DeleteLeaf(l)* deletes leaf l. The set of nodes F, is the set of nodes that participate in the MultiFusion of z. It is assumed, that operation MultiFusion contains the mechanism for performing both MultiFusion and sharing operations.

Component A is invalidated by a call to *Break(z)*. Assume that F is the set of nodes participating in the MultiFusion operation. All components rooted at nodes $v \in F$ are broken. The new node is added to component B. The pseudocode for the deletion operation is depicted in Figure 4 (the setting for the pseudocode is depicted in Figure 6).

4.2.4 Bounding the Parameters of the Construction

The upper bound a'_i equals $4 \cdot f_i$ for a node at level i. Moreover, Theorem 5 provides a relation between f_i and the lower bound a_i in the degree of nodes.

Consider a node w at level i and its children one level below. When a MultiFusion is performed at the children of w, then a component rooted at a node among the children of w was broken. This MultiFusion operation involves at most $2 \cdot r_{i-1}$ nodes, since half of them may be tiny. Thus, if initially the degree of w is at least f_i, then, after a MultiFusion at its children, its degree may be reduced to at least $f_i - 2 \cdot r_{i-1}$. The following restriction is imposed by the implementation details given in Section 5:

Restriction 4 $a_i > 2 \cdot r_{i-1}$

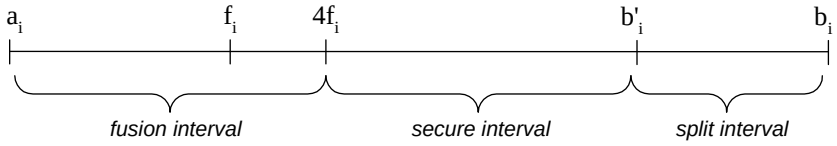

Fig. 8. The allowed degree of nodes at level i is depicted.

By Restriction 4 we get:

$$a_i > 2 \cdot r_{i-1} \Rightarrow a_i > 2 \cdot \frac{a'_{i-1}}{a_{i-1}} \Rightarrow a_i \cdot a_{i-1} > 8 \cdot f_{i-1} \tag{7}$$

In addition, the following restriction is due to Lemma 7:

Restriction 5 $a_i > a_{i-1} \cdot r_{i-1}^2$

We have to choose a value for a_i so as Theorem 5 and Restrictions 4 and 5 are satisfied.

We set a_i to be equal to $2^{2^{2i}}$ for nodes lying at level i. This choice of a_i is made, because we want to ensure a logarithmic complexity for the search procedure. In addition, we manage to be compatible with the case of insertions. As a result, $a_1 = 2^{2^2}$, while we set $f_1 = 2^{2^3}$. This choice of f_1 does not violate the lower bound for deletions, since by Lemma 1 a node at level 1 is rebalanced every second update operation at its leaves.

Thus, by substituting in Equation (6) we get:

$$f_i \geq 8^{2^{i-2}} \cdot a_i \cdot (\frac{2^{2^3}}{2^{2^2}})^{2^{i-1}} \Rightarrow f_i \geq 8^{2^{i-1}} \cdot 2^{2^{i+1}} \cdot a_i \Rightarrow$$

$$f_i \geq 8^{2^{i-1}} \cdot 2^{2^{i+1}} \cdot 2^{2^{2i}} \tag{8}$$

However, this choice of f_i must satisfy Restrictions 4 and 5. It is a matter of simple calculations to see that by choosing $f_i = 2^{2^{2i+1}}$, Restrictions 4 and 5 are satisfied.

4.3 Blending Insertions and Deletions

In this subsection, the virtual tree T is outlined, that supports insertions and deletions in worst case constant time, while retaining the upper and lower bounds on internal nodes. The description will be heavily based on the discussion made in the previous two subsections.

```
Procedure Update(l,l')
1.  f ← father(l)
2.  UpdateLeaf(l,l')
3.  z ← Find(f)
4.  Break(z)
5.  y ← Find(father(z))
6.  if (degree(z) < f_i) then
7.     for each node u ∈ F do
8.        if (u is root of a component) then Break(u)
9.     z' ← MultiFusion(z)
10.    Add(z',y)
11. else if (degree(z) ≥ b'_i) then
12.    ℤ ← MultiSplit(z)
13.    for each node u ∈ ℤ do
14.       Add(u,y)
```

Fig. 9. The pseudocode for update operations. In the case of a deletion, l' is not used. Procedure UpdateLeaf, is either a deletion of the leaf l or an insertion of a new leaf l' next to leaf l. Figures 1 and 6 depict the setting. The set of nodes F, is the set of nodes that participate in the potential MultiFusion of z. By $\mathbb{Z}$, we represent the set of new nodes generated by the MultiSplit of z.

Intuitively, we would like to let the two update procedures co-operate without having to radically change their functionality. This is achieved by combining the upper and lower bounds proved in each case. In particular, the range of degrees of nodes at level i is partitioned into *intervals* (see Figure 8). The *fusion interval*, is the range of degrees in which a MultiFusion/sharing of nodes is likely to occur. The *secure interval*, is the range of degrees in which no rebalancing operation is performed. The *split interval*, is the range of degrees in which splits of nodes are likely to occur. Tiny nodes may have degree less than a_i.

The update algorithm is given in Figure 9. Assume that an update operation at leaf l is performed. The father f of l belongs to the component with handle z. If z is in the secure interval, then no fusion with an adjacent node or split is necessary. If z is in the fusion interval, then the procedure is identical to the procedure described in 4.2.3. Similarly, if z is in the split interval, then the procedure is identical to the procedure given in 4.1.1.

5 The Simulation of the Virtual Tree

In this section the implementation of the virtual tree is given. The main goal is to remove Assumptions 3 and 4 as well as to show exactly how insertions and deletions are combined. The implementation of the virtual tree is called the *Real Tree*.

Initially, the description of the underlying structure of nodes is discussed. For the sake of clarity, the implementation of the MultiFusion (deletions only) and the MultiSplit (insertions only) will be separated. Finally, the combination of

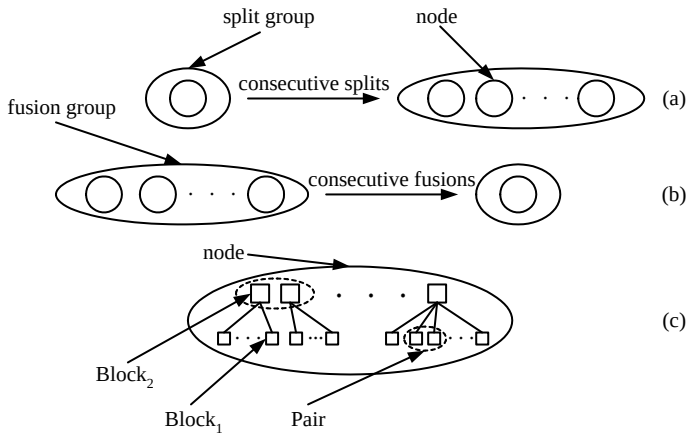

Fig. 10. The split group (a) and the fusion group (b). In (c) the structure of a node is depicted.

both implementations leads to a worst case constant update time search tree.

5.1 The Structure

MultiSplits and MultiFusions are implemented incrementally by introducing the notion of *split groups* and *fusion groups* respectively. In particular, a split group, is a set of nodes that has been incrementally constructed by consecutive binary splits, starting from a single node. Initially, split groups are singleton, meaning that they consist of a single node. All nodes inside a split group are called *nodes under construction*. On the other hand, a fusion group represents a single node generated by consecutive binary fusions of a number of adjacent nodes. As stated in 4.2.1, the number of nodes inside a fusion group varies, but it is always bounded by r_{i-1} at level i. The new node constructed incrementally in the fusion group, is called *node under construction*. In this way, a split group implements a MultiSplit operation, while a fusion group implements a MultiFusion operation (see Figure 10). In the following, when we refer to groups we mean either a fusion or a split group.

The child edges of nodes are organized into a two-level hierarchy. The lower level of this hierarchy, groups the child edges into *type 1 blocks* - represented in the sequel as *block*$_1$. The upper level of this hierarchy, groups blocks$_1$ into *type 2 blocks* - represented in the sequel as *block*$_2$ (see Figure 10). The degree of a block, is the number of its child edges, pointing to nodes one level below. A group G *is contained* in a node v, when v is the father of all nodes in G. Similarly, group G *is contained* in a block$_2$ p, when all father edges of the nodes in G, belong to p. All groups are contained in one and only one node/block$_2$.

Some of the objects described above will be structured into pairs. Generally,

a *pair of objects* consists of two objects, one of which is *full*, while the other is *underfull*. Both objects are called *mates*. An object is *full*, if its arity is equal to its predefined value, *underfull*, if its arity is smaller than this bound and *empty*, if its arity is zero. Each pair may contain at most two *pending objects*, one to the left of the pair and one to the right. These pending objects are underfull. However, when they become full from increments of their arity, they are removed from the pair. A pair *breaks*, when either both objects are full or one is empty and the other has become underfull. In the first case, two new pairs are formed by using the pending objects (if any), while in the second case the full object is paired with one of the pending objects. In a pair of objects there will always be a full object and an underfull/empty object. This is easy to achieve when pairs are subject to unary decrements and increments of their arity. If the full object's arity is decreased by 1, then a transfer is performed from the underfull object. If the underfull object becomes empty by this transfer, then the pair breaks. If the full object's arity is increased by 1, then a transfer is performed from the full object to the underfull object. If the underfull object becomes full from this transfer, then the pair breaks. In particular, the mechanism of pairs is applied to blocks_1 and on blocks_2 but the pending block mechanism is applied only to blocks_2.

