Finger Search Trees with Constant Insertion Time

Gerth Stølting Brodal*
Max-Planck-Institut für Informatik
Im Stadtwald
D-66123 Saarbrücken
Germany

Email: brodal@mpi-sb.mpg.de

September 26, 1997

Abstract

We consider the problem of implementing finger search trees on the pointer machine, *i.e.*, how to maintain a sorted list such that searching for an element x, starting the search at any arbitrary element f in the list, only requires logarithmic time in the distance between x and f in the list.

We present the first pointer-based implementation of finger search trees allowing new elements to be inserted at any arbitrary position in the list in worst case constant time. Previously, the best known insertion time on the pointer machine was $O(\log^* n)$, where n is the total length of the list. On a unit-cost RAM, a constant insertion time has been achieved by Dietz and Raman by using standard techniques of packing small problem sizes into a constant number of machine words.

Deletion of a list element is supported in $O(\log^* n)$ time, which matches the previous best bounds. Our data structure requires linear space.

1 Introduction

A finger search tree is a data structure which stores a sorted list of elements in such a way that searches are fast in the vicinity of a *finger*, where a finger is a pointer to an arbitrary element of the list.

The list operations supported are the following. We let n denote the length of the involved list.

*Supported by the Carlsberg foundation (Grant No. 96-0302/20). Partially supported by the ESPRIT Long Term Research Program of the EU under contract No. 20244 (ALCOM-IT).

- CREATELIST creates a new list only containing one element, say, $-\infty$. A finger to the single element is returned.

- SEARCH(f, x) searches for element x, starting the search at the element of the list given by the finger f. Returns a finger to x if x is contained in the list, otherwise a finger to the largest element less than x in the list.

- INSERT(f, x) inserts element x immediately to the right of the finger f. Returns a finger to x.

- DELETE(f) deletes the element pointed to by the finger f.

Brown and Tarjan [2] observed that by *level-linking* $(2,4)$–trees, finger searches can be done in worst case $O(\log \delta)$ time, where δ is the distance between the finger and the search element. The distance between two elements is the difference between their ranks in the list. In the following, we denote a data structure having $O(\log \delta)$ search time a finger search tree. Huddleston and Mehlhorn [10] showed that $(2,4)$–trees support insertions and deletions in amortized constant time, assuming that the position of the element to be inserted or deleted is known.

The question we consider is, if it is possible to remove the amortization from the result of Huddleston and Mehlhorn [10], *i.e.*, if finger search trees exist which support insertions and deletions in worst case constant time.

By assuming a unit-cost RAM, Dietz and Raman [3] have presented a finger search tree implementation supporting insertions and deletions in worst case constant time. The data structure of Dietz and Raman is based on standard RAM techniques of packing small problem sizes into a constant number of machine words. For the weaker pointer machine model no similar result is known. The results obtained for the pointer machine are as follows.

Search trees with constant insertion and deletion time on the pointer machine have been presented by Levcopoulos and Overmars [13] and Fleischer [6], but neither of them support finger searches.

Finger search trees with worst case constant insertion and deletion time for the restricted case where there are only a constant number of *fixed* fingers have been given by Guibas *et al.* [7], Kosaraju [12] and Tsakalidis [17].

Finger search trees which allow any element of the list to be a finger and which obtain worst case $O(\log^* n)$ insertion and deletion time have been given by Harel and Lueker [8, 9].

In this paper we present the first finger search tree implementation for the pointer machine which supports arbitrary finger searches and which supports insertions in worst case constant time. The data structure can be extended to support deletions in worst case $O(\log^* n)$ time which matches the previous best bounds of Harel and Lueker [8, 9]. The space requirement for the data structure is $O(n)$.

The technical contribution of this paper is a new approach to select the nodes to split in a search tree. The previous approaches by Levcopoulos and Overmars [13] and Dietz and Raman [3] were based on a *global* splitting lemma which guaranteed that each of the recursive substructures would have polylogarithmic size. For a detailed discussion and applications of this lemma we refer to the thesis of Raman [16]. Our approach is, in contrast, a local bottom-up approach based on a functional implementation of binary counting to select the nodes to split in a search tree. A weakly related bottom-up approach has been presented by Brodal [1] to remove the amortization from the partial persistence technique of Driscoll *et al.* [5].

The structure of this paper is as follows. Section 2 describes the basic idea of the construction, Section 3 describes how to maintain ancestor pointers in a tree by using a functional stack implementation, and Section 4 describes how to achieve constant time splitting of nodes of arbitrary degree. How to support finger searches is described in Section 5. In Section 6 the data structure is extended to support deletions in worst case $O(\log^* n)$ time. In Section 7 we describe how to make the space requirement linear. Finally some concluding remarks are given in Section 8.

2 A new splitting algorithm

In this section we present a new algorithm for splitting nodes in a search tree when new leaves are created. The trees generated by this algorithm do not have logarithmic height and do not support insertions in worst case constant time, but the algorithm presented is the essential new idea required to obtain the claimed result.

Throughout this paper we implicitly assume that each node in a search tree has associated the interval of elements spanned by the node. This is standard for all search tree implementations and we therefore take this as an implicit assumption for the remaining of this paper.

In this section we assume that the ancestor of height d of a leaf can be found in worst case constant time, and that a node of arbitrary degree can be split in worst case constant time. In Section 3 and Section 4 we show how to avoid these assumptions, and in Section 5 we show how to extend the data structure to support finger searches.

In the following, T is a tree where all leaves have equal depth. We define leaves to have height zero, the parents of the leaves to have height one, and so on. To each leaf $\ell \in T$ we associate a counter $c_\ell \geq 0$. Initially the tree consists only of a single leaf storing the element $-\infty$ and having a counter equal to zero, and the parent node of the leaf.

Let $\Delta_1, \Delta_2, \ldots$ be a list of integers satisfying $\Delta_d \geq 2^{2^d} - 1$. Theorem 1 gives the resulting relation between Δ_d and the degrees of the nodes of height d.

