Space-Efficient Finger Search on Degree-Balanced Search Trees

Guy E. Blelloch Bruce M. Maggs
Shan Leung Maverick Woo

September 2002
CMU-CS-02-184

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

An extended abstract of this paper has appeared in ACM-SIAM Symposium on Discrete Algorithms (SODA) 2003. This paper was last revised on April 15, 2003.

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 in-order 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.

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.

Keywords: Data Structure, Balanced Search Trees, Finger Search

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 representations of sorted lists. In addition to providing element access in logarithmic time, certain forms also allow efficient aggregated operations like set intersection and union. For example, Brown and Tarjan [6] have shown a merging algorithm using AVL trees [1] with an optimal $O(m \log \frac{n}{m})$ time bound, where m and n are the sizes of the two lists with $m \leq n$.

Their merging algorithm is, however, "not obvious and the time bound requires an involved proof" [7, p. 613]. As such, in their subsequent paper, Brown and Tarjan [7] proposed a new structure by introducing extra pointers into a 2-3 tree [2] and called it a *level-linked 2-3 tree*. The merging algorithm on level-linked 2-3 trees is simple and intuitive and it uses the idea of finger searches, which we will define shortly. But there is a trade-off in this design. Each node in a level-linked 2-3 tree contains not only a key and two child pointers, but also a parent pointer and two side links. Considering this relatively high space requirement and the elegance of their simple yet optimal merging algorithm, it is natural to wonder if finger searches can be supported in a more *space-efficient* manner on any existing balanced search trees such as 2-3 trees. This is the motivation of our work.

Finger search. Consider a sorted list A of n elements $a_1, \ldots, a_n$ represented by a search structure. Let the *rank* of an element be its position in the list and let $\delta_A(a_i, a_j)$ be $|\, i - j \,|$, i.e., the difference in the ranks of a_i and a_j w.r.t. the elements of A. We say that the search structure has the *finger search property* if searching for a_j takes $O(\log \delta_A(a_i, a_j))$ time, where a_i is the most recently found element. The time bound can be worst-case, expected-case or amortized and we will distinguish them explicitly when needed. (As usual, we let $\log x$ denote $\log_2 \max(2, x)$ and we will simply say $O(\log d)$ when the elements a_i and a_j are not made explicit.)

A *finger* is a reference to an element in the list and historically it is often realized by a simple pointer to an element. (Indeed some papers mandate this representation in their definitions, e.g., see [5].) Typically, we maintain the invariant that the finger is on the most recently found element and we refer to this element as 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 is appropriate when the search target is 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 [3] by Guibas et al. [10] 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 [12, (a, b) trees], Tsakalidis [23, AVL trees], Tarjan and Van Wyk [22, heterogeneous finger search trees] and Seidel and Aragon [20, treaps]. In their original paper on splay trees, Sleator and Tarjan [21] 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 [15] designed a more general structure with the finger search property using on a collection of 2-3 trees. Skip Lists by Pugh [19] 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 [13]. Their design not only has the finger search property, but it also requires very little space overhead. We will contrast our design with theirs in Section 6.

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 [18][1] 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 searches in worst-case $O(\log d)$ time. The finger searches can go in both 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 [12]. 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. (The focus of this paper is not on database applications, but we have documented our idea in Appendix C.)

Design overview. We notice that if supporting finger searches is really possible under our storage 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 searches. Notice that in the in-order walk, the future search targets are known in advance. However, this is not true in finger searches. Our understanding of eager walk suggests that we want to start shifting the finger *before* the actual search target arrives. For finger searches, that means we want to cache some portion of

[1]Apparently the term *degree-balanced* search trees was coined in this monograph. However, it does not cover the details of such search trees. See other references in Section 2 for more information.

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 [22]. Using this connection, our finger search algorithm becomes relatively straightforward. Section 4 will be devoted to presenting this connection.

In our presentation in Sections 3 and 4, we will assume for simiplicity 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. In addition, we also analyze how the hands can be used during splits and joins. Finally, we will contrast our design with Kaplan and Tarjan's [13] in Section 6 and then conclude with some remarks to deal with practical issues that may arise when using the hands.