As usual, the degree of a node in a tree is the number of children it has. Since we consider two different trees, we will distinguish two different notions of degree. The virtual tree is for purposes of explanation, and required the hypothetical assumptions (Assumptions 3 and 4) that MultiSplit and MultiFusion operations could be performed in worst case constant time; the degree of an object o in this tree is called the *virtual degree* and is denoted by $|o|$. The real tree is the tree actually maintained, by using incremental operations; the degree of an object o in this tree is called the *real degree* and is denoted by $||o||$. In this case, the rebalancing operations coincide with (in fact signal) the break of a component. It is easy to see, that in the case of insertions, the real degree of an object will always be at least equal to the virtual degree. On the other hand, the real degree of an object in the case of deletions, will always be at most equal to the virtual degree. In the following, when we refer to degree we imply the real degree. The virtual degree will be used mainly for the proof of upper and lower bounds.

Invariants 2 and 3 provide bounds on the degree of blocks_2 and blocks_1 respectively. Recall that the degree of a block_2, is the number of its child edges to nodes one level below. Similarly, the degree of a block_1, is the number of its child edges to nodes one level below.

Invariant 2 *The degree of a* block_2 p *at level* i *is* $||p||$*, so that* $a_i \leq ||p|| \leq b'_i + 2r'_{i-1}$.

Invariant 3 *The degree of a normal* block_1 q *at level* i *is* $||q||$*, so that* $||q|| = a_i$.

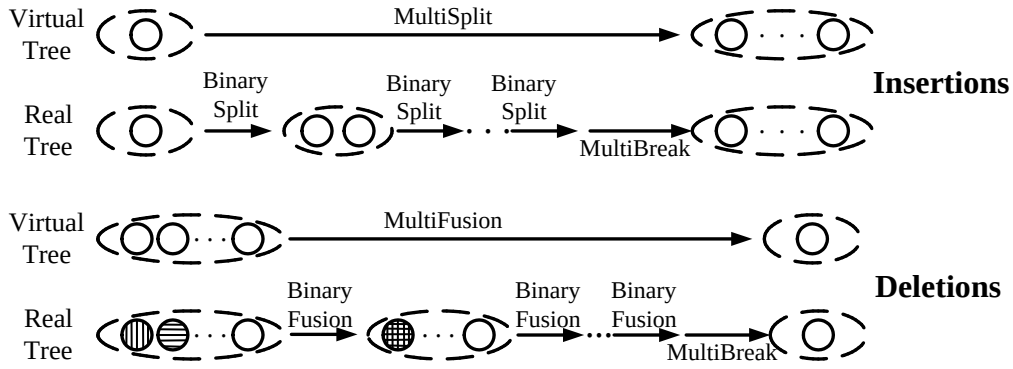

Fig. 11. The notion of the real (Real Tree) and the virtual (Virtual Tree) degree is depicted for the case of insertions and deletions. In the case of insertions, the new nodes are normal children of the nodes one level above and as a result, the real degree will always be larger or equal than the virtual degree. The equality holds exactly after performing the MultiSplit operation in the virtual tree. The case of deletions is symmetric.

Apart from these invariants it will be necessary to maintain the following two invariants. In particular, Invariant 4 states that each block_1, contains child edges that belong to one and only one node. Invariant 5 states that the nodes under construction inside a group, are contained in one and only one block_2.

Invariant 4 *The child edges of a* block_1 *belong to one and only one node.*

Invariant 5 *A group at level* i *will be fully contained in a* block_2 *at level* $i+1$.

In the following, we investigate the operations supported by each of the four objects (group, node, block_2 and block_1) as well as their representation so that all these operations are carried out in worst case constant time.

5.1.1 Representation

Before discussing the operations supported by each object it is necessary to describe their representation. Each group G is represented by a *group record* G_r that has a boolean field $G_r.valid$, indicating whether the group is *valid* or *invalid*. In the case of a split group, when G is valid ($G_r.valid = 1$), the nodes contained in G are generated by the incremental implementation of a MultiSplit (nodes under construction). When $G_r.valid = 0$, the nodes contained in G are singleton split groups. In the case of a fusion group G, when $G_r.valid = 1$, the fusion group is a set of nodes on which a MultiFusion is incrementally performed. When $G_r.valid = 0$, the generated node is a singleton fusion group. In addition, the group record contains a pointer $G_r.comp$ to a component record, as well as a pointer $G_r.block_2$ to the block_2 of the upper level in which it is contained. G has also a pointer $G_r.mate$ to an adjacent group and a pointer $G_r.incr$ to some block_1. Pointer $G_r.incr$ is used for the

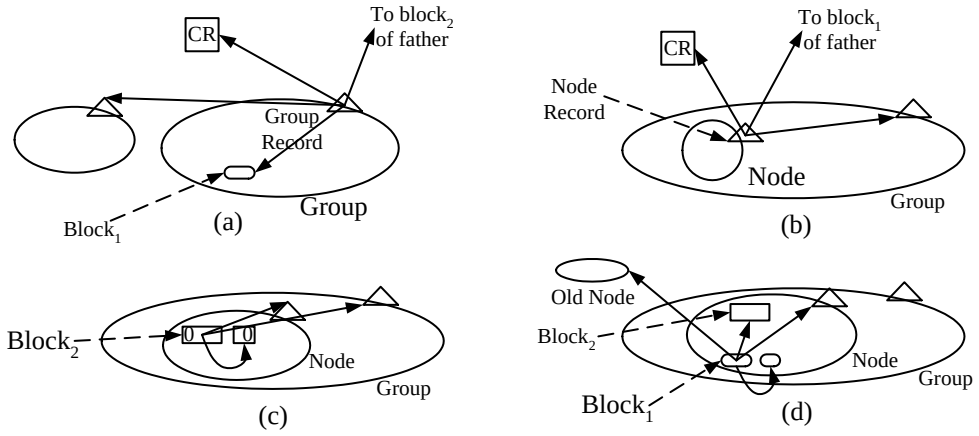

Fig. 12. The structure of a group at level i. (a) The group record, (b) the node record, (c) the block$_2$ record and (d) the block$_1$ record.

implementation of incremental transfers between groups and blocks$_2$. Finally, G contains a field $G_r.type$, that discriminates split groups ($G_r.type = 1$) from fusion groups ($G_r.type = 0$).

Each node v is represented by a node record v_r. This node record stores a pointer $v_r.group$ to the group into which it is contained, as well as a pointer $v_r.father$ to the respective block$_1$ that contains this node. In addition, it contains a pointer $v_r.comp$ to a component record and a boolean field $v_r.newnode$ that discriminates nodes ($v_r.newnode = 0$) from nodes under construction ($v_r.newnode = 1$). The nodes under construction lie inside groups and they are incrementally constructed by the fusions or the splits of many nodes. A node under construction contains at least one pair of blocks$_2$.

Each block$_2$ p is represented by a block$_2$ record p_r. The block$_2$ record contains a pointer $p_r.group$ to the group in which it belongs, as well as a pointer $p_r.mate$ to its mate block$_2$, if it belongs to a pair. In addition, it contains a bit field $p_r.pending$, indicating whether it is a pending block inside a pair ($p_r.pending = 1$) and a pointer $p_r.node$ that points to the node in which it belongs. A block$_1$ q is represented by a block$_1$ record q_r. The block$_1$ record maintains a pointer $q_r.father$ to the respective block$_2$ and a pointer $q_r.mate$ to its mate block$_1$, if it belongs to a pair. In addition, it contains two pointers $q_r.oldnode$ and $q_r.newnode$, that point to the node and to the node under construction in which the block$_1$ is contained. A block$_1$ may contain child edges that belong to one and only one node as stated in Invariant 4.

The component record cr maintains a bit field $cr.valid$, indicating whether the component is valid ($cr.valid = 1$) or not ($cr.valid = 0$). It also maintains a pointer $cr.root$ to the root of the component. The representation of the various objects is depicted in Figure 12.

In this representation, some pointers are used only for deletions and some pointers only for insertions, while most of them are used in both cases. In addition, each object has a pointer $left$ to its left adjacent object and a pointer *right* to its right adjacent object, as well as a field that stores its degree. These fields are maintained easily and they will not be considered again. In the discussion to follow, the functionality of each pointer will be clarified.

5.1.2 Operations

The four objects described in 5.1.1 are subject to a set of operations. These operations will be used extensively in Subsections 5.2 and 5.3. As a result, their implementation is crucial for the comprehension of the next subsections.

