Alcom-FT Technical Report Series
ALCOMFT-TR-02-77

Optimal Finger Search Trees in the Pointer Machine

Gerth Stølting Brodal* George Lagogiannis† Christos Makris†
Athanasios Tsakalidis† Kostas Tsichlas†

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.

Keywords balanced trees, update operations, finger 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 ([6, 14]). It has also been known ([6, 16]) that updates can be performed in $O(1)$ structural changes, but the nodes to be changed have to be searched in $\Omega(\log n)$ time. Levcopoulos and Overmars ([13]) presented an algorithm achieving $O(1)$ worst case update time by using a global splitting lemma that is based on a pebble game combined with the bucketing technique of Overmars ([14]). 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 [13] have size $O(\log^2 n)$, so they need a two level hierarchy of lists in order to guarantee $O(\log n)$ query time within the buckets. Deletions are handled by means of global rebuilding. Fleischer ([7]) 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. Insertions are performed by first inserting the new key into its respective bucket. Let u be the node pointed by the pointer of the bucket. If u has out-degree larger than b then u is split into two small nodes, otherwise u is left intact. In any case the pointer of the bucket is moved up one level. It is proved that starting from an (a, b) tree, the degree

*Department of Computer Science, University of Aarhus, BRICS (Basic Research in Computer Science), www.brics.dk, funded by the Danish National Research Foundation. Email: gerth@brics.dk. Partially supported by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT). Research conducted while visiting Computer Technology Institute (CTI) and University of Patras, Greece.

†Department of Computer Engineering & Informatics, University of Patras & Computer Technology Institute, Patra, PO 22500. Email: {lagogian,makri,tsihlas}@ceid.upatras.gr, tsak@cti.gr.

of the internal nodes cannot grow more than $2b$ and the size of the buckets can grow up to $2\log n$. Deletions are handled simply by using the global rebuilding technique.

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*) and the time complexity of the search procedure is asymptotically equal to the logarithm of the distance between the finger and the search element. In the RAM model of computation finger search tres with constant update time have already been devised by Dietz and Raman ([5]), while recently Andersson and Thorup ([1]) have surpassed the logarithmic bound on the search procedure. These two papers are based on a global rebalancing scheme combined with the bucketing techniques presented in [13]. For the pointer machine model of computation, steps have been made towards this direction by researchers (see [3, 4, 8, 9, 10, 12, 17]), but the problem remained tantalizingly open. The best solution is given by Brodal ([3]), who proposed a finger search tree with constant insertion, but with $O(\log^* n)$ deletion time. This time bound of the delete operation is a direct result of our difficulty to handle efficiently deletions in a local rebalancing setting.

In this paper we will present the first constant update finger search tree. Note that the space requirements of this structure will be, by contruction, linear since in constant time we can only access constant amount of space. However, the technique of global rebuilding is essential to guarantee a linear bound on space complexity in the long-term. In Section 2 we describe the basic technique used to achieve the alleged result. In Section 3 we describe a finger search tree that supports only the operation of insertion in worst-case constant time. This structure can be seen as an alternative solution to the insertion only algorithm of [3]. In Section 4 we describe a finger search tree that supports solely the operation of deletion in worst-case constant time. In Section 5 we sketch the mechanism needed to guarantee efficient finger searches. Finally, in Section 6 we combine both solutions analyzed in previous sections and we conclude at Section 7 with some final remarks. We must note that we left many technical details for a future journal version.

2 The components

The technique of *components* is based on an idea previously used in the work of Brodal on making worst-case partially persistent data structures ([2]). Components define a logical partition over the set of non-leaf nodes of the finger search tree into connected subtrees that dictate the position of the rebalancing operations. We assume that we are given a height-balanced search tree T and a component A over the nodes of T, where A is a subtree of T rooted at the node $A.root$. All leaves of T have equal depth. We say that the leaves are at level 0 while the level of a node is equal to the level of its children plus one. The maximum degree b_i and the minimum degree a_i of a node at level i is a function of i with the exception of the root that has minimum out-degree equal to two. Let $r_i = \frac{b_i}{a_i}$, be the ratio of b_i and a_i for level i.

Initially, all nodes of T are singleton components, that is components with only one node. The root of a singleton component is the node itself. For the general component A root of the component is $A.root$ and this node will be the handle of the component. In addition, all nodes $v \in A$ who have a child u such that $u \notin A$ consistute the *border* of component A, which is represented by $border(A)$. As we will see in the following sections the border of a component is the part of the component that absorbs all the disturbances caused by rebalancing operations

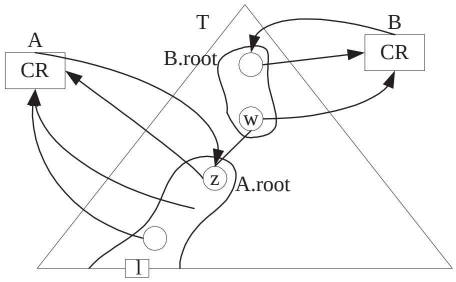

Figure 1: The structure of a component A in a tree T. By CR we represent the component record.

in their subtrees. In Figure 1 we depict the structure of a component A on a tree T with root $z = A.root$. These components must be maintained under the operations of *Link* and *Find*, which is well known that cannot be implemented in constant worst-case time in the PM (see [15]). However, here we exlpoit the special structure of the components and as a result we are able to acquire a worst-case constant time complexity. We would like to support the following operations on components in worst-case constant time:

1. *Break(z):* the component A with handle z is destroyed and the nodes that once belonged to A become singleton components.

