

Finger Search on Balanced Search Trees

Maverick Woo
maverick@cs.cmu.edu

April 7, 2006

Abstract

This thesis introduces the concept of a *heterogeneous decomposition* of a balanced search tree and apply it to the following problems:

- How can finger search be implemented without changing the representation of a Red-Black Tree, such as introducing extra storage to the nodes? (Answer: Any degree-balanced search tree can support finger search without modification in its representation by maintaining an auxiliary data structure of logarithmic size and suitably modifying the search algorithm to make use of this auxiliary data structure.)
- Do Multi-Splay Trees, which is known to be $O(\log \log n)$ -competitive to the optimal binary search trees, have the Dynamic Finger property? (Answer: This is work in progress. We believe the answer is yes.)

Thesis Committee

Guy E. Blelloch	Carnegie Mellon University, co-chair
Richard Cole	New York University
Bruce M. Maggs	Carnegie Mellon University, co-chair
Daniel Dominic Kaplan Sleator	Carnegie Mellon University

1 Introduction to Finger Search

Consider a search tree data structure that contains n sorted keys a_1, a_2, \dots, a_n . The *rank* of the key a_i is defined to be its position in the sorted list of keys, which in our notation is simply i . A search is considered successful if it is made to a key that is present in the tree. Consider a sequence of successful searches made to the tree. We say that the data structure supports *finger search* if we can search for a_i in $O(\log|i - j|)$ time¹ where j is the rank of the the preceding search target, or 1 if this is the first search of the sequence. In general, we allow the time bound to be worst-case, amortized, or in the expected case where the expectation is taken over the random bits used to build the data structure.

Comparing finger search to the “classical” search which starts at the root and generally takes $O(\log n)$ time on a balanced search tree, we can see that finger search provides a tighter performance guarantee when the search sequence exhibits spatial locality of reference, i.e., when successive search targets are close to each other in rank.

A simple application that benefits from the use of finger search would be *scanning*, namely a sequence of n searches targeting the keys in sorted order one by one. On a search tree that supports finger search, scanning takes $O(n)$ time since the rank of each successive search target only increases by one. This compares favorably against the “classical” bound of $O(n \log n)$.

A more sophisticated application of finger search is the merging² of two sorted lists of sizes m and $n \geq m$ in the information theoretically optimal $O(m \log \frac{n}{m})$ time. Even though this problem can be solved on AVL Trees [1] using a complicated algorithm [6], a conceptually much simpler solution can be obtained on (2, 3) Trees [2] using finger search [7].

1.1 A Brief History of Finger Search

The concept of finger search was first introduced on a variant of B-Trees [3] by Guibas et al. [11] in 1977 specifically for applications that exhibit spatial locality in their search sequences. Their data structure is arguably quite complicated for that period³, and their model of finger search can also be slightly less intuitive than the other newer work that follows. In their design, a finger has to be setup and the setup cost is logarithmic to the number of keys. Once a finger has been setup, finger searches can be started from it. Consequently, if we set up k fingers, then a finger search can be

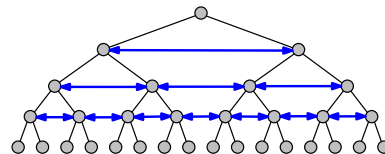
¹The usual disclaimer that $\log(x)$ is a shorthand for $\max(1, \log(x))$ applies.

²If only searches are required, then other solutions such as hash tables may also be used. Merging is a prime example showing the versatility of search trees that support finger search.

³Using a more modern terminology, their data structure is a leaf-store (8, 24)-Tree, with a redundant-counter regularity constraint on a root-to-finger path. Multiple fingers are possible, which causes the corresponding regularity constraints to interact with each other.

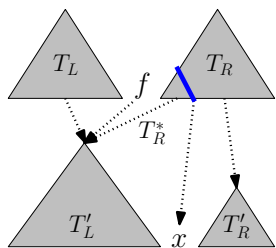
implemented $O(k + \log d)$ time, where d is the rank distance between the target and the closest finger.

