Space-Efficient Finger Search on Degree-Balanced Search Trees

Guy E. Blelloch  
Carnegie Mellon University

Bruce M. Maggs  
Carnegie Mellon University

Shan Leung Maverick Woo  
Carnegie Mellon University

Follow this and additional works at: http://repository.cmu.edu/compsci

Published In  
Space-Efficient Finger Search on Degree-Balanced Search Trees*

Guy E. Blelloch Bruce M. Maggs Shan Leung Maverick Woo
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213
{guyb,bmm,maverick}@cs.cmu.edu

Abstract
We show how to support the finger search operation on degree-balanced search trees in a space-efficient manner that retains a worst-case time bound of $O(\log d)$, where $d$ is the difference in rank between successive search targets. While most existing tree-based designs allocate linear extra storage in the nodes (e.g., for side links and parent pointers), our design maintains a compact auxiliary data structure called the “hand” during the lifetime of the tree and imposes no other storage requirement within the tree.

The hand requires $O(\log n)$ space for an $n$-node tree and has a relatively simple structure. It can be updated synchronously during insertions and deletions with time proportional to the number of structural changes in the tree. The auxiliary nature of the hand also makes it possible to introduce finger searches into any existing implementation without modifying the underlying data representation (e.g., any implementation of Red-Black trees can be used). Together these factors make finger searches more appealing in practice.

Our design also yields a simple yet optimal inorder walk algorithm with worst-case $O(1)$ work per increment (again without any extra storage requirement in the nodes), and we believe our algorithm can be used in database applications when the overall performance is very sensitive to retrieval latency.

1 Introduction
The problem of maintaining a sorted list of unique, totally-ordered elements is ubiquitous in computer science. When efficient element access (insert, delete, or search) is needed, one of the most common solutions is to use some form of balanced search trees to represent the list. Over the years many forms of balanced search trees have been devised, analyzed, and implemented.

Balanced search trees are very versatile representa-

*This work was supported in part by the National Science Foundation under grants CCR-0205523 and CCR-9900304 and also through the Aladdin Center (www.aladdin.cs.cmu.edu) under grants CCR-0085982 and CCR-0122581. The second author is also affiliated with Akamai Technologies.
the “current” element. The finger search operation uses the finger as an extra hint to search for its new target and also shifts the finger to the element found. (Section 2 has a precise definition that allows the search target to be absent from the list.) In the worst scenario, finger searching matches the $O(\log n)$ time bound of a classical search; but in applications like merging where there is a locality of reference in the sequence of search targets, finger searching yields a significantly tighter time bound.

Finger search was introduced on a variant of B-trees by Guibas, McCreight, Plass, and Roberts [9] in 1977. Since then, finger search based on modification of balanced search trees has been studied by many researchers, e.g., Brown and Tarjan [7, 2-3 trees], Huddleston and Mehlhorn [11, (a,b) trees], Tsakalidis [22, AVL trees], Tarjan and Van Wyk [21, heterogeneous finger search trees] and Seidel and Aragon [19, Treaps]. In their original paper on splay trees, Sleator and Tarjan [20] have conjectured that the splay operation has the finger search property. Known as the Dynamic Finger Conjecture, it was subsequently proven by Cole [8]. There are other designs that are not entirely based on balanced search trees as well. For example, Kosaraju [14] designed a more general structure with the finger search property using on a collection of 2-3 trees. Skip Lists by Pugh [18] also support finger searching. More recently, Brodal [5] has investigated finger search trees designed to improve insertion and deletion time. Of special note are the purely-functional catenable sorted lists of Kaplan and Tarjan [12]. Their design not only has the finger search property, but it also requires very little space overhead. We will contrast our design with theirs later.

**Challenges and results.** Supporting finger search in balanced search trees can be challenging. The main difficulty is in shifting the finger fast enough to achieve a worst-case $O(\log d)$ time bound. Observe that if we have to strictly adhere to the unique path induced by the tree, then two elements with similar rank can be stored far apart. As an extreme example, consider the root element and its successor: the tree path has length $\Theta(\log n)$, but we only have $O(1)$ time.

One way to circumvent this apparent difficulty is to store extra information in the nodes so that we do not have to adhere to the tree path. For example, this approach has been taken by Brown and Tarjan [7] who added a parent pointer and two side links to each node. (Side links are pointers to the previous and next node at the same depth.) With these extra pointers, it can be shown that there exists a path of length $O(\log d)$ between two nodes differing in rank by $d$. Finger search can now be supported by taking this new path. The problem with this design is that a total of $3n$ extra pointers are introduced and the size of the tree is doubled, assuming the key has the same size as a pointer. In fact, among the many other tree-based designs mentioned above, this $O(n)$ extra storage requirement is a common trait.

Our design is an attempt to avoid this $O(n)$ storage requirement but at the same time retain the structural simplicity of balanced search trees. To this end, we base our design on degree-balanced search trees [17] and we assume a compact $k$-ary node with only $(k-1)$ keys and $k$ child pointers. Since any extra storage we need must be stored in some auxiliary data structures outside of the tree, our goal is to minimize the amount of auxiliary storage while supporting the finger search operation in worst-case $O(\log d)$ time.