The implementation of the insertion of an element e into the list next to a finger f is as follows. Let ℓ denote the leaf given by the finger f, and p the parent of ℓ. First we create a new leaf ℓ' storing the new element e below p and to the right of ℓ. Next we increment c_ℓ by one and assign the resulting value to $c_{\ell'}$ too. Let d be the unique value satisfying

$$c_\ell \bmod 2^d = 2^{d-1},$$

i.e., d is the position of the rightmost bit equal to one in the binary representation of c_ℓ. Let v be the ancestor of ℓ and ℓ' of height d. The third and last step we perform is to split v, provided the degree of v is at least $2\Delta_d$. We assume v is split into two nodes v' and v'', each having a degree of at least Δ_d. If we split the root, we increase the height of the tree by creating a new root of degree two.

Theorem 1 *The above algorithm guarantees that all nodes of height d have degree at most $2^{2\cdot 2^d}\Delta_d$ and at least Δ_d, except for the root which only has degree at least two.*

Proof Essential to the proof is the following notion of *potential*. We define the potential of a leaf ℓ with respect to height d as:

$$\Phi_\ell^d = 2^{2^d-1-((c_\ell+2^{d-1}) \bmod 2^d)}.$$

Notice that $1 \leq \Phi_\ell^d \leq 2^{2^d-1}$. If v is an internal node of T of height d, we let T_v^d denote the subtree rooted at v and $|T_v^d|$ the number of leaves in T_v^d. We define the potential of T_v^d to be the sum of the potentials of the leaves in T_v^d with respect to height d, *i.e.*,

$$\Phi_v^d = \sum_{\ell \in T_v^d} \Phi_\ell^d.$$

We now consider how an insert operation in the subtree T_v^d affects Φ_v^d. Let d' denote the height of the node to be split. If $d' \neq d$, then $c_\ell \bmod 2^d \neq 2^{d-1}$ and by increasing c_ℓ by one the value of Φ_ℓ^d is halved. We conclude that the new value of $\Phi_\ell^d + \Phi_{\ell'}^d$ equals the old value of Φ_ℓ^d, and Φ_v^d remains unchanged. Otherwise $d' = d$, then the old value of Φ_ℓ^d is one and the new values of Φ_ℓ^d and $\Phi_{\ell'}^d$ are 2^{2^d-1}. We conclude that Φ_v^d increases by $2 \cdot 2^{2^d-1} - 1 = 2^{2^d} - 1$ before we try to split v.

By induction we now prove that for all heights d and nodes v of height d,

$$\Phi_v^d \leq 2^{2\cdot 2^d} \prod_{i=1}^{d} \Delta_i. \tag{1}$$

Initially the tree consists only of a single leaf with a counter equal to zero and the parent of the leaf as the single internal node. The potential of the single internal node is two and it follows that (1) is true for the initial tree.

As argued above, the only node which increases its potential when creating a new leaf is the node v of height d which is the candidate to be split. Recall that Φ_v^d increases by $2^{2^d} - 1$. If v is split into two nodes, v' and v'', then each of the two nodes have a degree of at least $\Delta_d \geq 2^{2^d} - 1$, and therefore also potential of at least $2^{2^d} - 1$. We conclude that

$$\Phi_{v'}^d \leq \Phi_v^d + 2^{2^d} - 1 - \Phi_{v''}^d \leq \Phi_v^d \leq 2^{2\cdot 2^d} \prod_{i=1}^{d} \Delta_i,$$

and similarly for $\Phi_{v''}^d$, and (1) is satisfied.

If v is not split we first consider the case $d = 1$. Then the degree of v is at most $2\Delta_1 - 1$, implying $\Phi_v^1 \leq 4\Delta_1 - 2 \leq 2^{2\cdot 2^1}\Delta_1$ and (1) is satisfied. Otherwise $d > 1$ and we have

$$\Phi_v^d \leq 2\Delta_d 2^{2\cdot 2^{d-1}} \prod_{i=1}^{d-1} \Delta_i 2^{2^d-1} \leq 2^{2\cdot 2^d} \prod_{i=1}^{d} \Delta_i,$$

because v has a degree of less than $2\Delta_d$, and each child of v spans at most $2^{2\cdot 2^{d-1}} \prod_{i=1}^{d-1} \Delta_i$ leaves (by the induction hypothesis (1) and each leaf has a potential of at least one with respect to height $d-1$), and each leaf has a potential of at most 2^{2^d-1} with respect to height d. We conclude that (1) is satisfied, and is indeed an invariant.

Because a node at level d is first split when the node has degree $2\Delta_d$ it follows that all nodes of height d have a degree of at least Δ_d (except for the root), implying that all nodes v (except for the root) satisfy $|T_v^d| \geq \prod_{i=1}^{d} \Delta_i$.

Together with (1) we get the result that the degree of a node v of height d is at most $\Phi_v^d / \prod_{i=1}^{d-1} \Delta_i \leq 2^{2 \cdot 2^d} \Delta_d$, and the theorem follows. □

Corollary 1 *If $\Delta_d = 2^{2^d}$ the algorithm maintains a tree of height $\log\log n - O(1)$ where all nodes of height d have degree $O(2^{3 \cdot 2^d})$.*

3 Maintaining pointers to ancestors

One of the main difficulties in giving an efficient implementation of the algorithm described in Section 2 is that we cannot find the level d ancestor of leaf ℓ that should be split in worst case constant time. In this section we describe how to solve this problem so that we can find the ancestor in constant time while still having constant insertion time, assuming that we can split a node of arbitrary degree in constant time. How to remove the assumption about splitting is postponed to Section 4.

The basic idea is to extend the information stored at each leaf so that in addition to the counter c_ℓ we also store a pointer to each of the $\log\log n$ ancestors of ℓ. In fact we only store a relevant subset of the pointers. The details are as follows.

With leaf ℓ we store a stack S_ℓ of triples (i, j, u_j) where $i \leq j$ are positions in the binary representation of c_ℓ and u_j is a pointer to the ancestor of ℓ of height $j+1$,[1] so that the triples on S_ℓ represents the intervals $[i, j]$ of positions in the binary representation of c_ℓ all containing a one. If $c_\ell = 001110011010_2$ then $S_\ell = (1, 1, \cdot), (3, 4, \cdot), (7, 9, \cdot)$. To clarify this, we require all intervals to be disjoint, nonadjacent, sorted with respect to i, and their union to be exactly the set of positions in the binary representation of c_ℓ which equals one.

Because S_ℓ implicitly represents the value of c_ℓ we do not need to store c_ℓ. In the following we let c_ℓ refer to the value implicitly represented by S_ℓ.