Since then, finger search based on modifying balanced search trees has been studied by many researchers, e.g., Brown and Tarjan [7, based on (2, 3) Trees] and also Huddleston and Mehlhorn [13, based on (a, b) Trees]. These two designs support finger search by augmenting the tree with parent pointers in each node, as well as extra pointers called “level-links” which form a doubly-linked list among the nodes at the same depth. With the presence of the level-links and parent pointers, it’s not hard to show that a path of length $O(\log d)$ exists between any two keys that differ in rank by d . Finger search can then be implemented by simply maintaining a pointer to the node that contains the most recent search target.



Along this line of design, randomized search structures including Skip Lists by Pugh [16] and Treaps by Seidel and Aragon [17] can also be shown to support finger search, but with no modification done to the data structure itself. Instead, one can show that the length of the path between two keys that are d apart in rank is expected to be $O(\log d)$, where the expectation is taken over the random bits used to build the data structure.

A different design has been proposed by Tarjan and Van Wyk [21, based on Red-Black Trees]. Instead of maintaining a finger pointer to any node inside the tree, they maintain a triple (T_L, f, T_R) . The singleton f is the most recent search target. The Red-Black Tree T_L contains all keys smaller than f while T_R contains all keys larger than f .



Suppose the finger needs to be moved to a new target x on its right. First, T_R will be split at x , creating (T_R^*, x, T'_R) . Then (T_L, f, T_R^*) will be joined to create the new T'_L , restoring the triple representation (T'_L, x, T'_R) . Moving the finger to the left is analogous. Their proof then goes on to show that split and join can be implemented in time logarithmic to the size of T_R^* , which is precisely the difference between the ranks of f and x .

As this design is conceptually maintaining the finger at the farthest end of the two trees and therefore favor these two positions and their vicinities, Tarjan and Van Wyk proposed to call all similar designs “heterogeneous”, in contrast to the other “homogeneous” designs that can favor any position. Other notable heterogeneous designs include a modification based on AVL Trees by Tsakalidis [22], the general search structure involving a collection of (2, 3) Trees by Kosaraju [15] and the purely-functional catenable sorted list of Kaplan and Tarjan [14].

All of the above heterogeneous designs use restructuring rules that are carefully chosen to support finger search. In this regard, Splay Trees by Sleator and Tarjan [18] are truly in a different class. Their restructuring rule, know as splaying, is extremely

simple and is arguably *not* designed to accommodate finger search in any obvious way. As we will see, the fact that Splay Trees support finger search is merely a consequence of a much more ambitious goal which Sleator and Tarjan attempted to achieve.

In their 1985 paper, Sleator and Tarjan conjectured that Splay Trees have the finger search property. Specifically, their Dynamic Finger Conjecture on Splay Trees is that a sequence of m “classical” searches, each followed by splaying the target to the root, takes $O(m + n + \sum_{i=1}^m \log d_i)$ time, where d_i is the difference between the rank of the $(i - 1)$ -th target and that of the i -th target. This conjecture, in the case when m is $\Omega(n \log \log n)$, was subsequently proven in 1995 by Cole [8] using a long and careful argument that spans over 60 pages.

Finally, researchers have also considered more sophisticated search structures that support multiple fingers and worst-case $O(1)$ time insertion and deletion at any given finger location. For more information on this line of work, see [5] and references within.