As we will show in this paper, our design requires $O(\log n)$ space on a degree-balanced search tree with $n$ nodes and supports finger search in worst-case $O(\log d)$ time. The finger searches can go in both the forward and backward directions without any restriction. We also show that once the finger has been placed on the position of change, insertions and deletions can be implemented in time proportional to the number of structural changes in the tree. This allows us to transfer any results previously proven on these two operations, such as an amortized $O(1)$ time bound and the actual distribution of work at different depths of the tree [11]. In the development of our finger search algorithm, we also obtain a simple in-order walk algorithm with worst-case $O(1)$ work per increment. We believe that this improvement over the previous amortized $O(1)$ bound can be used in database applications when the overall performance is very sensitive to retrieval latency.

**Design overview.** We notice that if supporting finger searches is really possible under our restrictions, then we must be able to support a special case of it: an in-order walk with worst-case $O(1)$ work per increment. Our solution is to eagerly schedule the in-order walk and walk the path in advance. We call this the eager walk technique. Because we can only see a constant number of nodes at a time, we also need to keep track of our progress and so we have devised a simple data structure called the hand for this purpose. We will document these two ideas along with our in-order walk algorithm in Section 3.

Having solved the in-order walk problem, we then go back to finger search. Notice that in the in-order walk, the future search targets are known in advance. However, this is not true in finger search. Our understanding of eager walk suggests that we want to start shifting the finger before the actual search target is known. For finger search, that means we want to cache
some portion of the tree so that when the actual search target arrives, the cache will contain a prefix of the path from the finger to the target. If the length of the prefix is chosen to be long enough, then we will be able to finish shifting the finger over the rest of the path in $O(\log d)$ time. As it turns out, the hand is precisely such a cache despite being initially designed just for the in-order walk. At this point, we will also bring in a connection between the hand and the inverted spine technique used in heterogeneous finger search trees by Tarjan and Van Wyk [21]. Using this connection, our finger search algorithm becomes relatively straightforward. Section 4 will be devoted to this connection.

In our presentation in Sections 3 and 4, we have assumed that the finger only goes forward. In Section 5, we will handle the backward direction by using two hands and also show how the hands can be updated during insertions and deletions. Finally, we will conclude with a very brief discussion including how to update the hands during splits and joins in Section 6.

2 Notations and definitions

Lists and elements. In the rest of this paper, all lists are sorted and have unique elements drawn from $(\mathbb{Z}, \leq)$ and the variables $a$ through $e$ will range over $\mathbb{Z}$ without any further quantification. (It would be more general to leave the domain as any total order. For example, some total orders such as $(\mathbb{R}, \leq)$ do not have a natural notion of immediate successor. However, this issue does not come up in this paper.)

Finger destination. To handle the possible case that the search target is not in the list, we define $a^+$ to be the smallest element in the list that is larger than or equal to $a$ (much like the limit notation). When $a$ is larger than all elements in the list, let $a^+$ be a sentinel denoted by $\infty$. We can similarly define $a^-$. With these two definitions, a finger search for $a$ should place $f$ at $a^+$ if $a^f \leq a$, or $a^-$ otherwise, where $a^f$ is the element under $f$ when the finger search is started. Note that if $a$ is in the list, then $a^+$ and $a^-$ are both equal to $a$ and therefore the finger will be placed at $a$ in either case. This allows us to say the finger will be placed on $a^+$ (or $a^-$) when we are finger searching for $a$.

Trees and nodes. In a search tree $T_A$ representing a list $A$ of elements $a_1, \ldots, a_n$, the node containing $a_i$ will simply be called $x_i$ and the variables $u$ through $z$ will range over nodes. When referring to a node $x$, we use $x^-$ and $x^+$ to denote its predecessor and successor respectively. We denote the depth of a node $x$ simply as $\text{depth}(x)$, with the root at depth 1. The depth of the tree $\text{depth}(T_A)$ is the maximum depth among all nodes. We regard nodes without children as leaves.

As stated, our design is based on degree-balanced search trees. All the leaves in such a tree are at the same depth and its balance is maintained by varying the degree of internal nodes between fixed constants. 2-3 trees [2], B-trees [3] and $(a, b)$ trees [11] are all variants of degree-balanced search trees. Red-Black trees [10] can also be viewed as degree-balanced easily via the isomorphism with 2-3-4 trees. We sometimes simplify our presentation by assuming a complete binary search tree (BST), but we also show how to account for this to retain full generality.

A $k$-ary node $x$ has $(2k - 1)$ fields. The keys are sorted elements from $A$ and are denoted as $x^j$, for $j = 1, 2, \ldots, k - 1$ and the children are denoted as $x[j]$, for $j = 1, 2, \ldots, k$. We define the $j$-th left child to be $x[j]$ for $j = 1, 2, \ldots, k - 1$ and denote it by $x^j[L]$. The corresponding $j$-th right child is then $x[j + 1]$ and denoted by $x^j[R]$. If $x$ is a leaf, then the child pointers are all nil. For binary nodes, we simply drop the superscript. We say that the finger is under a node $x$ when the finger is pointing at a key inside the tree rooted at $x$.

A node is overfull if it has $k$ keys. In a degree-balanced search tree, there are no overfull nodes and different nodes can have different arity. Any overfull node will be split into two during an update.

Spines and relatives. We first define spines on binary trees and we give only the version for the right (forward) direction.