Assume that G and G' are adjacent groups at level i and that v is a node. Groups support the following operations:

(1) *CreateGroup(G)*: It creates a new group by constructing a new group record.
(2) *GAdd(v,G)*: It adds node v to the group G by updating appropriately the record v_r. Node v is a node under construction ($v_r.newnode = 1$).
(3) *MultiBreak(G)*: It breaks the group G into singleton group(s) by setting $G_r.valid = 0$. In the case of a split group, $G_r.comp.valid$ is set to 0 (the component rooted at G is broken). In the case of a fusion group, the efficient implementation of MultiBreak, requires the simultaneous break of all components rooted at nodes inside this fusion group in worst case constant time. This operation is applied also to the mate of G, if there is one. The new singleton group(s) are added to the component in which their father belongs, by making $G_r.comp$ point to the respective component record. In the case of split groups, the new singleton groups will have their fields updated in a lazy manner when a rebalancing operation is applied to them. In a lazy manner, the field *type* is set to 1, meaning that the singleton group is considered a split group. The unique node inside the singleton split groups, is a node under construction. In the case of fusion groups, the new node v points immediately to the respective component record. The field *type* of the new singleton group, produced by the fusion group, is set to 0. This operation simulates the MultiSplit and MultiFusion operations with respect to the component mechanism.
(4) *GFuse(G,G')*: It fuses groups G and G'. Assume without loss of generality, that G is the small fusion group (it has degree less than f_i at level i). This fusion is achieved by transferring the block$_2$ p of G to G' in worst case constant time. Additionally, $G_r.mate$ points to G' and $G'_r.mate$ points to G. After the transfer of p to G', the blocks$_1$ of p and the respective nodes inside G are incrementally updated. This is realized by setting $G'_r.incr$ to the leftmost block$_1$ of G (if G is to the right of G') or to the

rightmost block$_1$ of G (if G is to the left of G'). More specifically, fields *oldnode* and *newnode* of blocks$_1$ and field *group* of nodes are updated incrementally. Group G' will not participate in a new GFuse operation as long as this incremental transfer continues. When the transfer ends, G is discarded and $G'_r.mate$ as well as $G'_r.incr$ are set to NULL. When GFuse involves a normal singleton group G' and a small group G, the node under construction will be the singleton node inside G'. In this way, there is no need to update the fields of blocks$_1$ inside the only node of G'.

(5) *GShare(G,G')*: Assuming that G is the small group, this operation transfers child edges from group G' to fusion group G. This sharing step is implemented by moving a block$_2$ p' from G' to G. After this step, both G and G' have degree at least $2 \cdot (f_i - r_{i-1})$. Since G' is also rebalanced by this operation, a MultiBreak is first applied to G'. Note that a restricted version of sharing is implemented, in the sense that G always consists of a single block$_1$ q. As a result, the sharing operation is performed in worst case constant time by moving the block$_2$ p' from G' to G and by inserting q to p'. As in GFuse, the block$_2$ p' will be transferred from G' to G incrementally, in order to update the corresponding pointer fields of blocks$_1$ and nodes. When the transfer ends, pointers $G_r.mate$ and $G'_r.mate$ are set to NULL.

A node w may be split into two nodes by a call to operation *Split(w)*. This operation creates a new node w' next to w and adds it to the split group of w. It is assumed that w has at least two pairs of blocks$_2$, otherwise this operation does not perform a split. The new node w' is formed by moving a block$_2$ from w to w' and by calling *GAdd(w',$w.group$)*.

There are two operations supported by blocks$_1$ and blocks$_2$ (we will refer to them generally as blocks): *Add(e,e')* and *Remove(e)*, where e and e' are child edges of blocks. The first operation adds a new child pointer e' next to the child pointer e in the block, while the second operation removes a child pointer e from a block. Assume that the child edge e belongs to block$_1$ q and let q' be its adjacent block$_1$. Assume also that q and q' belong to the block$_2$ p and let p' be its adjacent block$_2$. Without loss of generality assume that q and p are the full blocks, if they belong to a pair.

The operation *Add(e,e')* first adds a new child edge e' next to child edge e. Since the degree of the block containing e has increased by 1, it is necessary to check whether Invariants 2 and 3 are maintained. In this way, if q belongs to a pair and q' is its mate, then a transfer of one child edge from q to q' takes place. If q' becomes a full block$_1$ (recall Invariant 3), then the pair is broken and two new pairs are formed. When a pair breaks, the two full blocks are seperated (by changing the respective mate pointers) and constitute new pairs. Their mates in the new pairs are empty blocks. The same procedure applies

to the block$_2$ p with a small difference, since Invariant 5 must be maintained. In particular, assume that p and p' form a pair and become full after the addition of e'. In this case, this pair should be broken. However, this break is not realized if there is a child group, that its nodes are spanned by both p and p'. More specifically, there are nodes v and u inside this group so that v belongs to p and u belongs to p'. If there is such group, then the break of the pair is postponed until this group has been moved completely to one of the two blocks$_2$. This transfer is the reason for the additive factor of $2r'_{i-1}$ in the upper bounds of the size of blocks$_2$ (Invariant 2).

The operation *Remove(e)*, removes the edge e from a block. Similarly to operation Add, Invariants 2 and 3 may be violated. As a result, structural changes are necessary to re-establish these invariants. If q and q' are in the same pair, then a child edge is moved from q' to q, otherwise no transfer is made. However, if q did not belong to a pair, then its adjacent block$_1$ q' is considered. If q' does not belong to a pair then a new pair is formed containing q and q'. If q' already belongs to a pair with block$_1$ q'', a child edge is transferred from q' to q (a sharing step). As a result, a Remove operation is applied to q'. Since q' already belongs to a pair, the operation terminates after a transfer between q' and q''. The same procedure applies to blocks$_2$ p and p', but similarly to operation Add we must maintain Invariant 5. In particular, the maintenance of Invariant 5 presupposes that no sharing between blocks$_2$ takes place. The need of sharing is removed by using pending blocks$_2$. If p' belongs to a pair, but it is a pending block$_2$ of this pair, then p' is removed and forms a new pair with p. If p' is not a pending block$_2$, then p is set to be a pending block$_2$ of the pair containing p'. These pending blocks may violate Invariants 2 and 3. However, they are part of this pair as long as they violate these invariants. Finally, when transfers of blocks$_1$ between blocks$_2$ take place, the respective fields *oldnode* and *newnode* of the involved blocks$_1$ are updated accordingly.

Operations *InsertLeaf(l,f)* and *DeleteLeaf(l)* are used for the insertion and deletion of a leaf respectively. The former operation inserts a new leaf l at node f while the latter deletes leaf l. Both are implemented by the block operations Add and Remove respectively.

5.2 Implementing Insertions

In this subsection, the existence of Assumption 3 is justified by providing an implementation of MultiSplit in worst case constant time. This will be achieved by using multiple levels of indirection and incremental scheduling techniques. All incremental operations, precede the call to the MultiSplit operation. As a result, when MultiSplit is performed the new nodes are already constructed.

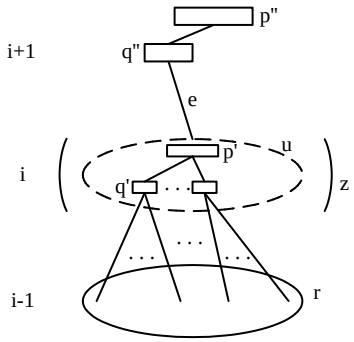

Fig. 13. The setting for the insertion algorithm.

In particular, the MultiSplit operation is implemented in worst case constant time by using split groups. Initially, each split group consists of only one node. The split group is the means of implementing incrementally a MultiSplit operation by performing ordinary binary splits. As a result, a split group may contain many nodes under construction, generated by these binary splits. A final detail concerns the relation between components and split groups. Since all nodes inside a split group originate from a single node, the component mechanism is applied to the split groups and not on each node under construction separately. The effect of the MultiSplit operation on the component mechanism is simulated by the MultiBreak operation.

In the following, the algorithm for the insertion of a leaf is given. This algorithm is heavily based on the operations described in 5.1.2. Finally, we prove that the rebalancing algorithm is correct.

5.2.1 The Insertion Algorithm

The algorithm for the insertion of a leaf l' is depicted in Figure 14. Assume that the new leaf l' is inserted immediately to the right of l. The split group at level 1 that contains leaf l is f and f belongs to the component with root the split group r at level $i-1$ ($i \geq 2$). The split group r, is the child of the split group z at level i. The nodes under construction of r, are contained in node u, in block$_2$ p' and a subset of its nodes in block$_1$ q'. Finally, some nodes of z are contained in block$_1$ q'' at level $i+1$ and all of them are contained in block$_2$ p'' (recall Invariant 5). This setting is visualized in Figure 13 while the pseudocode for the algorithm is depicted in Figure 14.

After inserting leaf l' to group f, the split group r is located. The group r, is the root of the component in which f belongs. This is achieved by the operation *Find(f)*. In particular, if $f.valid = 1$ and $f.comp.valid = 1$, then the root of the component in which f belongs, is $f.comp.root$. If $f.valid = 1$ and $f.comp.valid = 0$, then a new component record is created for the singleton component containing f. In this case, operation Find returns f. If $f.valid = 0$,

Procedure Insert(l,l')
1. $f \leftarrow$ father(l)
2. InsertLeaf(l',f)
3. $r \leftarrow$ Find(f)
4. MultiBreak(r)
5. $u' \leftarrow$ Split(u)
6. Add(e,e')

Fig. 14. The algorithm of the *insertion* operation. The edge e' is the edge between q'' and the new node u' and e is the edge between q'' and u. Operation *Add(e,e')* is responsible for the maintenance of block$_1$ q'' and block$_2$ p'' as shown in 5.1.2.

then f is an invalid group. In this case, the node f' containing leaf l becomes a singleton split group and constitutes a singleton component. As a result, operation Find returns f'. In this way, the root of the component is always located in worst case constant time.