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 ∞. We can symmetrically define a^-. With these two definitions, a finger search for a should place f at a^+ if $a^f \leq a$ (forward), or a^- otherwise (backward), 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 n elements $a_1, \ldots, a_n$, the node containing a_i will simply be called x_i and the variables w through z will range over nodes. (Notice that a node can contain multiple keys, in which case multiple x_i's can correspond to the same node. However, we will only use the x_i notation when referring to nodes by their ranks.) When referring to a node x_i, we use x_i^{--} and x_i^{++} to denote its *predecessor* x_{i-1} and *successor* x_{i+1} respectively. We denote the *depth* of a node x simply as $depth(x)$, with the depth of the root defined to be 1. The depth of the tree $depth(T_A)$ 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 [12] are all variants of degree-balanced search trees. Red-Black trees [11] 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 assumption 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 sub-tree rooted at x.

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

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.

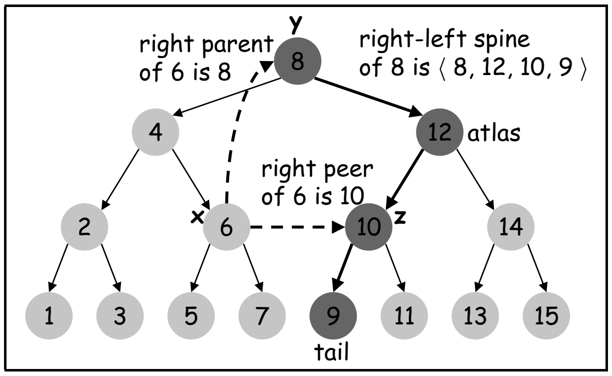


Figure 1: Parent, Peer, and Spine

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 $y^j[L]$, then we say y^j is the *right parent key* of x.

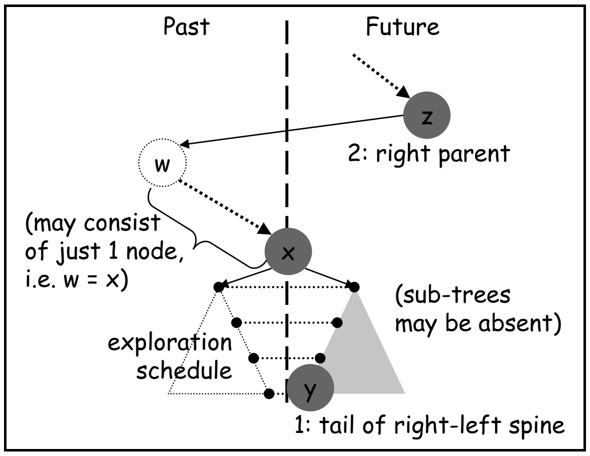
Deques. We will use doubly-linked queues (*deques*) as a building block of the hand. A deque is made up of *cells* and we denote a deque with k cells by $\langle c_k, \dots, c_1 \rangle_{\dashv}$, with the back on the right hand side. A deque supports the following operations in $O(1)$ time: MakeDeque, Push, Pop, Inject, Eject, Front, Back, 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 [14].

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 simpliest 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.


Past
Future
z
2: right parent
w
(may consist of just 1 node, i.e. w = x)
x
(sub-trees may be absent)
exploration schedule
y
1: tail of right-left spine
Figure 2: Possible locations of x^{++} and spine exploration schedule

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 ∞ 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 descend 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 INJECT and 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 Rps. Stored inside the cells of Rps are pointer pairs of the form (node, spine), where node is a pointer to a node in the underlying tree and 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 spine onto Rps.

Let the underlying tree be a complete BST T and Rps be a deque consisting of k pointer pairs $\langle (x_k, s_k), \ldots, (x_1, s_1) \rangle_{\dashv}$. 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* (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 $depth(x_{j+1}) - depth(x_j) - 1$, where $depth(x_{k+1})$ is defined to be $depth(T) + 1$.*