2. *Add(v,z):* adds the node v in the component with handle z (component A in Figure 1) containing the father of v.

3. *Find(v):* returns the root of the component where node v belongs.

4. *Link(u,z):* links components with handles u and z into a new single component with handle u. It is assumed that nodes u and z are siblings and that immediately after the Link operation z is absorbed by u (an incremental fuse operation in the setting of (a,b)-trees).

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 *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 $v \in A$ has a pointer to the component record of A. In this way operation *Find(v)* is performed in worst-case constant time. Operation *Add(v,u)* is also performed in constant time since we just need to make the pointer of node v to point to the component record of the component with root u. We assume that the component with root u is valid. Operation *Break(z)* is easily performed in constant time by setting the flag *valid* of the component record of the component with handle z equal to FALSE. Finally, operation *Link(u,z)* can be implemented in constant worst-case time due to the special structure of the components. This is achieved by using *fusion records*. The components that must be joined have their component records point to this fusion record which further points to the root of the joined components. Thus, each component may be represented by its component record or by the fusion record when the component record points to one. When two components that are represented by fusion records become joined, we incrementally move the component records from the one fusion record to the other. In Section 4 this operation is necessary and we will see there that we have enough time to perform it incrementally.

We will now sketch how the mechanism of components is integrated in a height balanced tree T to guide the rebalancing operations as a consequence of the update operations in its leaves. Assume a leaf l, its father f and its grandfather ff and assume that l is the *receiver* of an update operation (either insertion or deletion). The receiver of an update operation is the leaf pointed by the finger that dictates the position of the update operation. If f is a singleton component (either the component record to which f points indicates that f is the root or this component record is invalid) then a rebalancing operation is performed and it is added to the component of ff. If f belongs in a non-singleton component A, then we rebalance $w = A.root$, we break A and finally we add w in the component of its father. Thus, when an update operation takes place inside a component (that is at a leaf of a node that belongs to this component) then we make a rebalancing operation at the root of this component. Operation Link is necessary as we will see in Section 4 for the performance of fusion operations. We silently assumed that we have an efficient rebalancing scheme and that in addition this scheme guarantees a controllable out-degree of internal nodes. In the following sections we will see how to cancel these assumptions.

The main problem with the technique of components is that it is not fast enough concerning the traversal of the ancestors with large height. This fact is given in the following lemma.

Lemma 1 *A node v is rebalanced after its children are twice rebalanced in the worst-case.*

Proof: Assume a node v in the component A that belongs to $border(A)$. Then, all of its children must be rebalanced at least one time to ensure that $v \notin border(A)$. When A breaks, then all of the children of v may be rebalanced before v gets rebalanced as a root of a non-trivial component (a component which does not consist of a single node). In addition, a node is rebalanced three times if and only if the component mechanism does not add this node to the component of its father. However, it is clear by the mechanism of components that this situation is not possible and as a result the lemma follows. □

Lemma 1 means that the cost of rebalancing, when using this scheduling algorithm, is exponentially increasing with respect to height j. Thus, this technique is not enough by itself to guarantee a bounded out-degree of internal nodes. However, we will see in the following sections that by making the rebalancing operations more aggressive we can compensate for the inefficiency of the component mechanism. The term “aggressive” means that instead of making binary splits or binary fusions we perform multiple splits and multiple fusions respectively.

3 The case of Insertions

In this section we assume that we can only insert new elements in the tree structure (the discussion for deletions is postponed to the next section). The solution we are going to describe can be seen as a simplified/alternative solution to the insertions only algorithm described in ([3]). First we must formulate the main problem of the component mechanism. This problem is located in the border of a component. In Figure 2 we depict the four different situations that can happen in $border(A)$. The colors over the edges describe the potential of the child node to give new nodes through successive splitting operations.

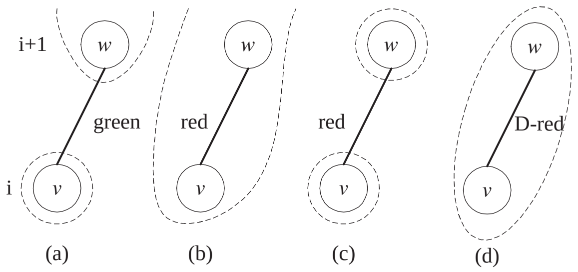

Figure 2: The definition of split groups at the border of components (v is at level i while w is at level $i+1$), (a) v is a singleton component while w belongs to a non-singleton component, (b) v is inserted into the component of w, (c) the component of w is broken and (d) v belongs to the component of w where w is the root.