Subsequently, the MultiBreak operation invalidates the component with handle r. Then, it adds all new nodes to the component in which their father group z belongs. This MultiBreak at level $i-1$ introduces at level i at most r'_{i-1} new child edges. The new nodes have already been inserted in z but the MultiBreak of r, signals the rebalancing operations that must be performed as a result of these new nodes. The node u at level i, which completely contains the new nodes produced by the MultiSplit of r, will be split and produce a new node u' by transferring its rightmost pair of blocks$_2$ to u'. If u consists of only one pair of blocks$_2$, then the split operation is not performed. Since all nodes inside a split group belong to the same component, this binary split does not affect the component mechanism. The binary split of nodes under construction must be reflected at level $i+1$ by adding a new child edge. This child edge is inserted at the respective block$_1$ and block$_2$ of level $i+1$.

From the description of the insertion procedure, it is clear that Invariants 2 and 3 hold since the degree of blocks$_1$ and blocks$_2$ is changed by at most a constant number of edges. Invariant 5 is maintained, since all operations used in the algorithm maintain this invariant as discussed in 5.1.2. In a nutshell, the algorithm for insertions consists of a MultiBreak at level $i-1$, a binary split at level i (incremental step of a MultiSplit) and the addition of a new edge at the blocks at level $i+1$. At level $i+1$, there are no new nodes and as a result the procedure terminates. In the following, we elaborate on the proof given in Subsection 4.1.2 in the light of the incremental implementation of MultiSplit.

5.2.2 Analysis

The analysis presented at this point, will be a refinement of the analysis given in 4.1.2. We elaborate on the arguments presented in 4.1.2, in the light of the incremental implementation of the MultiSplit, and show that none of them

changes. In addition, some points of the proof will be clarified (like Restrictions 1 and 2). In the following, when we refer to blocks$_2$ it will be clear from the context whether we refer to a single block$_2$ or to a pair of blocks$_2$.

A node v inside a group always contains at least one pair of blocks$_2$, since v is split when it contains at least two pairs of blocks$_2$. By Invariant 2 and this simple observation we conclude, that the real degree of a node at level i is lower bounded by a_i. However, since we consider the case of insertions only, we can assume that the size of blocks$_2$ is lower bounded by b'_i as stated in the virtual tree (Subsection 4.1).

When a MultiBreak is applied to split group r at level $i-1$, at most r'_{i-1} new nodes are added to the father node u at level i. However, as a result of this MultiBreak, u is split by transferring to a new node under construction a pair of blocks$_2$. This split operation on u removes at least b'_i child edges, as a result of the addition of at most r'_{i-1} child edges. The new r'_{i-1} child edges may be red edges (so they are generated from a green edge) or D-red edges. In the worst case, r'_{i-1} red edges may be added and b'_i D-red edges may be removed. As a result, r'^2_{i-1} potential is added and b'_i potential is removed. To guarantee that the potential does not increase, Restriction 1 must be valid. Restriction 2 stems from the fact that each pair of blocks$_2$ may consist of up to four blocks$_2$, three of which will have degree less than b'_i due to the existence of pending blocks. Consequently, the degree of a singleton split group at level i is at most $4 \cdot b'_i$, leading to Restriction 2.

The proof of the upper bound is heavily based on the following observation: when a MultiSplit is performed, the new singleton group has degree less than $4 \cdot b'_i$. In the incremental implementation, it may be the case that a singleton split group has degree much larger than $4 \cdot b'_i$. This observation follows from the fact that all nodes under construction are visible by child edges attached to blocks one level above. Recall that in the virtual tree, the addition of these child edges was performed during the MultiSplit operation. However, in the simulation of the MultiSplit operation, the nodes under construction are added as soon as they are generated. Consequently, the upper bound is not valid. However, by considering the relation between the virtual and the real degree, it is shown that although this case is possible the upper bound is not violated.

Assume a new singleton split group G generated by a MultiBreak operation. If the degree of G is larger than $4 \cdot b'_i$, then the additional degree corresponds to children nodes, which are nodes under construction inside split groups. Namely, the virtual degree of the unique node v in G is still $\leq 4 \cdot b'_i$. Assume that the virtual degree of v is $> 4 \cdot b'_i$. Then at some point of time, a MultiBreak operation was performed at a child group of v and v was not split. If v was not split, then v should have real degree $\leq 4 \cdot b'_i$. If v had real degree $> 4 \cdot b'_i$, then it would have been split. Since the virtual degree is always less than the

real degree, it follows that v always has virtual degree $\leq 4 \cdot b'_i$, meaning that it has at most $\leq 4 \cdot b'_i$ child groups. As a result, the upper bound is maintained.

Concluding, the incremental implementation of the MultiSplit operation does not affect the proof of the upper bound given in 4.1.2. As a result, considering the analysis made for the virtual tree for parameters b'_i and b_i, we derive the following theorem matching the bounds given in [6]:

Theorem 8 *Assuming that finger searches can be implemented efficiently, we can maintain a finger search tree with worst case constant insertion time, when deletions are not allowed.*

5.3 Implementing Deletions

In this subsection, the existence of Assumption 4 is justified by providing an implementation of the MultiFusion operation in worst case constant time. Similarly to insertions, all incremental operations precede the execution of the MultiFusion operation.

In particular, the MultiFusion operation is implemented in worst case constant time by using fusion groups. Initially, each fusion group consists of only one node. The fusion group is the means of implementing incrementally a MultiFusion operation by performing ordinary binary fusions. The effect of the MultiSplit operation on the component mechanism is simulated by the MultiBreak operation.

In the following, we discuss the fusion groups and certain implications of their use, we give the algorithm for the deletions and finally we prove that the structure is correct.

5.3.1 The Fusion Group

A fusion group consists of a set of nodes, from which a new node will be generated. Each fusion group generates one and only one new node. This node may contain at most two pairs of blocks_2. As a result, the real degree of the fusion group may be at most equal to $4 \cdot f_i$.

Since a fusion group contains a set of distinct nodes S, it is necessary to maintain two distinct representations of the fusion group. In particular, each node in S, may be a root of a component or belong to the component of its father, irrespectively of the component of its adjacent nodes. Note, that a similar problem does not exist in the split groups, since all nodes belong to the same component as long as the split group is valid. As a result, this representa-

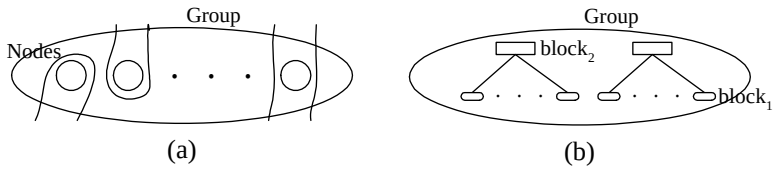

Fig. 15. The two conceptual representations of a fusion group. (a) the nodes inside the fusion group maintain their components and (b) the blocks_2 of the new node are incrementally constructed.

tion of a fusion group, called the *component representation*, is related to the maintenance of the components of its nodes.

The second representation, called the *structural representation*, is related to the incremental fusion of all nodes inside the fusion group. All child edges of the fusion group are stored in at most two pairs of blocks_2, and these child edges are further structured into blocks_1. The incremental implementation of the MultiFusion operation presupposes that both representations of fusion groups are maintained. In Figure 15, the component and the structural representations are depicted.

Considering the component representation, a fusion group G ($G_r.valid = 1$), contains nodes that are roots of components, just before it becomes invalid (just before a MultiBreak is applied to G). As a result, when a MultiBreak is performed on this fusion group, all components rooted at the nodes inside G are rendered invalid. Finally, the new node constructed by the fusion group, is added to the component into which its father node belongs. It is assumed that the new node is generated by the fusion of nodes with common father, as stated in Invariant 6. Assuming that Invariant 6 does not hold, then the component mechanism fails, since it is not clear in which component the new node should be added.

Invariant 6 *The nodes inside a group at level i, are children of one and only one node at level $i + 1$.*

Since the incremental construction of the new node precedes the execution of the MultiBreak, the fusion groups use a structural representation. If the incremental construction of the new node started immediately after the MultiBreak, then the fusion groups would use a component representation. In the current scheme, there is no choice but to simulate the component representation by the structural representation. This is achieved by using the block_1 fields *oldnode* and *newnode*. In fact, the discussion to follow is the implementation of the *Find* operation, in the case of deletions.

In particular, the pointers *oldnode* and *newnode* should point to nodes and nodes under construction respectively. Assume that the leaf l is removed from

node f inside the fusion group g at level 1. In addition, let q be the block$_1$ that contained l and let A be the component in which f is contained. The goal is to locate A, in worst case constant time by using these pointers. The main problem is that node f is not easily found. Assume that f is located. In this case, if $f.comp.valid = 1$, then A is $f.comp$, otherwise A is the singleton component with root f. Below, we show how to locate node f.