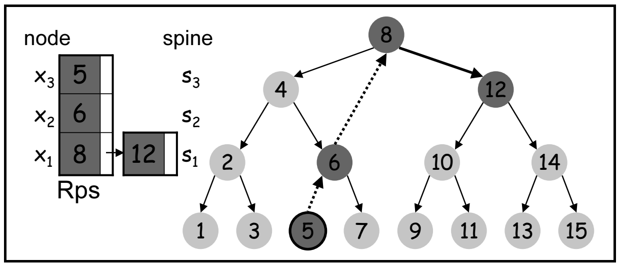


Figure 3: An example hand on 5

We now relate these two invariants with our design. First of all, the top node x_k in Rps will always correspond to the node x that we are currently visiting. Together with Invariant 3.1, Rps is indeed the right parent stack of x. Now consider the node x_{j-1}. By Invariant 3.1, it is the right parent of x_j. By Invariant 3.2, the length of its associated spine prefix s_{j-1} is $depth(x_j) - depth(x_{j-1}) - 1$. If z is the last node on the prefix, then z is at depth $depth(x_j) - 1$ and therefore $z[L]$ is the right peer of x_j. Since z is stored at the top of s_{j-1}, we will be able to reach the right peer of x_j in $O(1)$ time once we reach the cell containing s_{j-1}. A special case to notice is s_k. By Invariant 3.2 and our definition of $depth(x_{k+1})$, its length is $depth(T) - depth(x_k)$. This is precisely the length of its full right-left spine and this also reflects our design. (In our usage, "prefix" is not necessarily strict and so the full spine is a prefix of itself.)

The two invariants not only allow us to execute our desired schedule while we are visiting the nodes on the left-right spine of x_{k-1}, but also give us a very strong hint as to why the hand will facilitate finger search. By traversing down Rps, we can reach the right ancestors of the current node *as if* we had right parent pointers. Moreover, the right peer of any node on Rps can be reached with an additional $O(1)$ time, *as if* we had forward side links on each of the right ancestors. The power of these pointers has already been demonstrated by Brown and Tarjan [7] in level-linked 2-3 trees: these pointers are exactly the pointers introduced to facilitate finger searches.

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 reflect 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 $depth(T)$ cells.*

Proof Suppose 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 $k + (depth(x_{k+1}) - depth(x_1) - k)$ which is at most $depth(x_{k+1}) - 1 = depth(T)$. ■

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 Rps. We associate an empty deque with each spinal node and use an empty 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 INCREMENT and 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. (Refer to BUILDHAND for how this can be done.) 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 a demonstration.)

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.

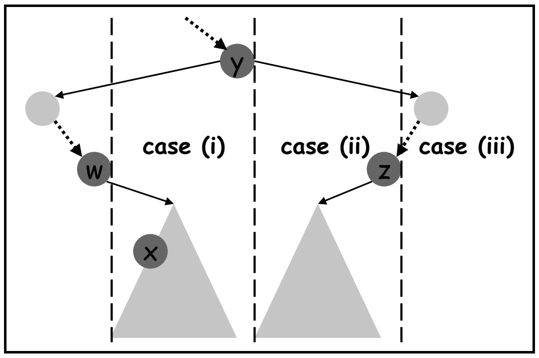


Figure 4: Possible destination of finger

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, ∞). 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 from our 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 Rps towards the back until we hit an element larger than a (or the bottom of Rps). Let x be the node in the *second to last* cell we scanned (or the bottom of 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 [22].)

Case (ii) can be handled by first popping the 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 Rps whose key is no larger than our target by successive popping. (Note that as we scan down the 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 seperate 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 $O(\log \delta(x_j, a))$, as we have already proved when analyzing case (i). 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, deletions, splits and joins.

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 $(\mathsf{node}, \mathsf{spine}, \mathsf{cross})$, where cross is a pointer to another cell. Let Lps be $\langle(x_{lk}, s_{lk}, c_{lk}), \ldots, (x_{l1}, s_{l1}, c_{l1})\rangle_{\dashv}$ and Rps be $\langle(x_{rk}, s_{rk}, c_{rk}), \ldots, (x_{r1}, s_{r1}, c_{r1})\rangle_{\dashv}$. Note that in general $lk \neq rk$ but $x = x_{lk} = x_{rk}$. We require the hands to satisfy one additional invariant:

Invariant 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_{lk} or c_{rk} 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.