Assume node v at level i with maximum degree b_i. To guarantee an upper bound on the out-degree of internal nodes when we use the component mechanism we must perform a multiple split operation, that is to split v into many nodes v_i with out-degree at least equal to a_i. Thus, we need to split v into at most r_i nodes. In Figure 2 it is easy to see that when edge (w, v) is green (phase (a)) then all children of w may split resulting in phase (b). Phase (c) is derived from phase (b) by performing a *Break* operation on the component that w belongs. Finally, phase (d) is derived from phase (c) when all children of w are again multiple split. 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. From the discussion above it is clear that D-red edges don't produce any new nodes. 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 $i+1$ due to splitting operations at level i. We define the potential function for an arbitrary node at level $i+1$ represented by v_{i+1} as follows:

$$\Phi(v_{i+1}) = r_i^2 \cdot \#\text{green} + r_i \cdot \#\text{red} + \#\text{D-red}, \tag{1}$$

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

We would like to bound this potential by the maximum degree of v_{i+1}, that is we need to choose b_{i+1} such that:

$$\Phi(v_{i+1}) \leq b_{i+1}. \tag{2}$$

In the above discussion we implied that multiple splitting of a node is imperative to guarantee an upper bound on the out-degree of internal nodes. The problem is how to implement this multiple splitting procedure in constant worst-case time while satisfying Equation 2. This will be achieved by using multiple levels of indirection and by incremental scheduling techniques.

It is essential to describe how child pointers are structured in this tree structure T. The *nodes* of tree T are partitioned into *split groups* and only consecutive nodes may be part of a split group. Nodes at level i contain at least a_i child pointers and at most b_{i-1}^3. The child pointers of a node are further partitioned into *blocks* with each block at level i storing less than $a_i + 2b_{i-1}$ edges. The partitioning of the child pointers of a node into blocks is accomplished by structuring blocks in a three-level tree structure. The component mechanism on tree T is defined over the set of split groups in T. Intuitionally, if we collapse each split group into a

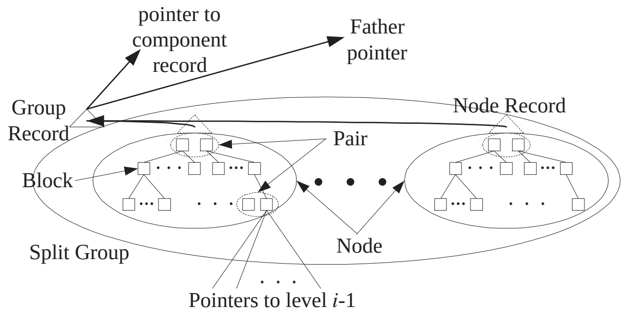

Figure 3: The structure of a single split group at level i. The split group consists of a set of nodes which are further partitioned into a three-level hierarchy of blocks.

single node then the tree that results (we can call it the *group tree*) in fact implements the multiple split operation, as we will see later in this discussion. The operations applied on split groups are *Break(G)*, which breaks the split group G into singleton split groups (split groups with just one node) and *Add(v,G)* that adds node v to split group G.

Each split group is represented by a split group record with a pointer to the respective component, a boolean field flag that indicates whether the split group is broken or not and a father pointer. Each node has a node record that stores a pointer to the split group of the node. For a node w we allow the operation *Split(w)* that creates a new node w' to the right of w (w and w' both belong in the same split group). The new node w' is formed by moving a set of blocks of child pointers from w to w'. Two operations can be applied on blocks, *Add(e,e′)* and *Remove(e)*. The former operation adds a new child pointer e' next to the child pointer e while the latter operation removes a child pointer e from the block. Finally, we apply on blocks the *pair* mechanism, which in fact implements the incremental splitting of this object. Each pair of blocks consist of two blocks and each block is contained in exactly one pair. All pairs of blocks at level i store at least a_i child pointers with the possible exception of the root. On pairs we apply the operation *Break(p)* so that if $p = (B_1, B_2)$, where B_i is a block and $|B_i| \geq a_i$, then after this operation two new pairs $p' = (B_1, B_1')$ and $p'' = (B_2, B_2')$ substitute p where B_1' and B_2' are empty blocks. In Figure 3 we depict the internal structure of split groups.

Firstly, we will describe the insertion algorithm based on Figure 1, and then we will proceed to the analysis discussion.

So assume that we want to insert a new element immediately to the right of an element l. We first locate the root $z = A.root$ of the component A that contains the father of l. Break component A and insert split group z into the component B of its father w (note that w is just a node that belongs to a split group). Break z into singleton groups and split w into two nodes, w and w'. The new node w' is formed by moving from w the rightmost block at level two of the internal structure of blocks. Then w' is added to the block that contains w as a child (this is the implementation of operation *Add(e,e′)*). The new node w' belongs to a pair of blocks. If the leftmost block of the pair has size $\geq a_i + b_{i-1}$ move the rightmost edge of the left block to the right block. If the right block has size $\geq a_i$ and there is a split group with edges in both blocks of the pair move one edge from left to right. Note that after at most b_{i-1} such operations, all elements of the split group have been moved to the right block and the pair can be split at this time. Finally, check the levels two and three of the hierarchy of blocks for pairs that need splitting.

In order to prove the correctness of the described procedure we need to guarrantee an upper bound on the degree of the internal nodes. So,going back to Figure 1 assume that $z = A.root$ is a split group and that its father is a node (inside a split group) w that belongs to the component B. Then after some operation in a leaf that belongs to component A, we break A and we break split group z into singleton groups containing only one node. Finally, we insert these singleton split groups to the component of w. The break of a split group into singleton split groups can be implemented in a lazy way by maintaining a flag in the split group record that indicates whether the split group is valid or invalid.