Initially, node $v = q_r.oldnode$ is considered. If $v_r.group.valid = 1$ then f is v. If $v_r.group.valid = 0$, then node v does not exist since it participated in a MultiFusion operation. In this case, node f is $q_r.newnode$. The following invariant guarantees that $q_r.newnode.group.valid = 1$:

Invariant 7 *For an arbitrary block$_1$ q, one of the pointers $q_r.oldnode.group$ and $q_r.newnode.group$ points to a valid group.*

Finally, for reasons of compatibility with the discussion made in Subsection 4.2, the upper bound on the degree of blocks$_2$ will be $f_i + 2r_{i-1}$. In Subsection 5.4 it will become clear how insertions and deletions are combined, so that the size of blocks$_2$ is governed by Invariant 2.

5.3.2 The Algorithm for Deletions

The algorithm for deletions is given in Figure 17. Assume that the leaf l is removed from the tree structure. The node at level 1 that contains leaf l is f. Assume also that f belongs to a component with handle a node v, in the fusion group r, at level $i-1$ ($i \geq 2$). The fusion group r is contained in the block$_1$ q_z at level i, since it contains only one node under construction. Block$_1$ q_z points to node u contained in block$_2$ p_z and in fusion group z. Let the adjacent fusion groups of z be w and y. Finally w, z and y are children of the fusion group x. This setting is visualized in Figure 16, while the pseudocode for the operation is depicted in Figure 17.

Initially, the split group r is located, containing the root of the component in which f belongs. Subsequently, a MultiBreak operation is applied to r. Assume that the MultiBreak results in the construction of a singleton fusion group r'' that consists of a node with a single block$_1$ (r was tiny - Line 5). In this case, a *GFuse* or a *GShare* operation is performed (Lines 6-10) with the adjacent fusion group r'. Note, that this share or fuse is of a restricted form, since r'' consists of a single block$_1$. As a result, both operations can be carried out in worst case constant time. Observe that when a *GShare(r'',r')* operation is executed, then $||r''|| + ||r'|| \geq 4 \cdot f_{i-1}$, meaning that r' consists of two pairs of blocks$_2$.

The MultiBreak of r introduces a new node to block$_2$ p_z. If the degree of z is less than f_i (Line 11), then it is fused with y, given that the degree of both

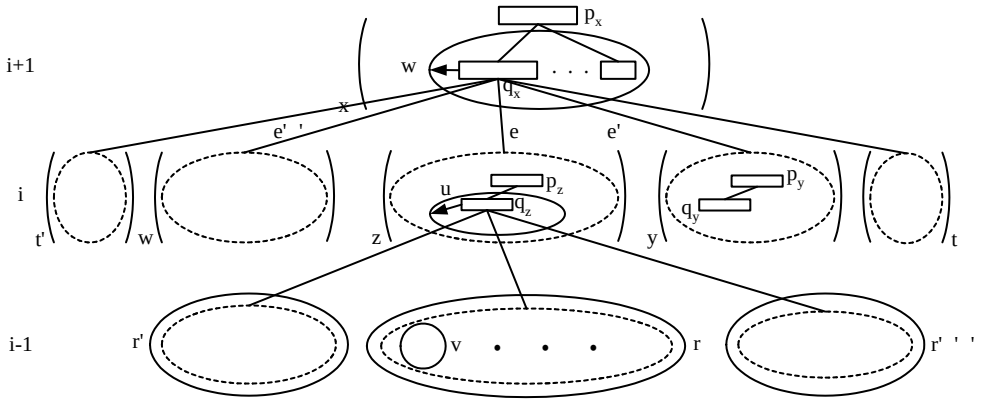

Fig. 16. Fusion group r contains node v, which is the root of the component containing the father f of the deleted leaf l. The solid ellipses represent nodes, while the dotted ellipses represent nodes under construction.

groups is less than $4 \cdot f_i$ (Line 12). However, if y has unfinished incremental transfers (recall operation GFuse or GShare), then z is not fused with y. It is easy to check whether a group G is in the middle of an incremental transfer, by checking if the pointer $G_r.mate$ has a non-NULL value. Although this fact seems to lead to fusion groups with small degree, the virtual degree of normal groups is always within bounds. Particularly, the virtual degree of y will surely be larger than f_i since at each MultiFusion of its children, at least two blocks_1 are transferred from its mate. In this case, z may be handled as a tiny group.

Assuming that z was a normal fusion group, then w and y may be tiny fusion groups. As a result, z cannot become tiny. However, if this is the case, then z will be fused with at least one of its adjacent fusion groups, leading to a normal fusion group (Lines 13-19). Consequently, Invariant 1 is maintained. If z was tiny, then its adjacent nodes will be normal and Invariant 1 is still maintained.

Furthermore, Invariant 6 is maintained, since a child edge corresponds to one and only one fusion group. Invariant 4 is also maintained, since blocks_1 are fused with other blocks_1 that belong to the same node. If there is no such block_1 available, then this block becomes tiny. The brothers of tiny blocks_1 are always normal. This is a consequence of the fact that block_1 q is tiny whenever the corresponding node $q_r.oldnode$ is also tiny. Since Invariant 1 is maintained for nodes, the same holds for blocks_1. In a lazy manner, when the fusion group breaks, all tiny blocks_1 are fused with their brothers (Lines 31-33). The sizes of the blocks are also maintained, with the exception of blocks_1 and blocks_2 contained in a tiny group.

Another important detail is the maintenance of Invariant 7. Invariant 7 states that the pointers *oldnode* and *newnode* maintained in the blocks_1, must be updated or at least they can be updated in worst case constant time. These

```
Procedure Delete(l)
1.  f ← father(l)
2.  DeleteLeaf(l)
3.  r ← Find(f)
4.  r'' ← MultiBreak(r)
5.  if (r'' has only one block_1) then
6.    if (||r''|| + ||r'|| ≤ 4 · f_i) then
7.      GFuse(r'',r')
8.    else
9.      r'_n ← MultiBreak(r')
10.     GShare(r'',r'_n)
11. if (||z|| < f_i) then
12.   if ((||z|| + ||y|| < 4f_i) AND (y_r.mate = NULL)) then
13.     temp ← ||y||
14.     y ← GFuse(z,y)
15.     Remove(e)
16.     if ((||y|| < f_i) AND (temp < a_i)) then
17.       if ((||y|| + ||w|| < 4f_i) AND (w_r.mate = NULL)) then
18.         w ← GFuse(y,w)
19.         Remove(e')
20. elseif (z_r.mate ≠ NULL) then
21.   g ← z_r.mate
22.   if (z_r.incr ≠ NULL) then
23.     if (z_r.incr.oldnode.group.valid = 0) then
24.       z_r.incr.oldnode ← z_r.incr.newnode
25.     z_r.incr.oldnode.group ← z
26.     z_r.incr.newnode ← z_r.incr.father.node
27.     if (z_r.incr.left.father.group = z_r.incr.father.group) then
28.       z_r.incr = z_r.incr.left
29.     else
30.       z_r.incr = null
31. if ((||q_x|| < a_i) AND (q_x.mate = NULL) AND
          ((q_x.oldnode = q_x.left.oldnode) OR (q_x.newnode = q_x.left.oldnode))) then
32.   Update q_x.oldnode and q_x.newnode as Lines 21-25
33.   Make a pair if possible between q_x and one of q_x.left or q_x.right
```

Fig. 17. The pseudocode for the *delete* operation. The setting is depicted in Figure 16. Certain variables (like z and y) are not set explicitly in the pseudocode for space reasons, but their initialization can performed in worst case constant time by following the proper pointers. The singleton fusion group r'' is the new group constructed by the MultiBreak of r. Symmetric cases have been omitted for space reasons. In particular, lines 5-9 should be repeated for the right adjacent group r'''. Lines 11-18 should be repeated to consider groups t and t'. Finally, in lines 20-25 the symmetric case, where g instead of z has incremental transfers, while in lines 26-27 the case where the blocks$_1$ to be transferred are to the right of $z_r.incr$.

pointers are updated as blocks$_1$ are moved from the one block$_2$ to the other in the same pair (Lines 20-30). In this way, between p_z and its mate, blocks$_1$ are moved while updating the corresponding pointers. This is incorporated in the definition of the GFuse and the GShare operations. Since a group involved in an incremental transfer does not participate in other GFuse or GShare operations, Invariant 7 is maintained.

5.3.3 Analysis

Invariants 3, 6 and 7 are proved to be maintained during the algorithm depicted in Figure 17. At this point, we will partly repeat the analysis of 4.2.4

and refine it in the light of the incremental constructions.

The easy part is to prove the upper bound of $a'_i = 4 \cdot f_i$, for a node at level i. By construction, a new node consists of at most two pairs of blocks$_2$. As a result, the upper bound is always $a'_i = 4 \cdot f_i$. This upper bound is maintained by simply not allowing fusions that violate it. This is expressed by Invariant 1.

Assume a node w at level i and its children one level below. When a MultiBreak is performed at the children of w, a component rooted at a node among the children of w is broken. As a result, the fusion group generated a new node. The maximum number of nodes inside a fusion group is $2 \cdot r_{i-1}$. This is a direct consequence of Lemma 1 and Invariant 1. In fact, the multiplicative factor of 2 is introduced by the existence of tiny nodes. Thus, if initially the degree of w is at least f_i, then after a MultiFusion at its children, its degree may be reduced to at least $f_i - 2 \cdot r_{i-1}$. The result of this MultiFusion is that a GFuse operation may take place at the level of w between two fusion groups G and G'.