An important detail of the algorithm described in the previous section is that when creating leaf ℓ', we assign $c_{\ell'}$ the new value of c_ℓ. Similarly we now assign $S_{\ell'}$ the stack S_ℓ. To avoid copying the whole stack (which would require $\Theta(\log\log n)$ time), we implement the stacks S_ℓ as *purely functional* stacks. A purely functional stack is just a standard LISP list. This allows us to assign $S_{\ell'}$ the value of S_ℓ in worst case constant time. Recent work on functional data structures can be found in [11, 14].

We now describe how to update S_ℓ corresponding to incrementing c_ℓ and how to determine the node v at level d that should be split. In the following, p_x denotes the parent of the leaf or internal node x. If S_ℓ is empty, we just push $(0, 0, p_\ell)$ onto S_ℓ. Otherwise let (i, j, u_j) denote the triple at the top of S_ℓ. If $i \geq 1$ we push $(0, 0, p_\ell)$ onto S_ℓ, otherwise $i = 0$ and we replace the top triple of S_ℓ by $(j+1, j+1, p_{u_j})$. The node v to split is now the last field in

[1] Due to the splitting of ancestors of ℓ, u_j can also point to a node of height $j+1$ which is not an ancestor of ℓ, but this turns out to be a minor problem.

the top triple of S_ℓ. Finally we check if the two top triples of S_ℓ have become adjacent, *i.e.*, if they are of the form $(i,k,\cdot)$ and $(k+1,j,u_j)$ in which case the two triples are replaced by (i,j,u_j).

The correctness of the implementation with respect to i and j is obvious, because it is just binary counting. The interesting property is how we handle the pointers to the nodes u_j. If after updating S_ℓ, $c_\ell \bmod 2 = 1$ then the node returned is the correct node p_ℓ and the only pointer which can be added to the stack is p_ℓ. If $c_\ell \bmod 2^d = 2^{d-1}$ for $d > 1$, then the returned and only new node on the stack is $p_{u_{d-1}}$ which is exactly the ancestor of ℓ of height $d+1$ — provided that before updating S_ℓ, u_{d-1} is in fact the ancestor of ℓ of height d.

If no nodes were ever split, the above argument could be used to give an inductive argument that a u_j stored in a stack S_ℓ would always be the ancestor of ℓ of height $j+1$. Unfortunately we split nodes, and cannot guarantee that this property is satisfied (at least not without doing a nontrivial update of a nonconstant number of purely functional S_ℓ stacks when doing a split). In the following we argue that we do *not* need to care about "wrong" pointers, provided that splitting a node does not introduce too many wrong pointers.

The insertion algorithm is now the following. First we update in constant time the set S_ℓ as described above. Let v be the node of height d which is returned to be split. We then create the new leaf ℓ' and assign S_ℓ to $S_{\ell'}$ in constant time. If the degree of v is $\geq 2\Delta_d$ we split v into two nodes as follows. First we create a new node v' to the right of v with $p_{v'} = p_v$, and then we move the rightmost Δ_d children of v to v'. It is essential to the algorithm that splittings are done nonsymmetrically. The details of how to perform a split in worst case constant time is described in Section 4.

In the following we prove that this modified algorithm basically achieves the same bounds on the degrees of the nodes as the algorithm in Section 2.

Theorem 2 *The above algorithm guarantees that all nodes of height d have a degree of at most $2^{3\cdot 2^d}\Delta_d$ and at least Δ_d, except for the root.*

Proof The proof is basically the same as for Theorem 1, except that we now have to incorporate the "wrong" pointers into the potentials.

Let ℓ be a leaf, d a fixed height, and v the ancestor of ℓ of height d. If $1 \leq c_\ell \bmod 2^d < 2^{d-1}$, let u_j be given by $j = \max\{j' < d | (\cdot, j', \cdot) \in S_\ell\}$ and $(\cdot, j, u_j) \in S_\ell$.

We now define the potential Φ_ℓ^d of ℓ with respect to height d. The potential is basically equal to two raised to the number of times we have to increment c_ℓ before we split v.

$$\Phi_\ell^d = \begin{cases} 2^{2^d+2^{d-1}-1-(c_\ell \bmod 2^d)} & \text{if } (1 \leq c_\ell \bmod 2^d < 2^{d-1}) \wedge (u_j \notin T_v^d), \\ 2^{2^d+2^{d-1}-1-(c_\ell \bmod 2^d)} & \text{if } 2^{d-1} \leq c_\ell \bmod 2^d, \\ 2^{2^{d-1}-1} & \text{if } c_\ell \bmod 2^d = 0, \\ 2^{2^{d-1}-1-(c_\ell \bmod 2^d)} & \text{if } (1 \leq c_\ell \bmod 2^d < 2^{d-1}) \wedge (u_j \in T_v^d). \end{cases} \qquad (2)$$

Notice that $1 \leq \Phi_\ell^d \leq 2^{2^d+2^{d-1}-2}$. We similarly define the potential of T_v^d to be

$$\Phi_v^d = \sum_{\ell \in T_v^d} \Phi_\ell^d.$$

We now show that incrementing c_ℓ by updating S_ℓ we either are allowed to split v or Φ_ℓ^d is halved. We do this by considering each of the cases in (2).

First we consider the case where $1 \le c_\ell \bmod 2^d < 2^{d-1}$ and $u_j \notin T_v^d$. We split this into two cases. If $c_\ell \bmod 2^d = 2^{d-1}-1$, then the new value of $c_\ell \bmod 2^d = 2^{d-1}$ and Φ_ℓ^d is halved. If $c_\ell \bmod 2^d < 2^{d-1}-1$, then c_ℓ is increased by one and u_j remains on the stack S_ℓ or is replaced by $p_{u_j} \notin T_v^d$ implying that Φ_ℓ^d is halved.

If $2^{d-1} \le c_\ell \bmod 2^d$, then we consider two cases. If $c_\ell \bmod 2^d < 2^d-1$, then Φ_ℓ^d is halved. If $c_\ell \bmod 2^d = 2^d-1$, then the new value of $c_\ell \bmod 2^d = 0$ and again Φ_ℓ^d is halved.