We stated previously that the size of a block at level $i+1$ is bounded by a_{i+1} and $a_{i+1}+2b_i$. Each node has at most $2b_i^2-1$ and at least 1 such block. In this way, each node groups at most $6b_i^3$ (requiring $a_{i+1} < b_i$) child pointers and at least a_{i+1} child pointers. As a consequence, when starting from a singleton split group at level $i+1$ we can clearly see that the split group has $< 6b_i^3$ out-degree. Since the maximum degree at level i is b_i and we want to have singleton split groups with minimum degree equal to a_i we may split each split group into at most $r_i = \frac{b_i}{a_i}$ singleton split groups. Thus, if we assume that initially we start with a singleton group g_{i+1} at level $i+1$ whose child pointers may be of arbitrary color (green, red or D-red) then the maximum number of nodes before being split into singleton groups is r_i^2.

For the split group g_{i+1} equation 2 states that $\Phi(g_{i+1}) \leq b_{i+1}$. We need to ensure that the potential increases only under certain conditions so that we can ensure the validity of equation 2. Since one green edge may produce r_i red edges and one red edge may produce r_i D-red edges and by equation 1 we deduce that the potential does not increase during the transitions between the four phases shown in Figure 2. This holds for all cases with the exception of the transition from phase (d) to phase (a), where D-red edges may again become green. In this way we have an increase in potential of order $O(r_i^2)$ for each D-red edge. However, multiple splitting corrects this problem by making singleton groups with a controllable out-degree.

From the above discussion it is easy to deduce that the potential bounds the number of child pointers which may be produced by a singleton group. Assuming that this singleton split at level $i+1$ has maximum capacity (it has $< 6b_i^3$ out-degree) and that all child pointers are green by equations 1 and 2 we deduce that:

$$6b_i^3 r_i^2 \leq b_{i+1} \Rightarrow 6b_i^3 (\frac{b_i}{a_i})^2 \leq b_{i+1} \Rightarrow 6b_i^5 \leq b_{i+1} \tag{3}$$

From recurrence 3 we get:

$$\begin{aligned} &6b_i^5 \leq b_{i+1} \Rightarrow 6(6(\dots)^5)^5 \leq b_{i+1} \Rightarrow \\ &6^{5^0} 6^{5^1} 6^{5^2} \dots 6^{5^i} \leq b_{i+1} \Rightarrow 6^{\sum_{j=0}^{i} 5^j} \leq b_{i+1} \Rightarrow \\ &6^{\frac{5^i-1}{5-1}} \leq b_{i+1} \Rightarrow 6^{\frac{5^i-1}{4}} \leq b_{i+1} \Rightarrow b_{i+1} \geq 2^{2^{3i-1}} \end{aligned} \tag{4}$$

The above analysis would be fully correct if we had a way to ensure that binary splittings caused by multiple splittings one level lower are terminal so that no cascading splittings of nodes are produced. Unfortunately, the definition of potential does not capture this situation but as we will see with the appropriate tuning of the mechanism that implements multiple splitting we circumvent this problem. We will base our argument on Figure 4. The mechanism

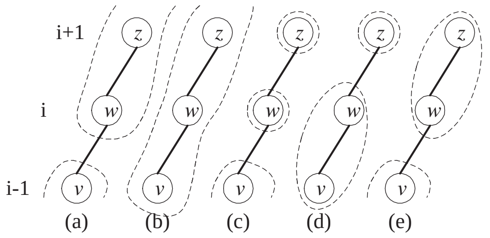

Figure 4: The consecutive phases of binary splits: (a) group w was just multiple split, (b) the multiple split of v results in the addition of a child pointer at z, (d) the multiple split at level $i-1$ results in the addition of one more child pointer at level $i+1$, (e) a multiple split takes place at level i. A single split group at level $i-1$ when in phase (a) may produce up to $b_{i-1}+1$ child pointers at level $i+1$ when in phase (e).

of components as shown in the previous section ensures that a split group at level i will be rebalanced only when the split groups at level $i-1$ are twice rebalanced. This means that a single child pointer of a node at level $i+1$ may produce at most $2b_{i-1}^4$ new edges due to multiple splittings at level $i-1$ that produce binary splittings at level i. Since a node at level $i+1$ has maximum capacity b_i^3 and by the above observations we deduce that after the phases depicted in Figure 4 the number of child pointers of the node at level $i+1$ will be at most $b_i^3 b_{i-1}^3 b_{i-1}$, where b_i^3 is the maximum initial out-degree of a node at level $i+1$, b_{i-1}^3 is the maximum out-degree of a node at level i and b_{i-1} is for the double splitting due to the component mechanism. This means that the out-degree of nodes is increased in an uncontrollable fashion. However, if we change slightly the multiple splitting mechanism we can ensure that this situation will not happen. When a multiple splitting at level i takes place (creates b_i new split groups) then at level $i+1$ we create a new node by moving b_i^2 child pointers. This means that we let at this node $b_i^2 - b_i$ free space for edges from binary splittings. Since we have b_i new nodes we demand that this free space is enough for all pointers from binary splittings. Thus, from the above discussion we demand:

$$b_i^2 \geq 2b_i b_{i-1}^4 \Rightarrow b_i \geq 2b_{i-1}^4 \tag{5}$$

However, equation 5 is fully covered by equation 4 and so by choosing a value for b_{i+1} such that equation 4 is satisfied we know for sure that a node will always have place for pointers due to binary splittings and so we can ensure that as far as insertions are considered the out-degree of internal nodes will be bounded.