The right spine of a binary node is defined to be the list of node(s) starting at the node itself, followed by the right spine of its right child, if it exists. The right-left spine of a node is the node itself and left spine of its right child. (Our notation stresses the direction taken to traverse the spine and is consistent if we view the right spine as the right-right spine.) Now given any spine of a node, its atlas is the second node on the spine (a child) and its tail is the last node. Suppose we have
three nodes $x$, $y$, $z$ in a tree. If $x$ is on the left-right spine of $y$, then we say $y$ is the right parent of $x$. The right ancestors of $x$ will then be $y$ and the right ancestors of $y$. If $y$ is the right parent of $x$ and the left parent of $z$ with $x$ and $z$ at the same depth, then we say $z$ is the right peer of $x$. In the special case when $y$ is the parent of both, then we say $z$ is the right brother of $x$ instead. Figure 1 illustrates some of these concepts on a complete BST. Note that the dashed arrows are only for the purpose of illustration. In particular, our work does not make use of such pointers in the nodes. The right-left spine of $8$ has also been highlighted.

The definitions for nodes of any higher degree is straightforward using our $j$-th child quantification. If a $k$-ary node $y$ is the right parent of $x$ and $x$ is in $y^i[L]$, then we say $y^i$ is the right parent key of $x$.

**Deques.** We will use doubly-linked queues (deques) as a building block of the hand. A deque is made up of cells and we denote a deque with $k$ cells by $(c_0, \ldots, c_k)_k$, with the back on the right hand side. A deque supports the following operations in $O(1)$ time: MAKEDEQUE, PUSH, POP, INJECT, EJECT, FRONT, and PREPEND. Note that INJECT and EJECT operate on the back of a deque and a deque can be used as a catenable stack. With an additional pointer to a cell, a deque also supports SPLIT in $O(1)$ time. For more information on deques, refer to Knuth [13].

**3 In-order walk**

In this section, we motivate and present the design of the hand by developing an in-order walk algorithm with worst-case $O(1)$ work per increment. Our goal is to develop our understanding of the hand through this discussion. To simplify our presentation, we start by working with a complete BST and then generalize to handle all degree-balanced search trees.

**3.1 Design.** The simplest in-order walk algorithm is the straightforward recursive solution, which takes amortized $O(1)$ time per increment. To achieve the worst-case $O(1)$ bound, we need to schedule the discovery of nodes that will be processed later in order to avoid traversing a long path during an increment. We refer to this discovery activity as an eager walk. To avoid confusion, in this section we say that we “visit” a node when it is the actual node being processed by the in-order walk and we “explore” a node when it is being discovered due to the eager walk.

Now, let’s look at each increment individually. Suppose we are currently visiting the node $x$ and so the next node to visit is $x^{++}$. Observe that in a search tree, there are only two possible positions for $x^{++}$ to appear. If $x$ is not a leaf, then $x^{++}$ is the tail $y$ on the right-left spine of $x$. Otherwise, it is the right parent $z$ of $x$. (If $z$ does not exist also, then we must be at the rightmost node of the tree. We let the root have an imaginary parent labelled $\infty$ and end the in-order walk there.) Figure 2 illustrates our situation.

To handle the first case, we must traverse the full right-left spine of $x$ before we visit $x$. Since we have only a constant amount of time in each increment but the spine can be long, we can only afford to explore a constant number of nodes at a time and perform this multiple times. As we need the spinal nodes in the bottom-to-top order in the in-order walk, we associate a stack with $x$ and we push the right-left spinal nodes of $x$, beginning with the atlas, onto the stack as we discover them. The scheduling on a degree-balanced search tree is intuitive because all of the leaves are at the same depth and so the left-right spine of $x$ has the same length as the right-left spine. Since all the nodes on the left-right spine must be visited before $x$, a natural choice is to explore one right-left spinal node when we visit one left-right spinal node. This way, by the time we have visited the tail of the left-right spine, we will have explored the tail of the right-left spine, namely $y$.

The second case is simpler. To go up the tree, we use a stack to keep track of the ancestors as we descend between visits. But as we show in Figure 2, $x$ can have any number of left ancestors (up to the atlas $w$). To get to $z$ in constant time, we keep track of only the right ancestors, i.e., we push a node when we descend left and pop it out when we return to it and ascend right. Now $z$ will be at the top of the stack when we visit $x$. (We note that the idea of right parent stack has been used before, e.g., see Brown and Tarjan [6].)

The right parent stack is related to our eager walk as well. Notice that as we approach $y$ in the eager walk,
all the nodes we explored are right ancestors of $y$. Since the right ancestors of $x$ are also right ancestors of $y$, we are in fact building the upper part of the right parent stack for $y$. A catenable stack will be perfect for our purpose because when we catenate the right-left spine of $x$ onto its right parent stack, we will immediately have the right parent stack of $y$. However, we will need $\text{Inject}$ and $\text{Eject}$ in Section 5.4 to handle insertions and deletions. Hence, we will use a deque as a catenable stack.

3.2 The “hand” data structure. The hand is an auxiliary data structure designed to keep track of our progress in the eager walk. For our in-order walk algorithm, it is a deque named $\text{Rps}$. Stored inside the cells of $\text{Rps}$ are pointer pairs of the form $(\text{node, spine})$, where $\text{node}$ is a pointer to a node in the underlying tree and $\text{spine}$ is a (null) pointer that can be used to point to a deque containing similar pointer pairs so that we can prepend the deque pointed by $\text{spine}$ onto $\text{Rps}$.