If $c_\ell \bmod 2^d = 0$, then the new value of $c_\ell \bmod 2^d = 1$ and $u_j = p_\ell \in T_v^d$ and the value of Φ_ℓ^d is halved.

The last case to be considered is where $1 \le c_\ell \bmod 2^d < 2^{d-1}$ and $u_j \in T_v^d$. If $c_\ell \bmod 2^d = 2^{d-1}-1$ then the new value of $c_\ell \bmod 2^d = 2^{d-1}$ and the node we split is $p_{u_j} = p_{u_{d-2}} = v$. The new potential of $\Phi_\ell^d = 2^{2^d-1}$. Finally if $1 \le c_\ell \bmod 2^d < 2^{d-1}-1$, then c_ℓ is increased by one and u_j remains on the stack S_ℓ or is replaced by $p_{u_j} \in T_v^d$ implying that Φ_ℓ^d is halved.

We conclude that incrementing c_ℓ either halves Φ_ℓ^d and a node different from v is to be split, or Φ_ℓ^d changes from one to 2^{2^d-1} and v is the node to be split. This is exactly the same statement as in the proof of Theorem 1, except that we now use different potentials.

We now make the observation that when an insertion creates a new leaf ℓ' next to ℓ after having incremented c_ℓ, then $\Phi_{\ell'}^d = \Phi_\ell^d$ and Φ_v^d does not change for any d and node v at level d, except for the node to be split which increases its potential by at most $2^{2^d}-1$.

We now give an inductive argument that

$$\Phi_v^d \le 2^{3\cdot 2^d} \prod_{i=1}^{d} \Delta_i. \tag{3}$$

But first we have to observe that a u_j pointer at leaf ℓ either points to the ancestor v of ℓ of height $j+1$ or is a node of height $j+1$ to the *left* of v. This is true because whenever a node is split the new internal node is created to the right of the old node.

The above observation implies that when splitting node v, no leaf in T_v^d points to a node in $T_{v'}^d$ and therefore no leaf in T_v^d changes its potential with respect to height d when splitting v, but for the leaves in $T_{v'}^d$ this is not true. No potential with respect to heights different from d changes due to the splitting.

That (3) is true for the initial tree is obvious. We know from the above arguments that the only potential that can change due to incrementing c_ℓ and adding a new leaf ℓ' is the node v of height d that is to be split.

If v has degree less than $2\Delta_d$ then we do not split v, and

$$\Phi_v^d \le 2\Delta_d 2^{3\cdot 2^{d-1}} \prod_{i=1}^{d-1} \Delta_i 2^{2^d+2^{d-1}-2} \le 2^{3\cdot 2^d} \prod_{i=1}^{d} \Delta_i,$$

because v has at most $2\Delta_d$ children each spanning $2^{3\cdot 2^{d-1}} \prod_{i=1}^{d-1} \Delta_i$ leaves (by the induction hypothesis), and each leaf has a potential of at most $2^{2^d+2^{d-1}-2}$ with respect to height d.

If v is split, then the increase of Φ_v^d due to the increase of c_ℓ and the leaf ℓ' is canceled out by the potential moved to v' of at least Δ_d, because each subtree has a potential of at least one. For the new vertex v' we have

$$\Phi^d_{v'} \leq \Delta_d 2^{3\cdot 2^{d-1}} \prod_{i=1}^{d-1} \Delta_i 2^{2^d+2^{d-1}-2} \leq 2^{3\cdot 2^d} \prod_{i=1}^{d} \Delta_i. \tag{4}$$

We conclude that (3) is satisfied, and is indeed an invariant. From (3) and that $|T^d_v| \geq \prod_{i=1}^d \Delta_i$, for v different from the root, the theorem follows. □

4 Incremental node splitting

The basic assumption in the previous section was that we could split a node v of arbitrary degree in constant time. In this section we show how to achieve this by basically maintaining the parent pointers as trees of height two.

We let all children of node v be maintained in a double linked list. Instead of letting all children have parent pointers directly to v, we introduce an intermediate level of indirection. We partition the children of v into blocks of size at least Δ_d and at most $2\Delta_d - 1$, such that there is one node in the intermediate level for each of the blocks. In the following the nodes in the intermediate level are denoted intermediate nodes.

The information maintained at each of the above mentioned nodes is the following. At v we just maintain a pointer to the leftmost and rightmost intermediate node below v. The children maintain pointers to their left and right sibling and a pointer to the intermediate node corresponding to the block the child belongs to. An intermediate node maintains a pointer to v, and pointers to the leftmost and rightmost child of v in the block spanned by the intermediate node.

Whenever a child u of v is split, we add the new child u' next to u in the double link list of children of v and let it belong to the same block as u. To avoid having too many children belong to the same block, which would imply that the block should be split, we do the splitting of the block incrementally as follows. Whenever an intermediate node w spans more than Δ_d children, we instead represent w by a *pair* of nodes w' and w'' such that w' spans the leftmost Δ_d children and w'' spans the remaining at most $\Delta_d - 1$ children. The additional information we associate with each intermediate node to achieve this is the number of children spanned by each intermediate node, and if a node is part of a pair, a pointer to the other node in the pair. The number of children spanned by an intermediate node immediately reveals whether the node is the left or right node in the pair. See Figure 1.

Whenever a new leaf is added to the block spanned by w we check if w is part of a pair. If w is not part of a pair, then w now has degree $\Delta_d + 1$. To satisfy the above constraints, we create a new intermediate node w' that, together with w, make a pair, and move the rightmost child of w to w' by appropriately updating the pointers. If w is part of a pair we check if w is the left node of the pair. If w is the left node, then w now spans $\Delta_d + 1$ children and we move the rightmost child of w to the other node of the pair to satisfy the condition that w has degree Δ_d. If both nodes of the pair now have degree Δ_d (the initial degree bound of $2\Delta_d - 1$ is violated) we split the pair by simply setting the two pair pointers to nil. The above updating when v gets a new child can clearly be done in worst case constant time.