The following theorem (matching the bounds stated in [3]) is the result of the above discussion:

Theorem 1 *Assuming that finger searches can be implemented efficiently, we can maintain a finger search tree with worst-case constant finger insertion time when deletions are not allowed.*

4 The Case of Deletions

In this section we assume that the only allowable operation on an initial set of elements, which is represented by the tree structure T, is *delete(l)*, where l is a pointer to the leaf that is going to be deleted. The scheduling mechanism for the rebalancing operations is the mechanism of components. Due to the inefficiency of this mechanism described in Section 2 we need to resort to a multiple version of known rebalancing operations for the case of deletions. In this case we are going to use the notion of *multiple fusion*. The multiple fusion is like the ordinary fuse operation for (a,b)-trees with the diference that many nodes participate in this operation. Generally, the strategy we follow for deletions is symmetric to insertions and this symmetry will be made explicit whenever necessary.

Assume that we have access to a procedure that we call *oracle*, which performs the multiple fusion procedure in constant worst-case time. Thus, the oracle is a mechanism that takes as input a set of adjacent brother nodes and outputs in $O(1)$ worst-case time a single node that results from the fusion of all these nodes. The set of nodes that participates in a call to the oracle is called *fusion group*. We will see later in this section how to cancel this assumption. First, recall Figure 1. In a nutshell, the algorithm for the deletion of a leaf l consists of five steps: a) find the set A in which the father of l belongs, b)remove leaf l, c) break component A, d) call the oracle for the root $x = A.root$ of the component A and e) add to the component of its father the new node x' produced by the oracle. Below we show, based on the assumption of the existence of such oracle, that the above algorithm is correct.

First, we need to define the *fusion factor*, a_i, for all nodes of each level i. The fusion factor for level i is the required out-degree for each new node produced by a call to the oracle. Thus, after the application of a rebalancing operation implemented by the oracle at level i, we can assure that the new node v has out-degree a_i. We set the lower bound on the degree of a node to be equal to 2 - in the worst-case we expect to have a binary tree. Considering the fusion factors of levels i and $i-1$ (a_i and a_{i-1} respectively) we may generate a recurrence relation that bounds the fusion factor, based on the fact that the lower bound in the degree of a node is equal to 2. In this way we produce the following recurrence relation: $\frac{a_i}{(a_{i-1}/2)^2} \geq 4$, where $a_1 = 4$ (at level 1 we want at least an out-degree of 4). To generate this recurrence relation we have to note that a node at level i with initial out degree a_i may, by applying multiple fuse operations twice at its children due to Lemma 1, have at least degree equal to 2. The oracle at level $i-1$ needs to fuse at most $a_{i-1}/2$ nodes (thus, the maximum number of nodes inside a fusion group is $a_{i-1}/2$) with minimum degree 2 during a multiple fuse operation at this level while by Lemma 1 the multiple fuses will involve at most $(a_{i-1}/2)^2$. It will be made clear below why we have chosen this fraction to be larger than 4 and not larger than 2.

Thus, by solving the above recurrence we get:

$$a_i \geq a_{i-1}^2, a_1 = 4 \Rightarrow a_i \geq 4^{2^i} \Rightarrow a_i \geq 2^{2^{i+1}} \tag{6}$$

Choosing a value for a_i such that equation 6 is satisfied we may ensure that the above algorithm implements deletions in constant worst-case time with a guaranteed non-trivial lower bound on the degree of the nodes. However, there are two issues that need clarification. The first one refers to the maintenance of components under the multiple fuse operation while the other refers to the structure of nodes and the maintenance of the lower bound on the degree of the nodes.

The first problem as we mentioned above is to maintain the components during a multiple fuse operation. We will assume that a multiple fuse operation is in fact a sequence of ordinary binary fusions. All possible cases for the fusion of adjacent nodes are depicted in Figure 5. In case (a) we break component rooted at w, we fuse it with x and add the new node to the component of z. In case (b) we break both components rooted at w and x and add the new

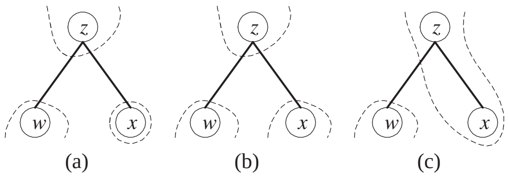

Figure 5: The three possible different situations in the fuse operation between fusion groups. All other situations are symmetric.

node that comes from the fusion of w and x at the component of its father. Finally, case (c) is easy to implement by breaking the component rooted at w and inserting w into its father component by fusing it with its brother x. We must also ensure that this fusion will not lead to a single fusion group as a child of a fusion group at the upper level. If this happens, then we cannot guarantee the lower bound of 2. This is due to the fact that the distribution of update operations between the different subtrees is not known. The following lemma solves this problem.

Lemma 2 *For every distribution of update operations between subtrees rooted at the children of a node v, the choice of the fusion factor given in equation 6 can guarantee that the number of fusion groups is at least* 2.