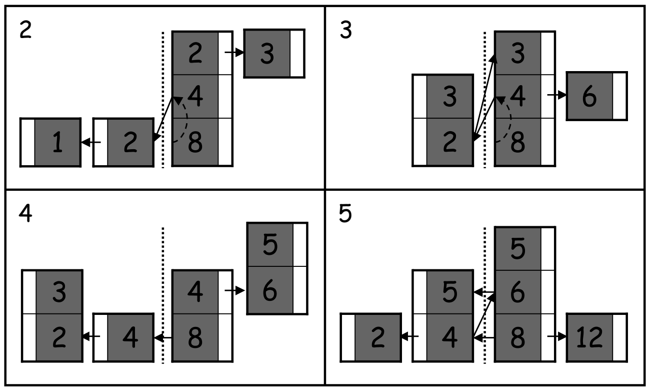


Figure 5: Example hands on 2 to 5, shown with cross pointers. (Dashed pointers are implicit.)

Before we pop the Rps, we check to see if the c_{lp} points to the top cell of Rps. If so, we set it to nil. (See 3 → 4.) Then we pop the Rps and extend the right-left spine of x_{rp} as usual. Let (x_{lj}, s_{lj}, c_{lj}) be the cell $cell_j$ pointed to by c_{rp}. (We can verify that this cell always exists.) If s_{lj} 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_{rp} to nil and split Lpr at $cell_j$ to obtain $\langle (x_{lk}, s_{lk}, c_{lk}), \ldots, (x_{lj}, s_{lj}, c_{lj}) \rangle_{\dashv}$ as s_{new}. (See $3 \rightarrow 4$.) We preprend s_{curr} to Rps as usual. Finally we push $(x_{new}, s_{new}, \mathsf{nil})$ 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 left-right 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 (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 Insertions and Deletions

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 [12] 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 will continue to hold even when we have to maintain the hands.

In the following discussion, we assumed familiarity with the insertion and deletion algorithms for degree-balanced search trees. (See Huddleston and Mehlhorn [12] 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 due to insertions and deletions propagates up from a leaf to the root. All the nodes involved must be in either Lps or Rps. Let Rps be $\langle (x_k, s_k), \ldots, (x_1, s_1) \rangle_{\dashv}$. 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: node fusion, children sharing and node split. (The last one is not to be confused with the splitting of a tree.) We will first analyze them and then return to insertions and deletions.

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 Rps. No further change is needed if x is in Rps. Otherwise, create a cell in Rps above that of p (or z if p is not in 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 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 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 Rps and similarly for p. In each of these cases, the structure of 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 overfull 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 Rps and let it contain p.

Suppose x^j is smaller than c. If p is on 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 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 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 Rps, then increment the offset in its cell. Also, if x is on 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 Rps. Notice that 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 need. 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.*

5.5 Splits and Joins

Again we assume familiarity with the split and join algorithms on degree-balanced search trees. (See Cormen et al.[9, p. 278, p. 399] for their coverage on Red-Black Trees and 2-3-4 Trees.) Our focus will be on analyzing the maintainence of the hands during these operations.

Join. Consider joining two trees T_1 and T_2 with b as the splitting key where $a < b < c$ for any $a \in T_1$ and $c \in T_2$. Let h_1 and h_2 be the heights of T_1 and T_2 respectively and by symmetry assume $h_1 \leq h_2$. The join operation should produce a new tree $T = T_1 \cup \{b\} \cup T_2$ in $O(h_2 - h_1)$ time, assuming the two trees maintain their own heights. For the purpose of our analysis, we will assume that we have our hands holding any key in T_2.

The first step in a typical join algorithm is to identify the node z on the left spine of T_2 such that the sub-tree rooted at z has the same height as T_1. When the algorithm is scanning for z down from the root, we can easily locate the cells in the hands that correspond to that height. (There can be one cell at this depth in both Lps or Rps and so there are exactly either one or two.) Notice that the cells located may not correspond to z, but they are the starting points for us to update the hands one depth at a time.

Once z has been identified on the spine, the root node of T_1 will be fused into z, with b as the key in-between. z may now get overfull and a sequence of node splits and children sharings may follow. The key observation is that this sequence of structural changes all occur on the left spine of T_2, starting at the depth of z. The situaton is very similar to an insertion, except the structural change propagates from z instead of from a leaf. Therefore, the hands can be updated in a way similar to during insertions and stop as soon as the propagation stops. The actual details are straightforward.