We now describe how the above substructure can be used to solve the splitting problem of the algorithm in Section 3. The algorithm is exactly the same as in Section 3, except for

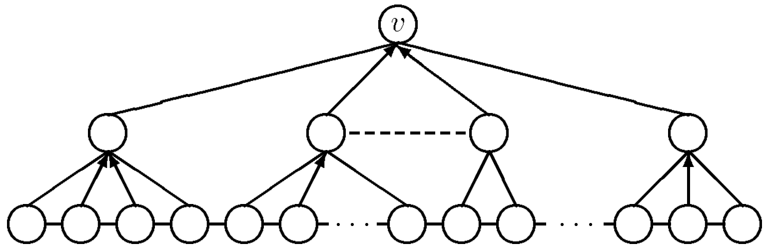

Figure 1: The implementation of the children of v. Undirected edges represent pointers in both directions. The dashed edge is the pair pointers of two intermediate nodes.

the constraints on how to split node v. The original constraint was that we split v if it has a degree of at least $2\Delta_d$ and that the new node v' has a degree equal to Δ_d.

We replace this by the following. We split v if it spans at least two intermediate nodes not belonging to the same pair (which is exactly the same as requiring that v should have a degree of at least $2\Delta_d$). We split v by first creating a new node v' to the right of v (in worst case constant time as described above), and then by moving the rightmost intermediate node of v to v'. If the intermediate node is part of a pair we move both nodes of the pair to v'. The degree of v' after the splitting is at least Δ_d and at most $2\Delta_d - 1$.

That the behavior of the algorithm remains the same is captured by the following theorem.

Theorem 3 *The above algorithm guarantees that all nodes of height d have degree at most $2^{3\cdot 2^d}\Delta_d$ and at least Δ_d, except for the root. New leaves can be created in worst case constant time.*

Proof The proof is exactly the same as for Theorem 2, except for (4) which is replaced by (5) below. Because v' after splitting has a degree of at most $2\Delta_d - 1$, we get

$$\Phi^d_{v'} \leq (2\Delta_d - 1)2^{3\cdot 2^{d-1}} \prod_{i=1}^{d-1} \Delta_i 2^{2^d + 2^{d-1} - 2} \leq 2^{3\cdot 2^d} \prod_{i=1}^{d} \Delta_i. \tag{5}$$

The time bound for updating the tree follows immediately from the previous discussions, and the time for updating the S_ℓ stacks only increases by a constant factor due to the introduced level of indirection. □

5 A semi-dynamic finger search tree

We now describe how the data structure of Section 4 can be extended to support finger searches. In this section we assume $\Delta_d = 2^{2^d}$. The basic idea is to replace each node of the tree in Section 4 by a balanced tree allowing constant time updates. By appropriately level linking the resulting data structure we get a finger search tree that supports insertions in worst case constant time.

By level linking [10] the search tree of Section 2 a finger search for element x starting at leaf ℓ can be done as follows, which is basically the same as in [10]. We without loss of generality assume x is contained in the tree. Notice that level linking does not introduce any new data fields in the nodes, because each node already stores a pointer to its right and left sibling. We just have to maintain the corresponding pointers for the leftmost and rightmost child of a node too.

If x is contained in a neighbor leaf of ℓ we are done. Otherwise we look at the parent p of ℓ. If x is contained in the subtree rooted at p or one of the neighbors of p we search for x in the corresponding subtree, otherwise we recursively consider the parent of p.

Before giving the details of how to search for x in a subtree of height d we give a lower bound for the distance between x and ℓ. If we are going to search for x in a subtree rooted at node v of height d, we know that x is not contained in the subtree below v containing ℓ or the neighbor subtrees. By Theorem 3 we have that the distance between ℓ and x is at least $2^{2^{d-1}}$. We conclude that we can use $O(2^d)$ time for the search for x.

If we could search for which subtree rooted at a child of v contained x in time logarithmic in the degree of v, we could recursively find x in time

$$\sum_{i=1}^{d} \log 2^{3\cdot 2^i + 2^i} = \sum_{i=1}^{d} 2^{i+2} \leq 2^{d+3} = O(2^d).$$

To achieve the logarithmic search time we add the following structure to the data structure of Section 4. With v we associate a search tree which stores each of v's intermediate nodes, and with each of the intermediate nodes we associate a search tree which stores the children of v. By choosing the search trees of Levcopoulos and Overmars [13] or Fleischer [6] we can add and remove new leaves to these search trees in worst case constant time, implying that the overhead introduced for splitting a node as described in Section 4 is only a constant.

To summarize we get the following theorem.

Theorem 4 *There exists a pointer-based implementation of finger search trees which supports arbitrary finger searches in $O(\log \delta)$ time and neighbor insertions in worst case constant time.*

6 Deletions

In the following we describe how to extend the data structure of the previous sections to support deletions in worst case $O(\log^* n)$ time. We basically implement delete as for (a, b)–trees by performing a sequence of fusion and sharing steps [10]. Due to the ancestor pointers introduced in Section 3, fusion and sharing steps need to be implemented carefully to guarantee that the potentials Φ_v^d remain bounded.

The first step towards achieving $O(\log^* n)$ deletion time is to decrease the height of the tree to $O(\log^* n)$. Let $2^{(d)}$ recursively be given by $2^{(1)} = 2$ and $2^{(d+1)} = 2^{2^{(d)}}$. By letting $\Delta_d = 2^{(d)}$, it follows by Theorem 3 that the resulting tree of Section 4 has a height of $O(\log^* n)$ and that new leaves can be added in worst case constant time. In the following we first describe how to support deletions in worst case $O(\log^* n)$ time and then how to support finger searches in worst case $O(\log \delta)$ time (for the finger search implementation presented in Section 5 it is crucial that $\Delta_d = 2^{2^d}$).

The basic idea of how to delete a leaf ℓ is as follows. First the leaf ℓ is deleted. If the parent v of the leaf ℓ has at least Δ_1 children left we are done. Otherwise we fuse v with the left or right sibling v' of v, by moving the children of v to v' and removing the node v. If v' now has too large a degree we split v' by creating a new node v'' to the right of v and moving a fraction of the children of v' to v''.[2] We postpone the exact thresholds to the discussion below. Otherwise p_v has lost one child and we recursively fuse p_v if it has obtained too low a degree.

There are two problems which should be considered when implementing deletions as outlined above.

The first involves the ancestor stacks stored at the leaves. Assume v is fused with v', and v is removed from the child list of p_v. Unfortunately many leaves can have ancestor pointers stored in their S_ℓ stacks pointing to v, and we cannot afford to update all these pointers. And it is even more complicated because a pointer to v from a leaf ℓ can be the essential u_j pointer in the potential definition (2) of ℓ with respect to a height larger than the height of v.