Let the underlying tree be a complete BST $T$ and $\text{Rps}$ be a deque consisting of $k$ pointer pairs $((x_k, s_k), \ldots, (x_1, s_1))$. $\text{Rps}$ must obey these two invariants:

**Invariant 3.1.** (node) $x_1$ is on the right spine of $T$ and $\forall j \in \{2, \ldots, k\} : x_{j-1}$ is the right parent of $x_j$ in $T$.

**Invariant 3.2.** (spine) $\forall j \in \{1, \ldots, k\} : s_j$ is a deque of $(\text{node, spine})$ pairs representing a prefix of $x_j$’s right-left spine, with the atlas stored at the back. The length of $s_j$ is $\text{depth}(x_{j+1}) - \text{depth}(x_j) - 1$, where $\text{depth}(x_{k+1})$ is defined to be $\text{depth}(T) + 1$.

Figure 3 illustrates an example hand at node 5 in a complete BST with 15 nodes. Notice that we have added two dotted arrows pointing upward in the tree to reveal the nature of the right parent stack. As a demonstration of Invariant 3.2, note that the right peers of nodes 5 and 6 are precisely one node away from the end of the spine prefix associated with their right parents.

Using Invariant 3.2, we can immediately bound the size of the hand by the depth of the tree.

**Theorem 3.1.** (Hand Size) The hand for a complete BST $T$ has at most $\text{depth}(T)$ cells.

**Proof.** Suppose $\text{Rps}$ has $k$ cells. The total number of cells in the hand is $\sum_{j=1}^{k} (1 + |s_j|)$. By invariant 3.2, this is $k + (\text{depth}(x_{k+1}) - \text{depth}(x_1) - k)$ which is at most $\text{depth}(x_{k+1}) - 1 = \text{depth}(T)$.

3.3 Algorithm. To start the in-order walk, we first build the hand on the leftmost node of the tree by pushing the left spine of the tree into $\text{Rps}$. We associate an empty deque with each spinal node and use an empty $\text{Rps}$ to indicate termination. (The actual algorithm for increment is very succinct, but we have grouped together all the pseudo-codes in this paper into Appendix A. Please refer to the pseudo-code of $\text{INCREMENT}$ and $\text{EXTENDRIGHTLEFTSPINE}$.) The correctness of our algorithm follows from the discussion in Section 3.1 and it clearly takes $O(1)$ time per invocation. Note that a hand can be built on any node in $O(\log n)$ time. In our case it is built on the leftmost node.
3.4 Extending to k-ary nodes. The in-order walk algorithm above works on a complete BST. When generalizing it to degree-balanced search trees, our j-th child quantification is very handy. We will consider \( x^j \) as a binary node, with \( x^j[L] \) and \( x^j[R] \) as its two children. Suppose we are now visiting the rightmost node of the sub-tree rooted at \( x^j[L] \). By a quantified version of Invariant 3.2, at this point we will have all but the tail \( y \) of the right-left spine of \( x^j \). The increment to \( x^j \) will complete the spine and the increment to \( y \) will put us in the same situation as if we are visiting the leftmost node of the sub-tree rooted at \( x^{j+1}[L] \). The remaining details are straightforward. (See Appendix B for more information.)

**Theorem 3.2. (In-order Walk)** In-order walk on a degree-balanced search tree can be performed with worst-case \( O(1) \) time per increment, using \( O(\log n) \) space and \( O(\log n) \) pre-processing (to obtain the initial hand).

4 Finger search

In this section, we demonstrate how the hand allows us to perform finger searches in a degree-balanced search tree. Again we will simplify our presentation by working with a complete BST and limiting the finger searches to go in the forward direction.

We now consider a finger search for element \( a \) with a finger \( f \) at node \( w \). Let \( y \) be the right parent of \( w \) and \( z \) be the right peer of \( w \), as shown in Figure 4. Observe that the destination of \( f \) can be divided into three rank intervals: (i) \( (w, y] \), (ii) \( (y, z] \) and (iii) \( (z, \infty) \). The first two intervals are characterized by the right sub-tree of \( w \) together with \( y \) and the left sub-tree of \( z \) together with \( z \). We can distinguish among these cases in \( O(1) \) time by comparing \( a \) with \( y \) and \( z \), both readily available in the hand on \( w \).

To handle case (i), we first do an increment as in the in-order walk. This takes \( O(1) \) time. Then we traverse the right-left spine of \( w \) bottom-up by scanning the \( \text{Rps} \) towards the back until we hit an element larger than \( a \) (or the bottom of \( \text{Rps} \)). Let \( z \) be the node in the second to last cell we scanned (or the bottom of \( \text{Rps} \)). Observe that if \( a \) is in the tree, then it must either be in \( x \) or its right sub-tree, where we will perform an additional binary tree search. In either case, it is straightforward to restore the two invariants of the hand on our destination. The whole process takes time proportional to the length of \( x \)'s left spine minus one, which is logarithmic in the size of the left sub-tree skipped by the finger. (We note that the algorithm for this case is precisely the inverted spine technique used in heterogeneous finger search trees by Tarjan and Van Wyk [21].)

Case (ii) can be handled by first popping the \( \text{Rps} \) twice (removing \( w \) and \( y \)) and prepending the right-left spine prefix of \( y \) onto it (now \( z \) is at the top). We then start a binary tree search for \( a \) at \( z \) while restoring the invariants. The logarithmic time bound follows because the finger skipped the right sub-tree of \( w \).