Theorem 5.4 (Join) *The hands in the larger tree can be updated synchronously during a join in time proportional to the total number of structural changes in that tree.*

However, we note that if there is a pair of hands on the smaller tree (T_1 in our case), then discarding the hands alone will take $O(h_1)$ time. Fortunately, there is no pointers from the nodes to the hands. Therefore, we may choose to discard the hands at a later time.

Split. When we split a tree T of height h using a key b, we obtain two trees T_1 and T_2 such that $a < b < c$ for any $a \in T_1$ and $c \in T_2$ in $O(\log n)$ time. In this analysis, we assume we have already placed the hands on b (see below if we do not have our hands on b yet) and by symmetry we assume b is in the left sub-tree of root. At the end of the split, we should have two pairs of hands, one at b^{--} and the other at b^{++}.

A typical split algorithm will decompose T into four groups: the left ancestors of b, their left sub-trees and symmetrically for the right ancestors. By symmetry, we will consider only the right ancestors and their right sub-trees. Let Rps be $\langle(x_{rk}, s_{rk}), \ldots, (x_{r1}, s_{r1})\rangle_{\dashv}$, i.e., x_{rk} contains b and x_{r1} is the root, and let T_{ri} be the right sub-tree of x_{ri}. To obtain T_2, split will be joining T_{ri} to the left of $T_{r(i-1)}$ using $x_{r(i-1)}$ as the splitting key for $i = k$ down to 2 (or from $k-1$ down to 2 if b is a leaf). We now show that the hands actually facilitate these joins in an intuitive way.

Consider joining T_{ri} with $T_{r(i-1)}$. Observe that, using the notation in our analysis of join, we don't have to scan for z any more. By Invariant 3.2, z will be the left-left grandchild of the node contained in the top cell of $s_{r(i-1)}$. The key $x_{r(i-1)}$ is also readily available on the Rps. Therefore, we can immediately start the join at z. Furthermore, notice that s_{ri} actually contains a fragment of the left spine of T_2. Therefore, the assembling s_{ri}'s into the Rps for T_2 is can be done straightforwardly. The Lps for T_2 will be a single cell containing b^{++}. The hands for T_1 can also be obtained symmetrically.

Theorem 5.5 (Split) *When splitting a tree T into T_1 and T_2 using the current key b held in the hands, the two new hands at b^{--} and b^{++} can be generated in time proportional to the total amount of work done by the joins to assemble the right spine of T_1 and the left spine of T_2.*

The above analysis assumes that we have already placed the hands on the splitting key b. Notice that the total time of the split is proportional to the depth of b in T. Therefore, if b is deep and the hands are not positioned close-by, then it will be simpler to discard the hands and split T as usual. Rebuilding the two hands can then be done in logarithmic time.

6 Discussion

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 [13] 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 searches 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 to 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 [22] in which splits and joins have an amortized time bound. (See Booth [4, Ch. 2], Mehlhorn [17] and Kosaraju [16] 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. This connection should be relatively straightforward if we review our discussion of split in Section 5.5.

Remarks on practice. We would like to conclude with several remarks on issues that may arise when the hands are used in practice. First, notice that the simultaneous maintainance of the forward and backward hands is often not necessary. In applications like merging, only the forward hand is needed and therefore we do not need to maintain Lps and the cross points on Rps. Furthermore, even when an application requires the ability to perform finger searches in both directions, we note that a new hand can be built on any node in only logarithmic time. Hence if the number of direction change is small and we must maintain exactly one hand, it may be more desirable to simply rebuild the hand at each direction change. Finally, we have included a brief discussion in Appendix C on how the hands can be used to facilitate pre-fectching in databases. While our in-order walk result is only a theoretical improvement, it will be interesting to see if the space savings in the tree also translates to a cache performance gain that outweights the overhead to maintain the hands.