Our solution is very simple. We just let v become a *dead* child of p_v. For a dead child of height d we only maintain a pointer from the child to its parent of height $d+1$. No pointer from the parent to the child is required. A dead child is never moved to another node, and a node can have an arbitrary number of dead children. The parent of a dead child can also be dead (due to a fusion step).

Because of the parent pointers of the dead nodes, a dead node u_j of height $j+1$ in a natural way belongs to a subtree T_v^d if and only if there is an ancestor path from u_j to v. This allows us to define the potential of a leaf ℓ with respect to height d as given by (2) in Theorem 2 and to replace u_j by p_{u_j} on a S_ℓ stack when incrementing c_ℓ.

The second problem to be considered is the change in the potential of $\Phi_{v'}^d$ when we fuse v with v'. We fuse v with v' if the degree of v becomes $\Delta_d - 1$, implying that the potential $\Phi_{v'}^d$ increases. If v' now has too large a degree, we split v' to insure that the children moved from v' to the new node v'' cancel out the increase in potential. Unfortunately it is not sufficient to move $\Theta(\Delta_d)$ children to v'', because the children we add below v' can have high potential whereas the children we remove below v' can have low potential. Let $\Gamma_d = 2^{3(d-1)2^{d-1}} \cdot 2^{2^d+2^{d-1}-2}$. It turns out that if we move at least $\Gamma_d \cdot \Delta_d$ children to v'', the potential of v' is guaranteed not to increase.[3]

To support the splitting of nodes in worst case constant time, we introduce an additional intermediate level at each node v, such that the intermediate nodes introduced in Section 4 (of degree at least Δ_d and at most $2\Delta_d - 1$) are partitioned into blocks of size at least Γ_d and at most $2\Gamma_d - 1$ (provided that there are at least Γ_d intermediate nodes). The additional intermediate level only increases the cost of finding a parent node p_{u_j} by a constant.

Each node of the original intermediate level, in the following referred to as intermediate level 1, points to a node in the new intermediate level, intermediate level 2. Nodes in intermediate level 2 point to v and the leftmost and rightmost node in the corresponding

[2] Intuitively we should move one child of v' to v, but this does not work due to the ancestor pointers introduced in Section 3.

[3] Γ_d is the maximum potential of a node of height $d-1$ divided by $\prod_{i=1}^{d-1} \Delta_i$, times the maximum potential of a leaf with respect to to height d.

intermediate level 1 blocks. If a block at intermediate level 2 has a size larger than Γ_d we similarly to the intermediate level 1 represent the block by a pair of nodes to support incremental splitting and fusion of intermediate level 2 nodes.

If a intermediate level 1 block is of size $\Delta_d - 1$ we consider fusing the block with a neighbor block. If the neighbor block has is larger than Δ_d we just move one child of the neighbor block to the block and are done. Otherwise we fuse the two blocks to a pair of size $2\Delta_d - 1$. If the corresponding intermediate level 2 block now is of size $\Gamma_d - 1$ we similarly fuse the intermediate level 2 block with a neighbor block (if a level 2 neighbor block exists). The necessary pointer updating is straightforward.

The implementation of insert remains unchanged, except that nodes are first split when there are two level 2 blocks, implying that a node not split can have degree $(2\Delta_d - 1)(2\Gamma_d - 1)$. Dead nodes are never split. When splitting a node v we now just move the rightmost intermediate level 2 block to the new node.

We are now ready to give the remaining details of how to perform deletions in worst case $O(\log^* n)$ time. If a node v different from the root reaches degree $\Delta_d - 1$ we move all the children of v to one of its neighbor siblings. Let v' denote this sibling. Because v is of degree $\Delta_d - 1$ all children of v belong to a single intermediate level 1 block. So we just have to move this block to v' and fuse it with a intermediate level 1 neighbor block as described above. This can clearly be done in worst case constant time. The node v becomes a dead child of p_v. If v' now has at least two level 2 blocks we split v' by creating a new node v'' to the right of v' and move the rightmost level 2 block of v' to v''. Otherwise we recursively consider the parent p_v of v which has lost one child.

Because we always fuse a node when it has a degree of less than Δ_d and always split a node into two nodes of a degree of at least Δ_d, the above algorithm guarantees that all nodes of height d (except for the root) have a degree of at least Δ_d, and therefore span at least $\prod_{i=1}^{d} \Delta_i$ leaves. Because delete spends only constant time for each height we get the result that delete can be implemented in worst case $O(\log^* n)$ time.

Theorem 5 *The above algorithm guarantees that all nodes of height d have degree at most $2^{3d2^d}\Delta_d$ and at least Δ_d, except for the root. New leaves can be created in worst case constant time and existing leaves can be deleted in worst case $O(\log^* n)$ time.*

Proof The time bounds and the lower bound on the degrees follow immediately from the previous discussion. What remains to be shown is the upper bound on the degrees.

Let Φ_ℓ^d and Φ_v^d be defined as in Theorem 2. We are going to prove that the potentials of the nodes are bounded by

$$\Phi_v^d \leq 2^{3d2^d} \prod_{i=1}^{d} \Delta_i. \tag{6}$$

That the initial configuration satisfies (6) is obvious. We first consider inserting a new leaf ℓ' next to a leaf ℓ. When incrementing the c_ℓ counter by updating a S_ℓ stack and adding the new leaf ℓ' it follows as for Theorem 2 that no potentials change except for the node at level d that is going to be split. This is true because if $u_j \neq v$, then $u_j \in T_v$ if and only if $p_{u_j} \in T_v$ — also if u_j refers to a dead node.