We handle case (iii) by reducing it to case (i) on a larger scale. We first locate the lowest node \( x_j \) on \( \text{Rps} \) whose key is no larger than our target by successive popping. (Note that as we scan down the \( \text{Rps} \), the key gets larger.) Then we shift the hand over to \( x_j \) by completing its right-left spine. At this point we re-start the finger search at \( x_j \) and we know we will be in case (i). Note that both case (i) and case (ii) are just specializations of case (iii) and we can handle them using this more general procedure. To analyze the running time, we separate the rank difference into \( \delta(w, x_j) \) and \( \delta(x_j, a) \). The time it takes to obtain the hand on \( x_j \) is \( O(\log \delta(w, x_j)) \) because the size of the right sub-tree of \( x_{j+1} \) is at most \( \delta(w, x_j) \). The subsequent finger search takes time \( O(\log \delta(x_j, a)) \). The time bound follows from the inequality \( \log(c) + \log(d) < 2\log(c + d) \).

We note that our algorithm can be similarly generalized to handle k-ary nodes as we described in Section 3.4 and the time bound remains the same. We also provide a more precise specification of our algorithm in Appendix A in the form of pseudo-code.

**Theorem 4.1. (Forward Finger Search)** Using the hand, forward finger searches on a degree-balanced search tree can be performed in worst-case \( O(\log d) \) time, where \( d \) is the difference in rank between successive search targets.

5 Extensions

In this section we will outline how to extend the hand to support finger search in both directions and how to update the hand during insertions and deletions.
5.1 Left and right hands. We say that invariants 3.1 and 3.2 specify the *right hand*. By flipping the left/right directions, we obtain the *left hand* which consists of the left parent stack $Lps$. For simplicity, we will use “the hands” to denote the left hand and the right hand collectively.

Consider the hands on a node $x$. By definition, each of the ancestors of $x$ will appear on either $Lps$ or $Rps$. In particular, the root node will be at the bottom of one of them. We now extend the stack cells to contain a triple $(node, spine, cross)$, where *cross* is a pointer to another cell. Let $Lps$ be $\langle (x_1 k, s_1 k, c_1 k), \ldots, (x_{11}, s_{11}, c_{11}) \rangle$ and $Rps$ be $\langle (x_r k, s_r k, c_r k), \ldots, (x_r 1, s_r 1, c_r 1) \rangle$. Note that in general $l_k \neq r_k$ but $x = x_{l_k} = x_{r_k}$. We require the hands to satisfy one additional invariant:

**Invaraint 5.1. (cross)** Starting at the cell containing the root, the path specified by chasing the *cross* pointers is the path from the root to $x$, with the encoding that if $c_{l_k}$ or $c_{r_k}$ is nil, then it points to the cell directly above the current cell. If $x$ is a left child, then the path ends on $Lps$. Otherwise, it ends on $Rps$.

5.2 Decrement. Instead of showing how to perform decrement, we will describe how to update the left hand in an increment. Decrement will follow by symmetry. This also serves as a demonstration of Invariant 5.1 and the *cross* pointers. As an aid to our description below, Figure 5 shows the hands on nodes 2 to 5 in a complete BST with 15 nodes, which was shown in Figure 3.

Before we pop the $Rps$, we check to see if the $c_{l_p}$ points to the top cell of $Rps$. If so, we set it to nil. (See 3 $\rightarrow$ 4.) Then we pop the $Rps$ and extend the right-left spine of $x_{r_p}$ as usual. Let $(x_{l_j}, s_{l_j}, c_{l_j})$ be the cell $cell_{l_j}$ pointed to by $c_{r_p}$. (We can verify that this cell always exists.) If $s_{l_j}$ is non-empty, then we pop off its top cell to shorten the spine prefix by one node and set $s_{new}$ be nil. (See 2 $\rightarrow$ 3 and 4 $\rightarrow$ 5.) Otherwise, we set $c_{r_p}$ to nil and split $Lpr$ at $cell_{l_j}$ to obtain $\langle (x_{l_k}, s_{l_k}, c_{l_k}), \ldots, (x_{l_j}, s_{l_j}, c_{l_j}) \rangle$ as $s_{new}$. (See 3 $\rightarrow$ 4.) We preprend $s_{curr}$ to $Rps$ as usual. Finally we push $(x_{new}, s_{new}, nil)$ onto $Lps$, where $x_{new}$ is the top node in $Rps$. (We can verify that $s_{new}$ is the correct left-right spine prefix of $x_{new}$.)

While the above procedure may seem complicated, it can be derived from the maintainence required by the three invariants. We also note that the increment algorithm still takes $O(1)$ time. Since we showed the left hand can also be maintained in worst-case $O(1)$ time during an increment, by symmetry, the following theorem holds.

**Theorem 5.1. (Backward In-order Walk)** An in-order walk in the backward direction takes worst-case $O(1)$ time per decrement.

5.3 Backward finger search. The description in Section 4 can easily be adapted to update the left hand in a backward finger search. Here we show how to preserve Invariants 3.1 and 3.2 for the right hand as well. The maintenance of Invariant 5.1 is straightforward.

Recall that our finger search algorithm will first locate the left parent $x$ containing the smallest key that is no smaller than the target. Let the last cell we popped from $Lps$ be $(z, s_z, c_z)$. We will pop the $Rps$ and clean up the associated spine prefixes until the cell pointed to by $c_z$ is removed. Note that we have enough time to do this because we have skipped the left sub-tree of $z$.