1.2 Finger Search and This Thesis

Homogeneous vs. Heterogeneous Finger Search. The Heterogeneous Red-Black Trees of Tarjan and Van Wyk [21] are closely related to the traditional Red-Black Trees [12] in the following sense. Given a Red-Black Tree T and a key f in T , first perform the split operation on T at f to obtain (T_1, f, T_2) . The left spine of a tree is defined to be the root of the tree together with the left spine of the root’s left subtree. The right spine is defined analogously. For each node on the left spine of T_2 , introduce a parent pointer that points from the node to its parent and let this augmented tree be T'_2 . Perform a similar operation on the right spine of T_1 to obtain T'_1 . As it turns out, (T'_1, f, T'_2) is one of the many possible representations of the Heterogeneous Red-Black Tree that Tarjan and Van Wyk would maintain when the finger is at f . The non-uniqueness of this transformation is merely a consequence of having multiple possible representations of a Red-Black Tree over the same set of keys. It is in fact possible to implement the algorithm by Tarjan and Van Wyk to obtain exactly the same structure as we would obtain from the transformation above.

Our observation is that conceptually there is very little difference between the heterogeneous design, which splits T on the access path to f to obtain two simple “inverted spines” as above, and the homogeneous design, which adds level-links and parent pointers to all nodes in T . The reason is that in the homogeneous designs, we only use the level-links that are close to the access path to f during a finger search (with an appropriate definition of “close”) and so the level-links can be seen as part of an implicit representation of the inverted spines. By making this representation explicit, we obtain a new way to implement finger search homogeneously by storing the two inverted spines in an auxiliary data structure instead of modifying the representation of T . More details of this auxiliary data structure—called the Hand—will be provided in Sections 2 and 3.

Dynamic Finger Conjecture on Multi-Splay Trees. Besides the Dynamic Finger Conjecture, Sleator and Tarjan [18] have also conjectured that Splay Trees are $O(1)$ -competitive, namely that to serve any any fixed search sequence, the time a Splay Tree takes is within a constant multiple of the time required by an optimal algorithm. Known as the Dynamic Optimality Conjecture on Splay Trees, this problem has remained famously open since its proposal over 20 years ago.

In fact, it wasn't even until 2004 when Demaine, Harmon, Iacono and Pătraşcu [9] introduced the first search tree that is $O(\log \log n)$ -competitive. Their design is dubbed Tango. While any balanced search tree of logarithmic height is trivially $O(\log n)$ -competitive, no $o(\log n)$ -competitive design is known before Tango was introduced. (Even though Splay Trees are conjectured to be $O(1)$ -competitive, as of this writing it is still not known if they are $o(\log n)$ -competitive.)

Subsequent to Demaine et al., Wang, Derryberry and Sleator [23] came up with Multi-Splay Trees, which can be viewed as a refinement over Tango. Besides also being $O(\log \log n)$ -competitive, Multi-Splay Trees have a distinctive advantage over Tango from a theoretical point of view. More specifically, it is easy to show that Tango cannot be $O(1)$ -competitive, but the question on Multi-Splay Trees remains open.

There is in fact an implicative relationship between the Dynamic Optimality Conjecture and Dynamic Finger Conjecture. Specifically, it is well-known that any $O(1)$ -competitive search tree must support finger search.⁴ This leads us to the other part of this thesis in which we attempt to make progress in settling the Dynamic Finger Conjecture on Multi-Splay Trees.

While it may be argued that checking a necessary but not sufficient condition to the Dynamic Optimality Conjecture on Multi-Splay Trees provides little insight to the problem, it can still be assuring to know that such condition actually holds. Furthermore, in view of the length and complexity of Cole's proof of the Dynamic Finger Conjecture on Splay Trees, it would be interesting to see if we can eventually prove this conjecture on Multi-Splay Trees in a way that is substantially easier to understand. As of this writing, Multi-Splay Trees are believed to have almost all the well-known properties⁵ that have been proven on Splay Trees except Dynamic Finger. Therefore, if we eventually settle the conjecture on the positive side, then Multi-Splay Trees can be regarded as an interesting alternative to Splay Trees, at least in the theoretical sense. Finally, it is also our hope that our work on Multi-Splay Trees would eventually lead to a simpler proof of the Dynamic Finger Conjecture on Splay Trees, but we must remark that we presently do not have any concrete idea on how this may happen yet.