These fusions, result in the transfer between blocks$_2$ of at least one block$_1$, with degree a_i. If G' is tiny, then it is necessary to perform another GFuse operation at the level of w, so that a block$_1$ with degree a_i is moved. To counterbalance this loss of degree due to the MultiBreak operation, the following restriction must be satisfied:

Restriction 6 $a_i > 2 \cdot r_{i-1}$

By imposing Restriction 6, the new nodes will always have virtual degree larger than f_i. In addition, this restriction ensures that the real degree of a normal group will not become less than a_i, since for each MultiFusion, a GFuse operation adds at least the same degree to the group. The same argument also holds for the algorithm depicted in Figure 17, concerning the sizes of the groups, due to the fact that no fusion is performed when incremental transfers are pending.

This argument is based on the fact that the initial virtual degree of w was f_i. By using Theorem 5, its virtual degree will not decrease below a_i. This is true if and only if Theorem 5 is maintained when fusions between nodes are performed. However, the proof of this theorem did not make any assumptions about the implementation of fusions. In addition, the mechanism of components is not affected by any of these operations among the nodes at level $i-1$, as we saw in this subsection.

From the discussion above and the analysis of 4.2.4, we complete the analysis for the case of deletions. The following theorem summarizes the result of this subsection.

Theorem 9 *Assuming that finger searches can be implemented efficiently, we can maintain a finger search tree with worst case constant deletion time, when insertions are not allowed.*

5.4 Constant Update Time Search Trees

In this subsection, the algorithms for the two distinct cases of insertions and deletions are merged. Our approach will be to keep both algorithms as independent as possible. In this way, each group can be in one of the following three states; (a) it behaves as a fusion group, (b) it behaves as a split group and (c) no rebalancing takes place when it lies somewhere between those extreme settings. In the following, we elaborate on the update algorithm.

5.4.1 The Update Operation

We describe a tree structure that supports insertions and deletions in worst case constant time, while retaining the upper and lower bounds on internal nodes. The description will be heavily based on the discussion made in the previous two subsections. For clarity, all invariants that have been imposed on the tree structure are given below:

(1) The degree of a block$_2$ p at level i is $||p||$, so that $a_i \leq ||p|| \leq b'_i + 2r'_{i-1}$. (Invariant 2)
(2) The degree of a block$_1$ q at level i is $||q||$, so that $||q|| = a_i$. (Invariant 3)
(3) The child edges stored in a block$_1$ belong to one and only one node (Invariant 4)
(4) A group at level i will be fully contained in a block$_2$ at level $i + 1$. (Invariant 5)
(5) The nodes inside a group at level i are children of one and only one node at level $i + 1$. (Invariant 6)
(6) For a block$_1$ q, one of the pointers $q_r.oldnode.group$ and $q_r.newnode.group$ points to a valid group. (Invariant 7)

We will prove, by combining the results from the previous sections, that the minimum degree of a normal group is a_i, while the maximum degree of all groups is b_i.

Groups are discriminated in split groups and fusion groups by using the field *type*. Recall, that the fusion interval, is the range of degrees in which a MultiFusion/sharing of nodes is likely to occur. The secure interval, is the range of degrees in which no rebalancing operation is performed. The split interval, is the range of degrees in which splits of nodes are likely to occur. In the secure interval, incremental operations are performed in order to realize

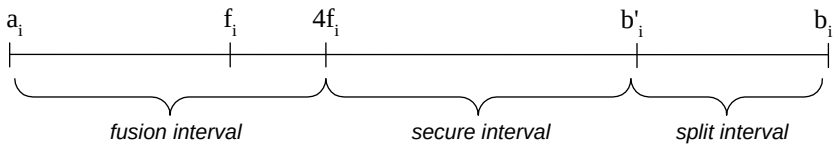

Fig. 18. The allowed degree of normal objects is depicted.

the transition between fusion and split groups. In Figure 18, the intervals are depicted.

The update algorithm is given in Figure 20. Assume that we perform an update operation at leaf l. The father f of l belongs to the component A rooted at a node inside group r. The implementation of procedure *Find(f)*, is the combination of the procedures described in 5.2.1 and 5.3.1.

In particular, the hard part is to designate node f. If node f is designated, then A is trivial to find. If f belongs to a split group, then it is a node under construction. As a result, A is designated by the field $f.group.comp$, as in the case of insertions. If f belongs to a fusion group, then f is not a node under construction. As a result, A is designated by $f.comp$, as in the case of deletions. Consequently, if we can discriminate these two cases, then A is easily found. This is easy to accomplish by using the field *type* of the valid group in which q lies. This field is updated, by the operation MultiBreak as well as in the algorithm in Lines 3-4 and 29-30.

Initially, a MultiBreak is performed on r, adding the new singleton group(s) in the component of their father node. Note, that a sharing operation is necessary as in the case of deletions (Lines 1-9).

If z is in the secure interval, then it does not participate in any rebalancing operation. However, the node under construction u' may have changed in size. In Subsection 5.2, the assumption was made that the lower bound in the size of blocks$_2$ is b'_i. In Subsection 5.3, the assumption was made that the upper bound in the size of blocks$_2$ is $f_i + 2r_{i-1}$. These bounds must be combined, so that the bounds given by Invariant 2 are valid. In particular, when a group lies in the fusion interval, the upper bound of the blocks$_2$ is $f_i + 2r_{i-1}$, while when it lies in the split interval, its lower bound is b'_i. It is in the secure interval, that this transition between the different bounds in the degree of blocks$_2$ take place.

If the size of u' is less than f_i, then its unique block$_2$ is moved to an adjacent node under construction. There is surely at least one such node, since we assumed that z is in the secure interval. Then, a fusion or a sharing operation between blocks$_2$ is performed by using the operations Add and Remove. If the size of u' is larger than $8 \cdot f_i$ and contains many blocks$_2$, then the fusion of

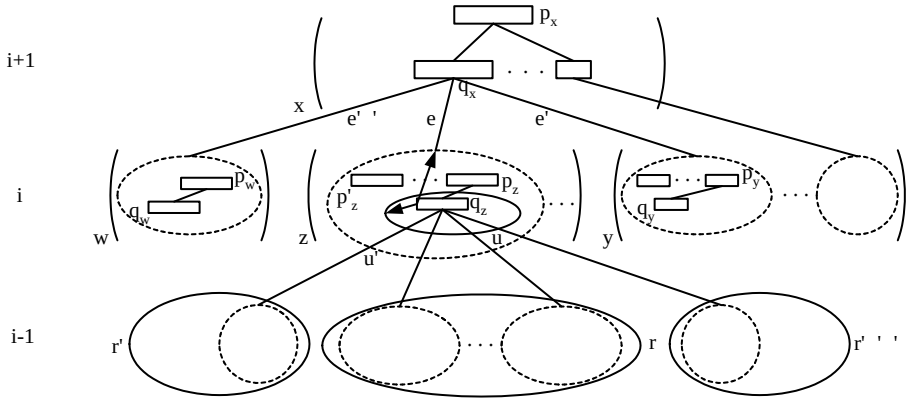

Fig. 19. The setting for the update algorithm given in Figure 20 is depicted. Dotted ellipses represent nodes under construction while dense ellipses represent ordinary nodes.

Procedure Update(f)
1. $r \leftarrow$ Find(f)
2. $r'' \leftarrow$ MultiBreak(r)
3. **if** $(4 \cdot f_{i-1} < ||r|| < b'_{i-1})$ **then**
4. $\quad r''.type \leftarrow 1$
5. **if** (r'' has only one block$_1$) **then**
6. $\quad$ **if** $(||r''|| + ||r'|| \leq 4 \cdot f_i)$ **then**
7. $\quad\quad$ GFuse(r'',r')
8. $\quad$ **else**
9. $\quad\quad$ GShare(r'',r')
10. **if** $(4f_i < ||z|| < b'_i)$ **then**
11. $\quad p_z \leftarrow r_r.block2$
12. $\quad$ **if** $(||z|| > 8f_i)$ **then**
13. $\quad\quad$ **if** $(||p_z|| > ||p'_z||)$ **then**
14. $\quad\quad\quad$ Transfer a block$_1$ from p'_z to p_z
15. $\quad\quad$ **else**
16. $\quad\quad\quad$ Transfer a block$_1$ from p_z to p'_z
17. $\quad$ **else**
18. $\quad\quad$ if $((||p_z|| > f_i)$ **AND** $(||p'_z|| < f_i))$ **then**
19. $\quad\quad\quad$ Transfer a block$_1$ from p_z to its mate p'_z
20. $\quad\quad$ **else**
21. $\quad\quad\quad$ Transfer a block$_1$ from p_z to a new block$_2$ p''_z
22. **elseif** $(||z|| \geq b'_i)$ **then**
23. $\quad u'' \leftarrow$ Split(u')
24. $\quad$ Add(e,$e_{u''}$)
25. $\quad$ **if** $((||q_x|| < a_i)$ **AND** $(q_x.mate =$ NULL) **AND**
$\quad\quad\quad\quad ((q_x.oldnode = q_x.left.oldnode)$ **OR** $(q_x.newnode = q_x.left.oldnode)))$ **then**
26. $\quad\quad$ Update $q_x.oldnode$ and $q_x.newnode$ as Lines 22-26 of *Delete(l)*
27. $\quad\quad$ Make a pair if possible between q_x and one of $q_x.left$ or $q_x.right$
28. **else**
29. $\quad$ **if** $(z.type = 1)$ **then**
30. $\quad\quad z.type \leftarrow 0$
31. $\quad$ Lines 5-33 of the *Delete(l)* algorithm in Figure 17

Fig. 20. The *update* operation is depicted when the update is performed at a leaf of group f. The setting is depicted in Figure 19. Certain variables (like z) are not set explicitly in the pseudocode for space reasons, but their initialization can performed in worst case constant time by following the proper pointers. The edge $e_{u''}$ corresponds to a child edge from q_x to the new node u'' (Line 21). Symmetric cases are omitted for simplicity.

these blocks into one block_2 starts incrementally. As a result, when the split interval is reached, there will be only one block_2 containing b'_i child edges. If u' has size smaller than $8 \cdot f_i$ and there are blocks_2 with degree larger than $f_i + 2r_{i-1}$, then these blocks are incrementally split into blocks_2 with degree at most equal to this bound. In any case, at level $i+1$ we just handle blocks_1 and blocks_2 by using the standard operations of Add and Remove (Lines 10-21).