At this point, the top cell in $Lps$ will be $(x, s_x, c_x)$. We split $Rps$ at $c_x$, push a new cell containing $x$ into $Rps$ and then associate the upper deque from the split to this cell as its right-left spine prefix. Finally, we extend the prefix to contain $z$ unless the finger search initially started at $z$. This step only takes $O(1)$ time.

Then our algorithm will complete the right-left spine of $x$ to obtain the hands on it. We update the right hand by completing its right-left spine prefix on $Rps$. Since the prefix already reaches $z$, the time it actually takes to complete the spine is logarithmic in the size of the left sub-tree of $z$, which we skipped.

If the target is not $x$, then it is in the left sub-tree. Our algorithm will preform a decrement and then start searching for the target by scanning the left-right spine of $x$ upward until we hit the smallest key that is no smaller than the target. Every time we go up a node, we also update the right hand by shortening the right-left spine prefix of $x$ in $Rps$. Finally, our algorithm will finish with a descent while restoring the invariants. The updates to the right hand in this part are straightforward and take the same amount of time as updating the left hand.

**Theorem 5.2. (Backward Finger Search)** The hands can be maintained in any sequence of finger
searches in $O(\log d)$ time per search, where $d$ is the difference in rank between successive search targets.

5.4 Insertion and Deletion. In a search structure that supports finger search, insertions and deletions are typically implemented by first placing the finger at the target element followed by the actual update. Huddleston and Mehlhorn [11] have shown that in a sequence of updates, the amount of structural changes in an $(a,b)$ tree is exponentially decreasing with the height of the propagation from the leaves and that each update takes amortized $O(1)$ time, both assuming an initially empty tree and discounting the time spent to shift the finger.

In this section, we will show that the hands can be updated to reflect each structural change in worst-case $O(1)$ time. Therefore, any result on the distribution of structural changes can be carried over to the hands directly. In particular, both of the above results by Huddleston and Mehlhorn hold.

In the following discussion, we assumed familiarity with the insertion and deletion algorithms for degree-balanced search trees. (See Huddleston and Mehlhorn [11] for more information.) Let the target element of the update be $t$. We will consider the hands for $k$-ary nodes. To simplify our presentation, we will only update the right hand w.r.t. the $k$-ary adaption of Invariants 3.1 and 3.2. The left hand can be updated by symmetry and it is also easy to maintain Invariant 5.1 throughout. We adopt the convention that the hands will be placed on $t$ after its insertion, or $t^{++}$ for deletion.

We will start with an observation. In a degree-balanced search tree, the structural change propagates up from a leaf to the root. All the nodes involved must be in either $\text{Lps}$ or $\text{Rps}$. Let $\text{Rps}$ be $\langle (x_k, s_k), \ldots, (x_1, s_1) \rangle_i$. We will update the hands by considering one depth at a time in a bottom-up fashion, provided that the hands are placed on a leaf first.

There are three kinds of possible structural changes at a depth: fusion, sharing and split. We will analyze them first and then return to insertion and deletion.

Fusion. Consider a node $x$, with the finger under it. Suppose $x$ has a right brother $y$ that will be fused into $x$ and $p$ is the right parent with key $c$. Note that $c$ is the key being demoted. Let $z$ be the right parent of $p$, if it exists. If $c$ is $p^k$, then first extend the spine prefix of $z$ and remove the cell of $p$ from $\text{Rps}$. No further change is needed if $x$ is in $\text{Rps}$. Otherwise, create a cell in $\text{Rps}$ above that of $p$ (or $z$ if $p$ is not in $\text{Rps}$ anymore) and let it contain $x$ with key $c$. Also eject the bottom cell from the spine prefix of $p$ and re-associate the result with $x$ instead.

Now suppose $x$ has a left brother $w$ and $x$ is being fused into $w$. No change is needed if $x$ is not in $\text{Rps}$. Otherwise, update the cell of $x$ to contain $w$ instead and adjust the offset in the cell accordingly.

Sharing. Consider a node $x$, with the finger under it. Suppose $x$ is sharing from its right brother $y$ and $p$ is the right parent with key $c$. We only need to update $\text{Rps}$ if $x$ is not originally in it. First create a new cell above that of $p$ and let it contain $x$ with key $c$. Then shorten the spine prefix of $p$ by ejecting the bottom cell and re-associate the result with $x$ instead.

Now suppose $x$ has a left brother $w$ where $x$ is sharing from. There are four possible cases depending on whether $x$ is in $\text{Rps}$ and similarly for $p$. In each of these cases, the structure of $\text{Rps}$ does not change except that the offset in the cell of $x$ needs to be updated to reflect the new key(s) in $x$.

Split. Consider a node $x$, with the finger under it and $c$ as its median key. Suppose after $c$ is promoted to the parent $p$, a new right brother $y$ of $x$ is created. Let the finger be under $x[j]$ and $z$ be the right parent of $p$, if it exists. We break down the analysis into three cases. In the first two, if $p$ is the new root, then inject an empty cell at the bottom of $\text{Rps}$ and let it contain $p$.

Suppose $x^j$ is smaller than $c$. If $p$ is on $\text{Rps}$, then no change is needed. Otherwise, shorten the spine prefix of $z$, create a new cell under that of $x$ and let it contain $p$ with key $c$.

Suppose $x^j$ is $c$ and let $d$ be $x^{j+1}$. If $p$ is not on $\text{Rps}$, then shorten the spine prefix of $z$, create a new cell under that of $x$ and let it contain $p$ with key $c$. Now remove the cell of $x$ from $\text{Rps}$ and create a new cell containing $y$ with key $d$. Finally, inject the new cell at the bottom of the spine prefix of $x$ and re-associate the result with $p$.