⁴It appears that the proof to this "folklore" has not been written down anywhere except being noted as non-trivial. This could be partly due to the large amount of details potentially required in such a proof. However, we note that a substantial part of this proof has in fact been given implicitly in the work of Kaplan and Tarjan [14].

⁵These include $O(\log n)$ time access, Static Optimality, Working Set and Static Finger.

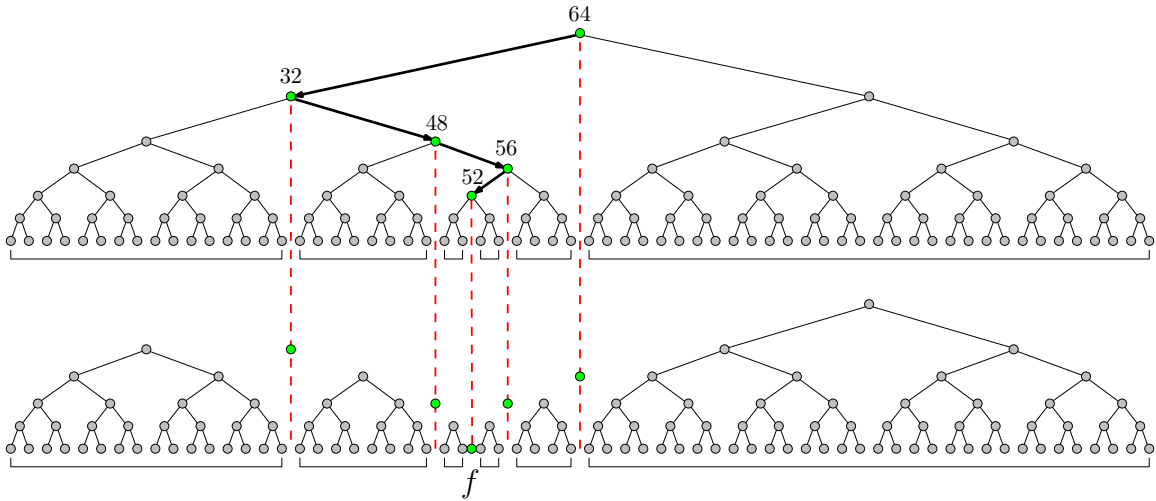


Figure 1: Heterogeneous Decomposition with respect to f (See Section 2.2)

2 The Heterogeneous Decomposition

Consider a search tree T and a key f in T . For this exposition, we assume that T is a binary tree and hence we will make no distinction between a key and the node containing that key. It is also convenient to refer to the key of rank i simply as i .

The *access path* of f is simply the path starting from the root of T to f . The *heterogeneous decomposition* of T with respect to f is a list of subtrees of T interleaved by the nodes on the access path of f , recursively defined as follows.

Let $\text{decomp}(f, x)$ be the heterogeneous decomposition of the subtree rooted at x with respect to f . If x equals to f , then return the list $(x.\text{left}, x, x.\text{right})$. Otherwise, the access path of f continues either to the left or right of x . In the former case, return $(\text{decomp}(f, x.\text{left}), x, x.\text{right})$. In the latter, return $(x.\text{left}, x, \text{decomp}(f, x.\text{right}))$. Intuitively, we are simply listing the subtrees hanging off the access path of f from left to right, with the nodes on the access path inserted between subtrees so that the list is in the sorted order in the rank space.

2.1 Heterogeneous Height

The heterogeneous decomposition is probably not very exciting to readers who are familiar with search trees. The novelty is in the definition of the *heterogeneous height* of the nodes based on this decomposition.

The following is stated for degree-balanced search trees where all the leaves of T are at the same depth. The definition for height-balanced search trees can be obtained in a similar spirit but will be less elegant.

Consider the heterogeneous decomposition of T with respect to f . Let $\text{hh}(f, x)$ denote the heterogeneous height of x in this decomposition. $\text{hh}(f, f)$ is defined to be

1. In what follows, we will inductively define hh for the elements of the sublist to the right of f . The definition for the elements of the sublist to the left of f is symmetric.