Proof: Note that in equation 6 we demand that the fraction be larger than 4. This means that if we were given a worst-case sequence of operations at the children of a node v at level i beforehand (offline updates), then we could guarantee that at the end (just before rebalancing v at level i), v would have exactly 4 children (this would work even for 3 children). However, since the updates are online we have to define fusion groups on the fly. However, even in this case we can guarantee a lower bound of two children for each node since there are always many children to construct two fusion groups indepedently of the distribution of the update operations. □

In addition, note that the early break of the components does not incur any problems with the component mechanism. In fact, this early break accelerates the mechanism of components.

We must also consider the internal structure of nodes. Assume a node at level i. This node is structured into at least a_{i-1} blocks of size exactly a_{i-1}. Thus, the out-degree of a node at level i is a_{i-1}^2 immediately after the multiple fuse operation, which of course satisfies equation 6. In this way, each block defines at least two new nodes at level $i-1$ (due to Lemma 2) after the application of the multiple fusion operation at this level. Thus, the block may have after this operation less that a_{i-1} child pointers. To remedy this problem we form a pair between this block and one of its adjacent blocks. If both adjacent blocks already belong in a pair then break an existing pair and create a new one. The mechanism of pairs guarantees that there will be no cascading breaks of pairs. Nodes (and as a result fusion groups) are also structured into pairs, so that incremental fusion between nodes is possible. Lemma 2 ensures that each node will have at least two children.

In the above discussion we assumed the existence of an oracle to show that by using components and the mechanism of multiple fusion one can come up with an implementation of deletions in constant worst-case time while keeping the structure balanced. At this point, we need to cancel this assumption. To achieve this, we have to implement the multiple fusion operation in an incremental way. This means that the fusion groups are constructed incrementally and are not formed in constant worst-case time as in the imaginary mechanism of the oracle. The fusion group, as defined above, is a set of consecutive nodes that at the end will form by fusion a single new node. The total out-degree of a fusion group at level i must be at least a_i, since each fusion group is in fact a node under construction. In addition, note that components are defined on fusion groups since these will become single nodes at some time in the future. Below we describe when and how the fusion groups are formed.

First we give some details of the deletion algorithm based on Figure 1, assuming that we want to delete leaf l. The algorithm follows:

1. Find the component A in which the father of l belongs. Let *new_fg* be the new fusion group which is the root of A at level $i-1$.

2. Break A. Let B the component of its father.

3. Add(*new_fg*,B). Let x_j be the node which is father of *new_fg* and let x_{j+1} be its brother node inside the pair or if it does not belong in the pair one of its adjacent nodes. Both nodes belong in the same fusion group fg at level i (note that fg is the real node while x_j and x_{j+1} are nodes which should be fused).

4. Make an incremental fuse operation between x_j and x_{j+1}.

5. If $out\_degree(fg) \leq a_i$ then fuse an adjacent fusion group with fg and fuse the respective adjacent blocks at level $i+1$ (all these fusions are incremental).

The above algorithm makes two assumptions. The first assumption is that a fusion group is fully contained in a block one level above. This assumption holds by using the inductive argument stemming from the maintenance of the blocks in the deletion algorithm given above. The second assumption is that the fusion between nodes that takes place as a result of a multiple fuse one level below may damage the out-degree of the fusion group one level above. However, this is not true since the fusion between two nodes due to a multiple fuse in the level below is contained in the incremental construction of the single node by the fusion group containing both nodes. Thus, generally one could say that a multiple fusion (the incremental fusion of fusion groups) at level $i-1$ results in a binary fusion (the incremental fusion of nodes) at level i which further results in the removal of a child edge from a block at level $i+1$. Symmetrically, in insertions a multiple split at level $i-1$ results in a binary split at level i which further results in the insertion of a new child pointer in a block at level $i+1$.

In addition, the fusion of adjacent fusion groups affects the component mechanism. All possible cases for components are depicted in Figure 5. Case (a) is easy to solve by inserting fusion group x into the component with root the fusion group w. Case (c) is also easy to solve by breaking the component with root w and inserting w into the component where x belongs, after initiating the incremental fusion of w and x. Since x is a full fusion group (it has the necessary out-degree) one can say that it may absorbs all of its brothers and thus violate the lower bound of 2 on the degree of fusion groups. However, this is not the case since we may fuse w with its right or left brother and as a result in the worst-case this procedure will lead to two fusion groups at the level of w. Case (c) can be tackled by using the *Link* operation described in Section 2. The aim here is to unite both components rooted at w and x. This can be accomplished by the *Link* operation. However, we mentioned in Section 2 that we may have to incrementally join two components represented by fusion records. The incremental step is to make a constant number of component records to point to one of the two fusion records and it is carried out while executing step 4. Since this procedure starts as soon as the fusion group has $a_i - 1$ out-degree we have enough time available to execute the transfer of componenet records from one fusion group to the other.

To summarize, the mechanism of components is applied on fusion groups. The components are represented by component records as in the case of insertions, but to achieve a worst-case constant implementation of the union of two adjacent components (the adjacency is defined with respect to their roots) we introduced another level of indirection, the fusion records. Thus, each component is represented either by a component record when the pointer field that points to the fusion record is nill or by the fusion record pointed by this pointer. Each fusion group represents a future node and consists of a set of nodes with the restriction that its out-degree is at least equal to a_i.