Suppose $x^j$ is larger than $c$. If $p$ is on $\text{Rps}$, then increment the offset in its cell. Also, if $x$ is on $\text{Rps}$, then update its cell to contain $y$ instead and adjust the offset accordingly.

Insertion. As we assumed the list maintains unique elements, $t$ must be absent and the hands are on either $t^-$ or $t^+$. Observe that at least one of $t^-$ and $t^+$ is in a leaf. Here we assume $t^+$ is in the leaf $x$ with the hands placed on it. If $t^+$ is an internal node instead, then perform a decrement to obtain the hands on $t^-$ and the rest is the same.

To begin the insertion, first increment the offset of the top cell of $\text{Rps}$. Notice that $\text{Rps}$ is a valid hand on $t$, but $x$ may have too many keys and a split or sharing will be needed. After we have handled $x$, its parent $p$ may have one more key and another split or sharing may occur at its depth. We will repeat until the propagation
stops. It should be clear that at each depth involved in the propagation, we spent only $O(1)$ time.

**Deletion.** Here we assume we have the hands on the leaf $x$ containing $t$. Observe that if $t$ is not in a leaf, then $t^{++}$ is. By Invariant 3.2, $t^{++}$ will be at the top of the spine stack associated with $x$. We replace $t$ with $t^{++}$ in $x$, perform an increment to obtain the hands on the tail $x'$, which contains the original $t^{++}$. Now we will consider deleting $t^{++}$ from $x'$ instead. A further decrement will put the hands back on $t^{++}$ in $O(1)$ time.

To begin the deletion, consider the leaf $x$. If $t$ is $x^k$, then first extend the spine prefix of the right parent of $x$ and remove the cell of $x$ from $Rps$. If $t$ is not $x^k$, then update the top cell of $Rps$ to contain $t^{++}$ instead of $t$. In both cases, notice that $Rps$ is still a valid right hand on $t$ (it is on $t^+$ now), but $x$ may have too few keys and a fusion or sharing will be needed. After we have handled $x$, its parent $p$ may have one less key and another fusion or sharing may occur at its depth. We will repeat until the propagation stops. At the end, if the root is empty and it is on the $Rps$, then we can simply eject the bottom cell. It should be clear that at each depth involved in the propagation, we spent only $O(1)$ time.

**Theorem 5.3. (Insertion and Deletion)** The hands can be updated synchronously during an insertion or a deletion in time proportional to the total number of structural changes in the tree.

**6 Discussion**

In this paper, we have shown how to support finger searches in a degree-balanced search tree with a worst-case $O(\log d)$ time bound using the hands as an auxiliary data structure. The hands are compact since they can be represented in $O(\log n)$ space for an $n$-node tree, and can be updated during insertions and deletions efficiently while preserving all existing time bounds proven on these operations. We note that the hands can also be updated similarly during the split and join operations. In fact, most of the structural changes involved are already analyzed in Section 5.4. The analysis of the dissection of the hands into multiple spine lists and the reassembling process is also straightforward. The details are documented in our technical report (CMU-CS-02-184), which also includes a discussion on how the hands can be used to improve performance in database applications by utilizing the pre-fetching capability available in many modern computer architectures. Finally, we note that the purely-functional catenable sorted lists of Kaplan and Tarjan [12] also support finger searches in worst-case $O(\log d)$ time and with a logarithmic space overhead. We provide a brief comparison between their design and ours in Appendix C.
A Finger search pseudo-code

We provide the pseudo-code of the forward finger search algorithm on a complete BST we presented in Sections 3.3 and 4 for reference.

\textbf{ExtendRightLeftSpine}(x, s)
1 if \(|s| = 0\) (* atlas vs. the rest *)
2 then \(y \gets x\).right
3 else \(y \gets s\).Front().node.left
4 if \(y \neq \text{nil}\)
5 then \(s\).Push((y, MakeDeque())

\textbf{CompleteRightLeftSpine}(x, s)
1 if \(|s| = 0\) (* atlas vs. the rest *)
2 then \(y \gets x\).right
3 else \(y \gets s\).Front().node.left
4 while \(y \neq \text{nil}\)
5 do \(s\).Push((y, MakeDeque())
6 \(y \gets y\).left

\textbf{Increment()}
1 \((x_{\text{curr}}, s_{\text{curr}}) \gets \text{Rps}.\text{Pop}()
2 \) if \(|\text{Rps}| > 0\)
3 then \((x_{\text{rp}}, s_{\text{rp}}) \gets \text{Rps}.\text{Front}()
4 \) ExtendRightLeftSpine\((x_{\text{rp}}, s_{\text{rp}})
5 \) \text{Rps}.\text{PREPEND}(s_{\text{curr}})