References

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

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison Wesley, 1974. 1, 2

[3] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972. 1, 2

[4] H. D. Booth. *Some Fast Algorithms on Graphs and Trees.* PhD thesis, Princeton University, 1990. 6

[5] G. S. Brodal. Finger search trees with constant insertion time. In *Proc. 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 540–549, 1998. 1

[6] M. R. Brown and R. E. Tarjan. A fast merging algorithm. *Journal of the ACM*, 26(2):211–226, 1979. 1, 3.1

[7] M. R. Brown and R. E. Tarjan. Design and analysis of a data structure for representing sorted lists. *SIAM Journal of Computing*, 9(3):594–614, 1980. 1, 1, 1, 3.2

[8] R. Cole. On the dynamic finger conjecture for splay trees part II: The proof. Technical Report TR1995-701, Courant Institute, New York University, 1995. 1

[9] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, 1989. 5.5

[10] L. J. Guibas, E. M. McCreight, M. F. Plass, and J. R. Roberts. A new representation for linear lists. In *Proc. 9th Annual ACM Symposium on Theory of Computing*, pages 49–60, 1977. 1

[11] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proc. 19th IEEE Symposium on Foundations of Computer Science*, pages 8–21, 1978. 2

[12] S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982. 1, 1, 2, 5.4

[13] H. Kaplan and R. E. Tarjan. Purely functional representations of catenable sorted lists. In *Proc. 28th Annual ACM Symposium on the Theory of Computing*, pages 202–211, 1996. 1, 1, 6

[14] D. E. Knuth. *The Art of Computer Programming, Volume 1.* Prentice Hall PTR, 3 edition, 1997. 2

[15] S. R. Kosaraju. Localized search in sorted lists. In *Proc. 13th Annual ACM Symposium on Theory of Computing*, pages 62–69, 1981. 1

[16] S. R. Kosaraju. An optimal RAM implementation of catenable min doubled-ended queues. In *Proc. 5th Annual ACM Symposium on Discrete Algorithms*, pages 195–203, 1994. 6

[17] K. Mehlhorn. *Data Structures and Efficient Algorithms, Volume 1: Sorting and Searching*, pages 214–216. Springer-Verlag, 1984. 6

[18] M. H. Overmars. *The Design of Dynamic Data Structures*, pages 34–35. Number 156 in LNCS. Springer-Verlag, 1983. 1

[19] W. Pugh. A skip list cookbook. Technical Report CS-TR-2286.1, University of Maryland, College Park, 1990. 1

[20] R. Seidel and C. R. Aragon. Randomized search trees. *Algorithmica*, 16:464–497, 1996. 1

[21] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32:652–686, 1985. 1

[22] R. E. Tarjan and C. J. Van Wyk. An $O(n \log \log n)$-time algorithm for triangulating a simple polygon. *SIAM Journal of Computing*, 17(1):143–178, 1988. 1, 1, 4, 6

[23] A. K. Tsakalidis. AVL-trees for localized search. *Information and Control*, 67:173–194, 1985. 1

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.

```
ExtendRightLeftSpine($x, s$)
1 if $|s| = 0$   (∗ atlas vs. the rest ∗)
2    then $y \leftarrow x$.right
3    else $y \leftarrow s$.Front().node.left
4 if $y \neq$ nil
5    then $s$.Push(($y$, MakeDeque()))
```

```
CompleteRightLeftSpine($x, s$)
1 if $|s| = 0$   (∗ atlas vs. the rest ∗)
2    then $y \leftarrow x$.right
3    else $y \leftarrow s$.Front().node.left
4 while $y \neq$ nil
5     do $s$.Push(($y$, MakeDeque()))
6        $y \leftarrow y$.left
```

```
Increment()
1 $(x_{curr}, s_{curr}) \leftarrow$ Rps.Pop()
2 if |Rps| $> 0$
3    then $(x_{rp}, s_{rp}) \leftarrow$ Rps.Front()
4         ExtendRightLeftSpine($x_{rp}, s_{rp}$)
5 Rps.Prepend($s_{curr}$)
```

```
RootedSearch($b$)
1 $(x_{curr}, s_{curr}) \leftarrow$ Rps.Pop()
2 if |Rps| $> 0$
3    then $(x_{rp}, s_{rp}) \leftarrow$ Rps.Front()
4    else $(x_{rp}, s_{rp}) \leftarrow (x_{curr}$, MakeDeque())
```

```
 5  while $x_{curr} \neq$ nil
 6      do (∗ restore invariants while descending ∗)
 7          if $b \leq x_{curr}$.Key
 8              then $(x_{rp}, s_{rp}) \leftarrow (x_{curr}$, MakeDeque())
 9                   Rps.Push($(x_{rp}, s_{rp})$)
10                   $x_{curr} \leftarrow x_{curr}$.left
11              else ExtendRightLeftSpine($x_{rp}, s_{rp}$)
12                   $x_{curr} \leftarrow x_{curr}$.right
```