In the above discussion we devised a data structure that implements deletions in $O(1)$ worst-case time in the PM model of computation. The only problem remaining is to combine both solutions into a new one that implements the worst-case constant update finger search tree and to show how to support finger searches in $O(\log d)$ time, where d is the distance between the finger and the element we search.

Theorem 2 *Assuming that finger searches can be implemented efficiently, we can maintain a finger search tree with worst-case constant finger deletion time when insertions are not allowed.*

5 The Finger Search Operation

In this section we sketch how the tree structure for insertions and for deletions can support efficiently finger searches. In a finger search we are given a finger f and an element x and we want to find element x in the tree structure starting the search from f and completing the task in time $O(\log d)$, where d is the distance between f and x in the structure. The main problem with the structure is that there are nodes which have large degree and thus we need to investigate the complexity of the search procedure in these nodes.

If we solve the problem for the insertion structure then we can apply the same approach for for the structure supporting deletions as well as for the general structure. Each block at level $i+1$ - considering the insertion tree structure - has maximum degree $3b_{i-1}$, for which by equation 4 holds that $b_i \geq 2^{2^{3(i-1)-1}}$. Thus, the degree of the blocks at level i is $O(2^{2^i})$. First we use level linking ([11]) on the node tree structure. As a result, the search procedure basically starts from a leaf pointed by a finger f and traverses the ancestors of f until we find the first ancestor v (or one adjacent to this node by using level pointers) that contains x in his range. Then we search the specific subtree for x. It is imperative to show an upper bound on the distance between x and f based on the level of node v. Assume that v is at level i. Then, the maximum distance d will be $O(2^{2^i})O(2^{2^{i-1}})\ldots O(2^{2^1}) = O(2^{2^{i+1}})$. The lower bound on the distance between f and x when v is at level i is $O(2^i)$ since the minimum degree of internal nodes is 2. The following Lemma from [3] will help:

Lemma 3 *There exists a pointer-based implementation of finger search trees which supports arbitrary finger searches in* $O(\log\log n + \log d)$ *time, and finger updates in worst-case constant time.*

Proof: The lemma is obtained by combining the finger search trees of Dietz and Raman ([5]) and the search trees of Levcopoulos and Overmars ([13]). For more details see [3]. □

We represent each block by using the structure of Lemma 3. Internal blocks (they are used in the three level tree structure for blocks) are also structured by using Lemma 3. Thus, the search procedure at the blocks B_k of node v can be performed in $O(\log\log|B_k| + \log d)$. In the worst-case $|B_k| = b_i = 2^{2^{3i}}$ and as a result the finger search can be performed in $O(\log\log 2^{2^{3i}} + \log d) = O(i + \log d)$. Assuming that they have maximum distance ($d = \Omega(2^{2^i})$) we get (for minimum distance it is symmetric):

$$O(i + \log d) + \sum_{j=1}^{i-1} O(\log 2^{2^j}) \Rightarrow$$

$$O(2^i + \log d) \Rightarrow O(\log d) \tag{7}$$

since $\log d = \Omega(2^i)$.

The procedure sketched above can be as well applied in the case of deletions with minor modifications. In addition, the same techniques can also be applied in the combined solution for insertions and deletions described in Section 6. The following theorem summarizes the discussion in this section.

Theorem 3 *There exists a pointer-based implementation of finger search trees that support arbitrary finger searches in $O(\log d)$ time and finger insertions in worst-case constant time. The same holds for the tree structure that supports deletions.*

6 Constant Update Finger Search Trees

In this section we sketch an implementation of a worst-case constant update finger search tree in the pointer machine based on the structures given in Sections 3 and 4. In general we have considered two strategies to join the structures for deletions and insertions in a single data structure. The first one is to allow the existence of fusion groups and split groups simultaneously into the structure. However, with this approach the structure becomes much more complicated since we have to make a distinction between fusion components and split components. The second approach, which we analyze in this paper, is to define a hierarchy between split and fusion groups. As a result, components refer to update operations and not solely to insertions or deletions.

The hierarchy of objects that we use inside the tree structure consists of blocks, nodes, fusion groups and finally split groups and it is depicted in figure 6. In this way, each split group consists of fusion groups, which are further partitioned into nodes. Finally, nodes consist of blocks that group the child pointers. The idea of this approach is the following: the split group that resides at the root of the component is first split into singleton split groups. A singleton split group contains only one fusion group (as in insertions only case where each singleton split group contains only one node). In the extreme case where only insertions are allowed, the fusion groups will always consist of single nodes while in the case where only deletions are allowed, the split groups will always contain only one fusion group with a sufficient number of nodes as shown in Section 4. In the general case, the split groups will contain a number of fusion groups which further contain a number of nodes. Thus, intuitively

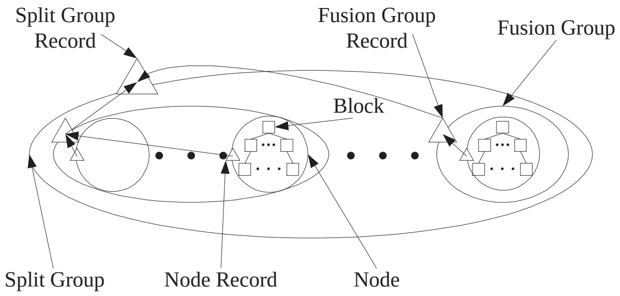

Figure 6: The hierarchy of objects in the tree structure is depicted (split groups consisting of fusion groups consisting of nodes consisting of blocks). This hierarchy is maintained by pointers between the different objects, which are stored in the respective records (eg. for a node the respective node record has a pointer field that points to the fusion group in which this node belongs).

we managed to have components handle both multiple splits and multiple fusions at the same time. This is achieved by applying the component mechanism only on split groups.