\textbf{RootedSearch}(b)
1 \((x_{\text{curr}}, s_{\text{curr}}) \gets \text{Rps}.\text{Pop}()
2 \) if \(|\text{Rps}| > 0\)
3 then \((x_{\text{rp}}, s_{\text{rp}}) \gets \text{Rps}.\text{Front}()
4 else \((x_{\text{rp}}, s_{\text{rp}}) \gets (x_{\text{curr}}, \text{MAKEDEQUE}())
5 \) while \(x_{\text{curr}} \neq \text{nil}\)
6 do (* restore invariants while descending *)
7 if \(b \leq x_{\text{curr}}\).Key
8 then \((x_{\text{rp}}, s_{\text{rp}}) \gets (x_{\text{curr}}, \text{MAKEDEQUE}())
9 \text{Rps}.\text{Push}(x_{\text{curr}}, s_{\text{curr}}))
10 \(x_{\text{curr}} \gets x_{\text{curr}}\).left
11 \) else \text{ExtendRightLeftSpine}\((x_{\text{rp}}, s_{\text{rp}})
12 \(x_{\text{curr}} \gets x_{\text{curr}}\).right

\textbf{ForwardSubTreeSearch}(b)
1 \((x_{\text{curr}}, s_{\text{curr}}) \gets \text{Rps}.\text{Pop}()
2 \) (* ascend along the inverted spine *)
3 while \(|\text{Rps}| > 0 \land \text{Rps}.\text{Front}().\text{node}.\text{key} \leq b
4 \) do \((x_{\text{curr}}, s_{\text{curr}}) \gets \text{Rps}.\text{Pop}()
5 \text{Rps}.\text{Push}(x_{\text{curr}}, s_{\text{curr}}))
6 (* descend as in a binary tree search *)
7 \text{RootedSearch}(b)

\textbf{ObtainInitFinger}(T)
1 \text{Rps} \gets \text{MAKEDEQUE}()
2 \text{Rps}.\text{Push}(T.\text{root}, \text{MAKEDEQUE}())
3 \text{RootedSearch}(\text{- infinity})

\textbf{ForwardFingerSearch}(b)
1 (* assumes hand is not at infinity and \(x_{\text{curr}}.\text{key} < b \) *)
2 \(x_{\text{curr}} \gets \text{Rps}.\text{Front}().\text{node}
3 \) if \(|\text{Rps}| \geq 2\)
4 then \((x_{\text{rp}}, s_{\text{rp}}) \gets \text{Rps}.\text{Front}().\text{next} (* 2^nd cell *)
5 if \(b \leq x_{\text{rp}}.\text{key} (* \text{case (i) } *)
6 then \text{INCREMENT()}
7 \text{ForwardSubTreeSearch}(b)
8 return \((x_{\text{rp}}, s_{\text{rp}}) \gets \text{Rps}.\text{Pop}() (* \text{case (ii) and case (iii) } *)
9 \) while \(|\text{Rps}| > 0 \land \text{Rps}.\text{Front}().\text{node}.\text{key} \leq b
10 \) do \((x_{\text{rp}}, s_{\text{rp}}) \gets \text{Rps}.\text{Pop}()
11 \text{Rps}.\text{Push}(x_{\text{rp}}, s_{\text{rp}})

B Handling \(k\)-ary nodes

We only require a slight adjustment to the cells when we extend the hands to handle \(k\)-ary nodes. In particular, instead of storing a pointer to a \(k\)-ary node \(x\), we now also store the offset, which indicates the sub-tree that contains the finger. For example, if the finger is under \(x[j]\), then \(x\) will appear on the \(\text{Rps}\) as \((x, j)\) instead of just \(x\). This is to reflect the fact that \(x[j]\) is the right parent key. For concision, we will simply say \(x[j]\) in our discussion and a cell will be written as \((x[j], s)\). Here we present the increment algorithm that has been adapted to handle \(k\)-ary nodes as an example of how we can adapt our algorithms.

\textbf{INCREMENT()}
1 \((x_{\text{curr}}, s_{\text{curr}}) \gets \text{Rps}.\text{Pop}()
2 \) if \(j < k - 1\)
3 then \text{Rps}.\text{Push}(x_{\text{curr}}, s, \text{NIL})
4 else if \(|\text{Rps}| > 0\)
5 then \((x_{\text{rp}}, s_{\text{rp}}) \gets \text{Rps}.\text{Front}()
6 \) ExtendRightLeftSpine\((x_{\text{rp}}, s_{\text{rp}})
7 \text{Rps}.\text{PREPEND}(s_{\text{curr}})

C Hands and inverted spines

In this paper, we have demonstrated that our view of finger search as a property, rather than an operation, allows us a much bigger design space. In fact, there are other previous works that do not use an element pointer. A recent exception to this is the purely-functional catenable sorted list designed by Kaplan and Tarjan [12] in 1996. Instead of an element pointer, their structure allows splitting the list at the \(d\)-th position in worst-case \(O(\log d)\) time and catenating in time doubly logarithmic in the size of the smaller list. Finger search can thus be realized by splitting and catenating between two instances of their structure, with the finger pointing at the element at the break.

Although it was not mentioned explicitly, the modified 2-3 finger search tree representation in their paper actually uses only \(O(\log n)\) extra storage for an \(n\)-element list. The key to their design is carefully relax the degree constraint on the spines to allow a suitable storage redundancy, which can in turn be used to absorb the propagation of structural changes due to splits and catenations. We can view their design as an improvement upon the heterogeneous finger search trees [21] in which splits and joins have an amortized time bound. (See Booth [4] (Ch. 2), Mehlhorn [16] and Kosaraju [15] for the analysis.) As we have pointed out in Section 4, the power of the hands also comes from the inverted spine technique used in the heterogeneous finger search trees. However, instead of relaxing the degree constraint on the spines, we showed that it is possible to avoid the splits and joins if we view the “inverted spine” by the way of the hands.