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

```
BuildHand($T, b$)
1  Rps $\leftarrow$ MakeDeque()
2  Rps.Push(($T$.root, MakeDeque()))
3  RootedSearch($b$)
```

```
ObtainInitFinger($T$)
1  BuildHand($T, -\infty$)
```

```
ForwardFingerSearch($b$)
 1  (∗ assumes hand is not at $\infty$ and $x_{curr}$.key $< b$ ∗)
 2  $x_{curr} \leftarrow$ Rps.Front().node
 3  if |Rps| $\geq 2$
 4     then $(x_{rp}, s_{rp}) \leftarrow$ Rps.Front().next   (∗ $2^{nd}$ cell ∗)
 5  if $b \leq x_{rp}$.key   (∗ case (i) ∗)
 6     then Increment()
 7          ForwardSubTreeSearch($b$)
 8          return
 9  $(x_{rp}, s_{rp}) \leftarrow$ Rps.Pop()   (∗ case (ii) and case (iii) ∗)
10  while |Rps| $> 0 \land$ Rps.Front().node.key $\leq b$
11      do $(x_{rp}, s_{rp}) \leftarrow$ Rps.Pop()
12  Rps.Push($(x_{rp}, s_{rp})$)
13  CompleteRightLeftSpine($x_{rp}, s_{rp}$)
14  Increment()
15  ForwardSubTreeSearch($b$)
```

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 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.

```
Increment()
1  $(x^j_{curr}, s_{curr}) \leftarrow$ Rps.Pop()
2  if $j < k - 1$
3     then Rps.Push($x^{j+1}_{curr}$, nil)
4     else if |Rps| > 0
5             then $(x^j_{rp}, s_{rp}) \leftarrow$ Rps.Front()
6                  ExtendRightLeftSpine($x^j_{rp}, s_{rp}$)
7  Rps.Prepend($s_{curr}$)
```

C In-order walk and databases

Typically we will not index every column in a database table. When a query involves columns that are not indexed, we may need to scan the whole column. This corresponds to an in-order walk in the B-tree that contains the actual table. The in-order walk algorithm in Section 3 is already an improvement over the straightforward recursive solution asymptotically, since its $O(1)$ time bound is worst-case instead of amortized. However, it is not clear that in practice our algorithm will be faster if we just use the simple implementation outlined in our pseudo-code.

We do want to briefly describe an observation about the hands and the idea of pre-fetching. In particular, our in-order walk algorithm suggests a pre-fetching schedule that seems implementatable. We can lock the upper portion of the hands and all the nodes referenced by those cells in the data cache (and as the height of the hand shrinks, we will also use pre-fetching to bring the lower cells and their associated spine lists into the cache). This guarantees that accessing any cells in that portion as well as the nodes referenced will not generate a cache miss. Also, we can use pre-fetching in ExtendRightLeftSpine since the right-left spine s_{rp} will not be needed immediately. Further, it is possible to do the eager walk "over-eagerly" by extending Invariant 3.2 to mandate the full right-left spines to be stored for the top few cells. Finally, we can associate second level spines to the cells in the spine lists. This corresponds to a very aggressive pre-fetching schedule that guarantees all the nodes required in the near future are pre-fetched by the hands.

We note that, however, in practice more sophisticated trees are used in databases. For example, B*-trees is designed with fast scanning in mind. Since these trees use more space than a B-tree, their cache performance may be worse than a simple B-tree with the hands. It will be interesting to implement and benchmark the performance of using the hands and see what our theoretical result would translate to in the real world.