If a node v cannot be split, then for $d = 1$ we have

$$\Phi_v^1 \leq (2\Delta_1 - 1)(2\Gamma_1 - 1)2^{2^1+2^{1-1}-2} \leq 2\Delta_1 \cdot 3 \cdot 2 \leq 2^{3\cdot 1\cdot 2^1}\Delta_1, \tag{7}$$

and for $d \geq 2$ we have

$$\Phi_v^d \leq (2\Delta_d - 1)(2\Gamma_d - 1)2^{3(d-1)2^{d-1}} \prod_{i=1}^{d-1} \Delta_i 2^{2^d+2^{d-1}-2}$$

$$\leq 2^{3(d-1)2^{d-1}+2^d+2^{d-1}-2+3(d-1)2^{d-1}+2^d+2^{d-1}} \prod_{i=1}^{d} \Delta_i \leq 2^{3d2^d} \prod_{i=1}^{d} \Delta_i. \tag{8}$$

If v is split it similarly follows that the new node v' satisfies (6). Because nodes of height d have a degree of at least Δ_d, v' spans at least $\Gamma_d \prod_{i=1}^d \Delta_i$ leaves.

Finally we have to consider the potential of v when v is split. We know that the potential of v can at most increase by $2^{2^d} - 1$ by the new leaf added, and that the potential moved to v' is at least $\Gamma_d \prod_{i=1}^d \Delta_i \geq 2^{2^d} - 1$. This guarantees that the potential Φ_v^d does not increase due to the insertion — provided that splitting v does not increase the potential of any leaf of T_v^d with respect to height d.

To guarantee this, we again need the observation that u_j stored at leaf ℓ points to the ancestor of ℓ of height $j+1$ or a node to the *left* of the ancestor of height $j+1$. This guarantees that no leaf in T_v^d maintains a pointer into the new subtree $T_{v'}^d$. Unfortunately a u_j pointer can point to a dead node, and dead nodes do not belong to the tree. By defining the dead children of a node to always be the leftmost children of the node (in any arbitrary order), the above constraint will be satisfied. This is true because splitting a node always moves the children to a new node to the right of the node. For deletions we only have to argue that when we fuse v with a sibling v' to the right or left of v, the constraint is also satisfied. When we fuse v and v' all leaves in T_v^d are moved to $T_{v'}^d$. But because v becomes a dead node we, by definition, let v (and its dead subtree) be a node to the left of v', implying that u_j pointers to v in $T_{v'}^d$ points to a node to the left of their level d ancestor. We conclude that (6) is true for insertions.

For deletions we have to argue that (6) is satisfied. Let v be a node we consider to fuse with v' because v gets degree $\Delta_d - 1$. This implies $\Phi_{v'}^d$ increases by at most

$$(\Delta_d - 1)2^{3(d-1)2^{d-1}} \prod_{i=1}^{d-1} \Delta_i 2^{2^d+2^{d-1}-2} \leq \Gamma_d \prod_{i=1}^{d} \Delta_i.$$

If v' is not split we know from (7) and (8) that v' satisfies (6). If v' is split we similarly know that v'' satisfies (6), and because v'' spans at least $\Gamma_d \prod_{i=1}^d \Delta_i$ leaves v' also satisfies (6).

From (6) we conclude that all nodes at height d have a degree of at most

$$\frac{2^{3d2^d} \prod_{i=1}^d \Delta_i}{\prod_{i=1}^{d-1} \Delta_i} \leq 2^{3d2^d} \Delta_d,$$

and the theorem follows. $\square$

Unfortunately the modified trees do not support finger searches as described in Section 5, because the degree of a node of height d is exponential in the maximum size of a child subtree root at height $d-1$. Except for the searching at each of the ancestor nodes of the finger f, the implementation of a finger search remains the same as described in Section 5.

We need the following lemma to achieve $O(\log \delta)$ time for finger searches.

Lemma 1 *There exists a pointer-based implementation of finger search trees which supports arbitrary finger searches in $O(\log\log n + \log \delta)$ time, and neighbor insertions and deletions in worst case constant time.*

Proof The lemma is obtained by combining the finger search trees of Dietz and Raman [3] and the search trees of Levcopoulos and Overmars [15].

The basic data structure of Dietz and Raman [3] is a $(2,3)$–tree where each leaf stores a bucket of $\Theta(\log^2 n)$ elements. By level-linking the $(2,3)$–tree a finger search can easily be done on this part of the data structure as described by Brown and Tarjan [2]. Dietz and Raman [3] implement the buckets by using the RAM model. This is the only part of their construction requiring the RAM. They show that, if buckets of size $O(\log^2 n)$ support insertions and deletions in worst case constant time and buckets can be split and joined in $O(\log n)$ time, it is possible to support leaf insertions and deletions in worst case constant time and finger searches in $O(\log \delta)$ time plus the time for a finger search in a bucket. Whereas Dietz and Raman support finger searches in a bucket in time $O(\log \delta)$ by using the RAM, we show how to obtain $O(\log\log n)$ time by using the weaker pointer machine.

Our bucket representation is quite similar to that of [3, 4, 13]. We represent a bucket by a tree of height two where all nodes of height one have a degree between $\log n$ and $2\log n - 1$. If a node of height one has a degree of at least $\log n + 1$ we, as with the intermediate nodes in Section 4, represent the node by a pair of nodes (see Figure 1). Adding or deleting a leaf is handled in a similar way a as for the intermediate nodes. By using the search trees of Levcopoulos and Overmars [13] to store the children of each node in a bucket, we can insert and delete leaves from a bucket of size $O(\log^2 n)$ in worst case constant time and support searches in worst case $O(\log\log n)$ time. A bucket can be split in worst case $O(\log n)$ time by simply incrementally moving $O(\log n)$ nodes of height one to a new bucket.

The lemma follows from [3]. $\Box$

If we represent each intermediate level 1 node of our data structure by the search tree of Lemma 1, a finger search can be implemented as follows.

The first search at node v of height d is performed as follows. If x is spanned by the same or a neighboring intermediate level 1 block of the block spanning f, we perform a finger search for the child of v spanning x in time at most $O(\log \delta + \log\log 2^{(d)})$ as described in Lemma 1. Otherwise $\delta \geq 2^{(d)}$ and we sequentially find the intermediate level 1 block spanning x and perform a search in this block. Because there are at most 2^{3d2^d} intermediate level 1 blocks this can be done in $O(2^{3d2^d} + \log 2^{(d)}) = O(\log \delta)$ time. We conclude that the search at height d can be performed in $O(\log \delta + \log\log 2^{(d)})$ time. For each recursive search we find the intermediate level 1 block spanning x sequentially as described above and perform a search in the block to find the child spanning x. For level i this requires $O(2^{3i2^i} + \log 2^{(i)})$ time.