If z is a split group, then the procedure is similar to the one described in Subsection 5.2 (Lines 22-27). On the other hand, if z is in the fusion interval, then the procedure is similar to the one described in Subsection 5.3 (Lines 29-31).

The only remaining detail is to examine the transition of a group between intervals. Assume that z switches from fusion interval to secure interval. In this case, the node under construction will contain at most four blocks_2. If in a future update operation, z switches back to fusion interval, then in the worst case z will have at most four blocks_2 again. If in a future update operation, z switches to the split interval, then it will surely have just one block_2. This is a direct consequence of the fact, that in the intermediate interval of size $b'_i - 8f_i = O(b'_i)$, incremental fusion operations on the blocks_2 are performed.

Assume that z switches from the split interval to the secure interval. Then, the degree of z is at least equal to $b'_i - r'_{i-1}$. In this case, z consists of only one node under construction that further consists of only one block_2. This is true, if and only if fusions between nodes under construction are allowed when the degree of these nodes becomes less than b'_i. The transition of z to the fusion interval is symmetric to the case discussed above.

5.4.2 Analysis

For the proof of the upper and lower bounds in the degree of groups we use the machinery used in 5.2.2 and 5.3.3 as well as the arguments in Subsection 4.3. The basic idea is depicted in Figure 18. The secure interval is so large, that there is no chance that a group may pass immediately from the fusion interval to the split interval and vice-versa. This means, that when a group is in the fusion interval, then it behaves like a fusion group. When a group is in the split interval, then it behaves like a split group. This is not completely true, because we allow the fusion of nodes under construction, in the case of insertions. However, this operation between blocks_2 does not affect the analysis of 5.2.2. In fact, the degree of nodes becomes even better since it is reduced. As a result, the lower and the upper bounds of the degree of nodes are still maintained.

The mechanism of components is not affected by the merge of the algorithms described in Subsections 5.2 and 5.3. The only detail, is the maintenance of components when a group that was in the fusion interval reaches the split

interval. The problem is that the fusion group contained many nodes under fusion, each of which, may as well belong to a different component. Under certain restrictions, this case will never occur. We will show that at least one MultiBreak operation will be applied to a group, that goes from the fusion interval to the split interval. This MultiBreak will be performed while the group lies in the secure interval. Initially, the degree of group G is at most $4 \cdot f_i$. Assume that consecutive splits are performed at its children. If all children groups of G are split groups, then in the worst case, G will have at most $4 \cdot f_i \cdot r'^2_{i-1}$ children before a MultiBreak is performed at G. If the following restriction holds, then we are certain that the MultiBreak is performed while G is in the secure interval:

Restriction 7 $4 \cdot f_i \cdot r'^2_{i-1} < b'_i$

To ensure Restriction 7 (recall that $\frac{b^2_{i-1}}{4} \leq b'_i \leq b^2_{i-1}$ and $r'_i = \frac{b_i}{b'_i}$) we need to have:

$$4 \cdot f_i \cdot (\frac{b_{i-1}}{b'_{i-1}})^2 < \frac{b^2_{i-1}}{4} \Rightarrow 16 \cdot f_i < b'^2_{i-1}$$

From the above we get the following restriction:

Restriction 8 $16 \cdot f_i < b'^2_{i-1}$

As a result the choice of f_i and b'_i must be made in such a way so that Restriction 8 holds. In our case, parameters have the following values: $b_i = 2^{2^{2i+4}}$, $b'_i = 2^{2^{2i+3}-2}$, $a_i = 2^{2^{2i}}$ and $f_i = 2^{2^{2i+1}}$.

Thus, verifying that Restriction 8 is true for the given parameter settings, we get:

$$2^4 \cdot 2^{2^{2i+1}} < (2^{2^{2(i-1)+3}-2})^2 \Rightarrow$$

$$2^{2^{2i+1}+4} < 2^{2^{2i+2}-4}$$

which is true for every value $i \geq 1$.

The following theorem summarizes the result of the discussion made in this section:

Theorem 10 *Assuming that finger searches can be implemented efficiently, we can maintain a finger search tree with worst case constant update time.*

From the choice of the values in the parameters, the following remark is made:

Remark 11 *The parameters b_i and a_i are related as follows: (i) $b_i = a_i^c$ and (ii) $a_i = a_{i-1}^{c'}$, where $c = 2^4$ and $c' = 2^2$.*

This remark will be crucial in the analysis of the time complexity of the search procedure.

6 The Search Operation

In this section, we show how the tree structure for insertions and deletions described in the previous sections, can support efficiently finger searches. In a finger search, we are given a finger f and an element x. The goal, is to locate element x in a tree structure, starting the search from element f, in $O(\log d)$ worst case time, where d is the number of elements between f and x. The following lemma, adopted from [19,10], will help in our discussion:

Lemma 12 *There exists a pointer-based implementation of search trees that supports updates in worst case constant time, and searches for arbitrary elements in $O(\log n)$ time, where n is the cardinality of the stored set.*

We represent each group, node under construction, block_1 and block_2 by using the structure of Lemma 12. In particular, the pointers of nodes under construction to groups are structured as Lemma 12. The same goes for the pointers of blocks_2 to the node under construction. Finally, Lemma 12 is used to structure the pointers from blocks_1 to blocks_2 and the child edges emanating from blocks_1. The search procedure starts from a leaf pointed by a finger f. It traverses the ancestor groups of f, until it locates the first ancestor u that contains x in its range. Then, it searches the subtree rooted at u, for element x. It is clear that if u is at the lowest level then the time complexity of the search procedure is constant (everything at the lowest level is $O(1)$). As a result, we will assume that the level of u is greater than 1.

It is imperative to show an upper bound on the distance between x and f based on the level of group u. Assume that u is at level i and that u has been reached from v (see Figure 21). The time to ascend a group by following parent pointers is clearly constant. Moreover, since the groups and the internal blocks are structured as in Lemma 12, the time to descend a group at level j is bounded from above by $\log b_j$, where b_j is the maximum out-degree of a group. As a result, the time complexity t of the described search procedure is bounded from above by $O(i) + \sum_{j=1}^{i} \log b_j$. Since the dominant factor of this equation is the sum $\sum_{j=1}^{i} \log b_j$, there exists a constant c_1 such that:

$$t \leq c_1 \sum_{j=1}^{i} \log b_j \Rightarrow t \leq c_1 \log \prod_{j=1}^{i} b_j \tag{9}$$

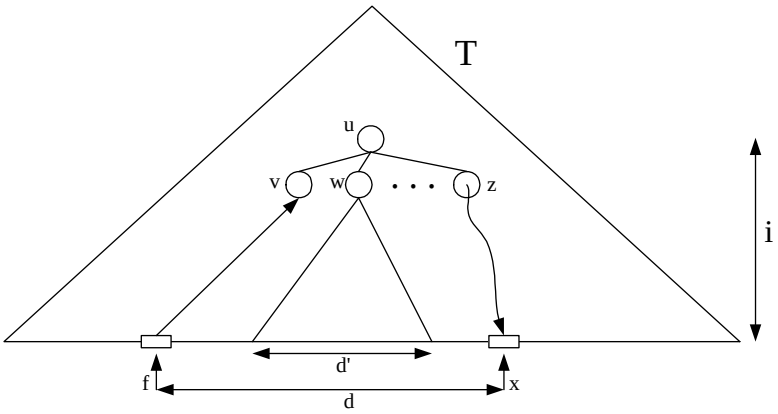

Fig. 21. The finger search procedure for element x.

We would like to bound t by $O(\log d)$. This will be achieved by using Equation (9).