Let the list be (T_R^x, y, R') , where T_R^x is the subtree of T that appears at the head of the sublist and y is the key following it. R' is the remaining tail on which we recurse. It can be proven that T_R^x is in fact the right subtree of the node x that precedes the sublist, explaining our choice of notation. We define the heterogeneous height of any node in T_R^x to be its height in T_R^x . As for $\text{hh}(f, y)$, it is defined to be one plus the height of T_R^x , which can be proven to be the (real) height of x in T .

2.2 The Running Example

Figure 1 shows an example of the heterogeneous decomposition of a complete binary search tree of 127 nodes at the node f , which is at rank 52. The upper part of the Figure shows the original tree, with some of the nodes labelled with their ranks for easy identification. The lower part shows a collection of subtrees interleaved with nodes on the access path to f . By following the definition we stated above, we obtain the following list as our decomposition of the tree:

$$(T_L^{32}, 32, T_L^{48}, 48, T_L^{52}, \mathbf{52}, T_R^{52}, 56, T_R^{56}, 64, T_R^{64})$$

(The finger has been typeset in boldface for clarity.)

In Figure 1, the nodes on the access path to f have been colored green. These nodes have also been drawn at their respective heterogeneous heights, although technically the decomposition is simply a list of keys and subtrees of T and does not involve heights. The vertical dashed red lines are drawn to assist identifying the subtrees in the decomposition. In particular, between every two red lines lies a subtree that forms part of the decomposition.

3 The Hand

Assume that T is a complete binary search tree and f is a key in T . The Hand data structure is designed to represent the implicit “inverted spines” in the heterogeneous decomposition of T with respect to f . We will explain what the implicit inverted spine on the right of f is and how it is represented. The one on the left is defined analogously.

First let us introduce two definitions. The *right parent* of a key x is the smallest key on the access path to x that is larger than x . The *right parent stack* is a stack representing the right ancestors of x , with x at the top of the stack followed by its right parent, its right grandparent and so on. In our example in Figure 1, these include the node f (rank 52) and the other green nodes to the right of f .

Associated with each key on the right parent stack is another stack that represents a prefix of the left spine of the subtree to the right of the key. The top of each stack is the lowest node in the corresponding prefix. The length of the prefix, measured in nodes, is defined to be the difference in the heterogeneous height of the key and

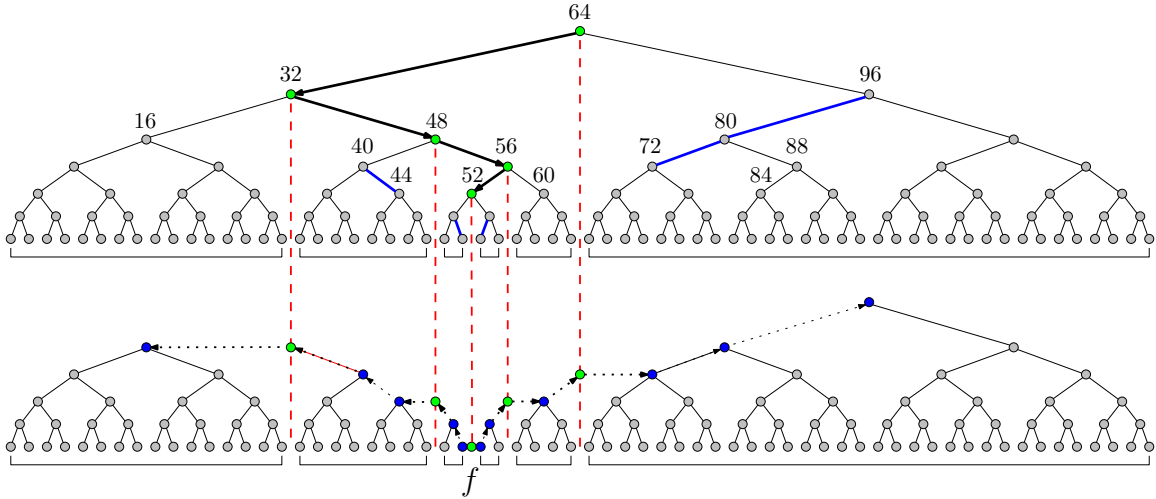
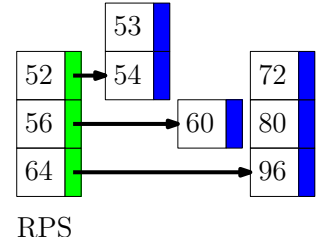