In Sections 3 and 4 certain invariants where applied on the size of the blocks and on the size of the nodes. These invariants must also hold in the general setting where insertions and deletions are allowed simultaneously. The sizes of fusion and split groups are derived based on the validity of the invariants given below.

Invariant 1 *The maximum out-degree of a block at level i is $a_i + 2 \cdot b_{i-1}$.*

Invariant 2 *The minimum out-degree of a block at level i is 2.*

Invariant 3 *Each node at level i has out-degree at least 2 and at most b_{i-1}^3.*

Invariant 4 *The fusion factor at level $i+1$ must be less than b_i ($a_{i+1} < b_i$).*

It is obvious that the initial state satisfies invariants 1 to 4. For the extreme cases also holds he same invariants. For example, if we consider deletions then the split group coincides with the fusion group and thus, invariants 1 to 4 hold by the discussion made in Section 4. In the case of insertions only, the fusion groups coincide with nodes and as a result invariants 1 to 4 hold by the discussion made in Section 3.

The main difficulty lies in the general setting where insertions and deletions are allowed simultaneously. This problem is related to the interaction of the mechanisms of deletions and insertions. First, we sketch the update algorithm, based on Figure 1. Assume that an update operation takes place at leaf l. The father f of the leaf l belongs in the component A, while the father (the split group) of $A.root$ belongs in the component B. We make a rebalancing operation at $A.root$, which assume that lies at level $i-1$. This rebalancing operation involves a break of the split group $A.root$ into singleton split groups (groups with one fusion group). At this point, if the father node at level i is large or small we respectively split it or fuse it with an adjacent node. Thus, we need to apply a size control mechanism for nodes. We can achieve this in a straightforward way by requiring that a node with out-degree less than a_i is *small* while a node with out-degree larger than b_{i-1}^3 is *large*. The result of a binary split (fusion) operation at level i is that an edge is inserted (deleted) at (from) level $i+1$. At this level we just have to update blocks in a manner similar to the one described in the previous sections. Below we sketch the main steps of the update algorithm at a leaf l (the algorithm is based on figure 1).

1. Find the component A in which the father of l belongs. Let new_sg be the spit group which is the root of A at level $i-1$.

2. Break A. Let B the component of its father.

3. Add(new_sg,B) and Break(new_sg). Let x_j be the node which is father of new_fg and let x_{j+1} be its brother node inside the pair or if it does not belong in the pair one of its adjacent nodes. Both nodes belong in the same fusion group fg at level i (note that fg is the real node while x_j and x_{j+1} are nodes which should be fused).

4. If x_j is small then start an incremental fusion of x_j and x_{j+1}. If x_j is a big node then split x_j.

5. If $out\_degree(fg) \leq a_i$ then fuse an adjacent fusion group with fg and fuse the respective adjacent blocks at level $i+1$ (all these fusions are incremental). The split groups should be reorganized accordingly if affected by this operation.

In step 5 of the algorithm split groups are affected by the fusion of adjacent fusion groups. If the split group is not singleton, that is it contains more than one fusion groups, then none split group is affected. However, if the split group is singleton then if in step 5 the unique fusion group needs to be fused with some other node then we put this singleton in an adjacent split group and make the incremental fusion.

The split and fuse operations applied on nodes are incrementally implemented by using the mechanism of pairs. In general, all size invariants are maintained by applying the same mechanisms as in Sections 3 and 4 with some minor modifications. Note that the incremental fusion and splits of nodes affects the fusion and split groups in the same way as described in Sections 4 and 3 respectively. In addition, fusions of nodes does not affect the split groups. However, it is necessary to introduce the split operation (apart from the fuse operation) in fusion groups due to splits of nodes. When a node is split into two nodes, it is necessary to split the fusion group into two fusion groups.

This can be acomplished in worst-case constant time if we build a structure on nodes similar to the structure of blocks that consist a single node. This three level structure will allow us to implement the split of a fusion group in worst-case constant time in a similar way to the split of nodes. It is easy to maintain a balanced partition (with respect to the out-degree of the nodes) of these nodes since their out-degree is increased or decreased by one (the insertion or deletion of an edge respectively).

The following theorem summarizes the result given in this paper:

Theorem 4 *We can maintain a finger search tree in the pointer machine model of computation such that finger searches are performed in worst-case optimal time $O(\log d)$ while updates are performed in constant worst-case time.*

7 Conclusions

In this paper we sketched 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 a multiple 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 interested to find applications of the component technique in other problems. Finally, we must note that several technical details were not included in this extended abstract but they will be considered in a future journal version.

References

[1] 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.

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

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

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

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

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

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

[15] 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.

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

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