Assume that w is the adjacent group of v, from which we reached u. Since we used level links during the procedure to locate node u, the leaves of the subtree T_w lie between the finger f and the element x. We would like to provide a lower bound for the distance d, between the finger f and the element x. This will be achieved by using the number of leaves d' in the subtree T_w. By using Invariant 1, we get:

$$d' \geq \prod_{j=1}^{i-2} a_j \frac{1}{2} \Rightarrow d' \geq \frac{1}{2^{i-2} a_i a_{i-1}} \prod_{j=1}^{i} a_j$$

where a_j is the minimum degree of a group, at level j. Since $d \geq d'$, we get:

$$d \geq \frac{1}{2^{i-2} a_i a_{i-1}} \prod_{j=1}^{i} a_j \Rightarrow \log d \geq \log \frac{1}{2^{i-2}} + \log \frac{1}{a_i} + \log \frac{1}{a_{i-1}} + \log \prod_{j=1}^{i} a_j \Rightarrow$$

$$\log d \geq -(i-2) - \log a_i - \log a_{i-1} + \log \prod_{j=1}^{i} a_j \tag{10}$$

Since, $t \leq c_1 \log \prod_{j=1}^{i} b_j$ (Equation (9)), and $b_j = a_j^c$ (Remark 11) we get:

$$t \leq c_2 \log \prod_{j=1}^{i} a_j \tag{11}$$

where $c_2 = c_1 c$. By using Equation (11) into Equation (10) we get the following:

$$\log d \geq \frac{t}{c_2} - (i-2) - \log a_i - \log a_{i-1} \Rightarrow$$

$$t \leq c_2(\log d + i + \log a_i + \log a_{i-1} - 2) \tag{12}$$

From Remark 11 it is known that $a_i = a_{i-1}^{c'}$. As a result, $\log a_i = c' \log a_{i-1}$ and $(\prod_{j=1}^{i-2} a_j)^{c'} \geq a_{i-1}$. Since $d \geq \frac{1}{2^{i-2}} \prod_{j=1}^{i-2} a_j$:

$$d \geq \frac{1}{2^{i-2}} a_{i-1}^{\frac{1}{c'}} \Rightarrow \log a_{i-1} \leq c'(\log d + (i-2)) \tag{13}$$

By combining Equations (12) and (13) the following holds :

$$t \leq c_2(\log d + (i-2) + c'(\log d + (i-2)) + c'^2(\log d + (i-2)) \Rightarrow$$

$$t \leq c_2 \cdot (c'^2 + c' + 1) \cdot (\log d + (i-2)) \tag{14}$$

The only remaining detail is to find the relation of i and $O(\log d)$. Since, $d \geq d' \geq \prod_{j=1}^{i-2} a_j \frac{1}{2}$ and $a_j \geq 4$ for all j, we get that $d \geq 2^{i-2} \Rightarrow i - 2 \leq \log d$. From the discussion above, the time complexity of the finger search is bounded as follows:

$$t \leq 2c_2(c'^2 + c' + 1) \log d \tag{15}$$

Concluding:

Lemma 13 *The time complexity of the search procedure in our structure is $O(\log d)$, where d is the distance between the finger and the element we wish to find.*

7 The Space Complexity

In the previous sections, we have shown that both insertion and deletion operations, can be performed in worst case constant time. It is tempting to use this fact, in order to argue that the space complexity is linear. However, there exist a weaknesses in this argument. This weakness stems from the fact that in each operation (insertion/deletion), we access a constant amount of memory elements, and leave some useless memory elements behind, without freeing

them. This results in the deterioration of the space complexity of the structure in the long term. This particular problem can be easily solved by applying standard incremental global rebuilding techniques. In this particular case, we make use of the new structure after at most $n/2$ update operations in the old structure.

Concluding, by Lemma 13 and Theorem 10 we get the final theorem that states the result of this paper:

Theorem 14 *There exists a tree structure that supports update operations in worst case constant time, finger searches in $O(\log d)$ worst case time and uses linear space. This structure is implemented in the Pointer Machine model of computation.*

8 Conclusions

In this paper we gave a solution to the long-standing problem of devising worst case constant update finger search trees in the Pointer Machine. This was accomplished, by using an innovative scheduling mechanism of rebalancing operations as well as an aggressive version of known rebalancing operations in (a,b)-trees (fusions and splits).

The solution is complicated and we would surely like to see a simpler solution to this problem that could also be applied to trees of constant out-degree. However, it is our intuition, that to do so, one must either enhance the technique of components or maybe combine it with some other techniques. In addition, it would be interesting to find applications of the component technique in other problems.

Acknowledgements

We would like to thank two anonymous referees for their crucial observations concerning the presentation and the correctness of the paper. We would also like to thank Mrs. Areti Papathanasiou for her help in the proof reading of the paper.

References

[1] G.M. Adel'son-Vel'skii and E.M. Landis: An Algorithm for the Organization and Information. *Dokl. Acad. Nauk SSSR*, 146:263-266 (In Russian). English Translation in *Soviet. Math.*, 3:1259-1262, 1962.

[2] Amir M. Ben-Amram: Pointer Machines and Pointer Algorithms: an Annotated Bibliography. Technical Report D-351, Department of Computer Science at the University of Copenhagen (DIKU), 1998.

[3] A. Anderson and M. Thorup. Tight(er) Worst case Bounds on Dynamic Searching and Priority Queues. In *Proc. 32nd Annual ACM Symposium On Theory of Computing (STOC)*, pages 335-342. ACM, 2000.

[4] M.J. Atallah, M. Goodrich and K.Ramaiyer. Biased Finger Trees and Three-Dimensional Layers of Maxima. In *Proc. 10th ACM Symposium on Computational Geometry*, pp. 150-159, 1994.

[5] G.S. Brodal. Partially Persistent Data Structures of Bounded Degree with Constant Update Time. *Nordic Journal of Computing*, 3(3):238-255, 1996.

[6] G.S. Brodal. Finger Search Trees with Constant Insertion Time. In *Proc. 9th Annual ACM-SIAM Symposium on Discrete Algorithms(SODA)*, pages 540-549, 1998.

[7] M.J. Clancy and D.E. Knuth. *A programming and problem-solving seminar.* T.R. STAN-CS-77-606, Dept. of Comp. Science, Stanford University, 1977.

[8] P. Dietz and R. Raman. A Constant Update Time Finger Search Tree. *Information Processing Letters*, 52:147-154, 1994.

[9] J.R. Driscoll, N. Sarnak, D.D.Sleator and R.E. Tarjan. Making Data Structures Persistent. *Journal of Computer and System Sciences*, 38:86-124, 1989.

[10] R. Fleischer. A Simple Balanced Search Tree with $O(1)$ Worst Case Update Time. *International Journal of Foundations of Computer Science*, 7:137-149, 1996.

[11] L. Guibas, J. Hershberger, D. Leven, M. Sharir, and R.E. Tarjan. Linear Time Algorithms for Visibility and Shortest Path Problems Inside Simple Polygons. *Algorithmica*, 2:209-233, 1987.

[12] L.J. Guibas, E.M. McCreight, M.P. Plass and J.R. Roberts. A New Representation for Linear Lists. In *Proc. 9th Annual ACM Symposium On Theory of Computing (STOC)*, pages 49-60. ACM, 1977.

[13] D. Harel. *Fast Updates with a Guaranteed Time Bound per Update.* T.R. 154, Dept of ICS, University of California at Irvine, 1980.

[14] D. Harel and G. Lueker. *A Data Structure with Movable Fingers and Deletions.* T.R. 145, Dept of ICS, University of California at Irvine, 1979.

[15] J. Hershberger. Finding the Visibility Graph of a Simple Polygon in Time Proportional to its Size. In *Proc. 3rd ACM Symposium on Computational Geometry*, pp. 11-20, 1987.

[16] K. Hoffman, K. Mehlhorn, P. Rosenstiehl and R.E. Tarjan. Sorting Jordan sequences in linear time using level-linked search trees. *Information and Control*, 68(1-3):170-184, 1986.

[17] S. Huddleston and K. Mehlhorn. A New Data Structure for Representing Sorted Lists. *Acta Informatica*, 17:157-184, 1982.

[18] S.R. Kosaraju. Localized Search in Sorted Lists. In *Proc. 14th Annual ACM Symposium On Theory of Computing (STOC)*, pages 62-69. ACM, 1981.

[19] C. Levcopoulos and M.H. Overmars. A Balanced Search Tree with $O(1)$ Worst Case Update Time. *Acta Informatica*, 26:269-277, 1988.

[20] G. Lagogiannis, C. Makris, Y. Panagis and K. Tsichlas. New Dynamic Balanced Search Trees with Worst Case Constant Update Time. *In Proc. of the 13th Australasian Workshop on Combinatorial Algorithms (AWOCA 2002)*, 2002.

[21] K. Mehlhorn and A. Tsakalidis. *Handbook of Theoretical Computer Science – Vol I: Algorithms and Complexity*, Chapter 6: Data Structures, pp.303-341, The MIT Press, 1990.

[22] M.H. Overmars. An $O(1)$ Average Time Update Scheme for Balanced Search Trees. *Bulletin of EATCS*, 18:27-29, 1982.

[23] R.E. Tarjan. A Class of Algorithms which Require Nonlinear Time to Maintain Disjoint Sets. *Journal of Computer and System Sciences*, 18:110-127, 1979.

[24] R.E. Tarjan. Updating a Balanced Search Tree in $O(1)$ Rotations. *Information Processing Letters*, 16:253-257, 1983.

[25] A.K. Tsakalidis. AVL-trees for Localized Search. *Information and Control*, 67:173-194, 1985.