The total time for a finger search therefore becomes

$$\log \delta + \log\log 2^{(d)} + \sum_{i=1}^{d-1} \left(2^{3i2^i} + \log 2^{(i)} \right) = O(\log \delta + \log\log 2^{(d)}) = O(\log \delta),$$

because $\delta \geq 2^{(d-1)}$.

We are now ready to state our main theorem.

Theorem 6 *There exists a pointer-based implementation of finger search trees which supports arbitrary finger searches in $O(\log \delta)$ time, neighbor insertions in worst case constant time, and deletions in worst case $O(\log^* n)$ time.*

7 Space requirement

In the previous sections we have not considered the space requirement of our data structure. It immediately follows that if only insertions are allowed, the data structure only requires linear space because each insertion only requires additional constant space. If deletions are allowed the space requirement can become nonlinear due to the dead nodes and the stacks stored at the leaves. Because deletions take $O(\log^* n)$ time each deletion only increases the space requirement by $O(\log^* n)$. In the following we describe how the space requirement of our data structure can be made linear by applying the global rebuilding technique of Overmars [15].

The details are as follows. Assume the finger search tree T at some time stores N elements. Throughout the next $\Theta(N)$ time (not operations) spent on updating T we incrementally build a new finger search tree T' storing the same elements as T. For each element in T we maintain a pointer to its position in both T and T'. An element not yet inserted into T' stores a null pointer. Initially T' is an empty finger search tree. We build T' by incrementally scanning through the list stored by T from left-to-right by having a pointer to the next element in T to be scanned. Whenever a new element is inserted into T we also insert the element into T' if the neighbor list elements have been inserted into T'. For each insertion we scan two elements of T and insert the elements into T' in constant time. For deletions we similarly delete the element from T' if the element already has been inserted into T'. For each deletion we scan $\max\{2, \log^* N\}$ elements of T and insert the elements into T' in $O(\log^* N)$ time. The time required for insertions and deletions only increases by a constant. After at most N insertions and $N/\log^* N$ deletions, in total requiring $\Theta(N)$ time and space, T and T' store the same set of elements, and we can discard the finger search tree T and let T' play the role of T.

The discarding of T can be done by applying standard incremental garbage collecting techniques, provided that no element in T' stores a pointer to its position in T. We therefore, before discarding T, perform a second scan through the elements in time $\Theta(N)$ as described above where we set all pointers into T to null. Throughout this scan updates and finger searches are only performed on T'.

Let N' denote the number of elements stored in T' when we discard T. The number of neighbor insertions done during the two scans is at most $3N$ and the number of deletions is at most $2N/\log^* N$. Because $N(1 - 2/\log^* N) \leq N' \leq 4N$ and there has been at most $2N/\log^* N$ deletions done on T', T' requires $O(N')$ space. By always starting a new rebuilding when the previous rebuilding is finished, it follows that the data structure requires linear space.

8 Conclusion

We have presented the first pointer-based finger search tree implementation allowing insertions to be done in worst case constant time. The previous best bounds were $O(\log^* n)$ [6, 8, 9].

It remains an open problem if our data structure can be extended to support deletions in worst case constant time too. Our data structure can be extended to support deletions in worst case $O(\log^* n)$ time, matching the bounds of Harel and Lueker [8, 9].

An interesting and related question to consider is if some of the presented ideas can be used to remove the amortization from the node splitting technique of Driscoll *et al.* [5] to make data structures fully persistent.

Acknowledgments

Thanks goes to Leszek Gąsieniec and Arne Andersson for patient listening, and Rudolf Fleischer for comments on the manuscript.

References

[1] Gerth Stølting Brodal. Partially persistent data structures of bounded degree with constant update time. *Nordic Journal of Computing*, 3(3):238–255, 1996.

[2] Mark R. Brown and Robert Endre Tarjan. Design and analysis of a data structure for representing sorted lists. *SIAM Journal of Computing*, 9:594–614, 1980.

[3] Paul F. Dietz and Rajeev Raman. A constant update time finger search tree. *Information Processing Letters*, 52:147–154, 1994.

[4] Paul F. Dietz and Daniel D. Sleator. Two algorithms for maintaining order in a list. In *Proc. 19th Ann. ACM Symp. on Theory of Computing (STOC)*, pages 365–372, 1987.

[5] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert Endre Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38:86–124, 1989.

[6] Rudolf Fleischer. A simple balanced search tree with $O(1)$ worst-case update time. *International Journal of Foundations of Computer Science*, 7:137–149, 1996.

[7] Leo J. Guibas, Edward M. McCreight, Michael F. Plass, and Janet R. Roberts. A new representation for linear lists. In *Proc. 9th Ann. ACM Symp. on Theory of Computing (STOC)*, pages 49–60, 1977.

[8] Dov Harel. Fast updates of balanced search trees with a guaranteed time bound per update. Technical Report 154, University of California, Irvine, 1980.

[9] Dov Harel and George S. Lueker. A data structure with movable fingers and deletions. Technical Report 145, University of California, Irvine, 1979.

[10] Scott Huddleston and Kurt Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982.

[11] Haim Kaplan and Robert Endre Tarjan. Persistent lists with catenation via recursive slow-down. In *Proc. 27th Ann. ACM Symp. on Theory of Computing (STOC)*, pages 93–102, 1995.

[12] S. Rao Kosaraju. Localized search in sorted lists. In *Proc. 13th Ann. ACM Symp. on Theory of Computing (STOC)*, pages 62–69, 1981.

[13] Christos Levcopoulos and Mark H. Overmars. A balanced search tree with $O(1)$ worst-case update time. *Acta Informatica*, 26:269–277, 1988.

[14] Chris Okasaki. *Purely Functional Data Structures.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1996. Tech report CMU-CS-96-177.

[15] Mark H. Overmars. *The Design of Dynamic Data Structures*, volume 156 of *Lecture Notes in Computer Science.* Springer Verlag, Berlin, 1983.

[16] Rajeev Raman. *Eliminating Amortization: On Data Structures with Guaranteed Response Time.* PhD thesis, University of Rochester, New York, 1992. Computer Science Dept., U. Rochester, tech report TR-439.

[17] Athanasios K. Tsakalidis. AVL-trees for localized search. *Information and Computation*, 67:173–194, 1985.