Figure 2: Visualizing The Hand at f

that of its right parent. This is one of the two invariants governing the content of the Hand.

Let's consider f and work out the length of the spine prefix that is associated with f to its right. The heterogeneous height of f is 1 and that of its right parent is 3. Therefore, the prefix should have $3 - 1 = 2$ nodes in it. A similar calculation for the key at the root (rank 64) of T shows that its associated prefix should have three nodes, whereas the one for the right parent of f (rank 56) should have one.



To help us visualize these spine prefixes, we have annotated Figure 1 and turn it into Figure 2. In the upper part, we have colored the edges that are in the spine prefixes in blue, so are the corresponding nodes in the lower part. The edges in the prefixes have also been changed to upward dotted arrows to indicate the stack nature of how the nodes are stored in the Hand. The horizontal dashed arrows are added to complete the inverted spines, even though they do not correspond to any edges in T . These arrows will be explained in Section 3.1.

A Second Invariant. In order to support moving the finger forward and also backward, we have to maintain some extra pointers between the right parent stack and the left parent stack of f . These pointers are controlled by the second invariant but we will suppress the details here.

3.1 Finger Search Using the Hand

With the inverted spines stored in this stack of stacks representation, finger search can be implemented in two phases. For illustration, let us assume that the target x is at rank 84.

In the first phase, we scan the right parent stack for the node w whose right subtree S_R^w contains x . We may imagine that our finger has now arrived at w in the heterogeneous decomposition and is about to enter the left spine of S_R^w at height $\text{hh}(w)$. Due to a lack of good variable name that preserves the symmetric order in all cases, we will call this node on the left spine of S_R^w the *entry node*. In our example, w would be 64, $\text{hh}(w)$ is 4 and the entry node is 72. Notice that so far the blue nodes on the spine prefixes are not used in this first phase. Only the green nodes are used and they are all on the right parent stack.

Now *if only* our finger can really get to the entry node in constant time, as suggested by the horizontal dashed arrow pointing towards 72 in Figure 2, then we may really continue the search from this entry node. The spine prefixes are precisely designed to make this happen. In particular, in the Hand built at f , the spine prefix associated with w will be exactly long enough to reach the entry node, which is exactly at the top of the stack representing the prefix. As such, our imaginary journey of the finger can indeed continue into the second phase, of which there are two simple cases.

In the first case, the target x is actually contained in the subtree rooted at the entry node. The finger can thus proceed to traverse down as in a normal binary search tree. Otherwise, the finger will continue traversing up the spine prefix associated with w until the first case applies. Back to our example in the search for 84, this means the finger will first traverse up from 72 to 80 and then starts traversing down its right subtree from there.

Readers familiar with the Heterogeneous Red-Black Trees of Tarjan and Van Wyk will notice that the situation is completely analogous.⁶ This, of course, is no coincidence given how the design of the Hand was inspired. The proof of the running time bound is also similar by noting that a target of rank d away is contained in the right subtree rooted at a node at heterogeneous height $O(\log d)$. It is also not hard to show that the invariants on the right parent stack and the spine prefixes can be restored in the desired time bound as the finger pointer moves to the new location. This completes the sketch of how to use the Hand for finger search.

We note that the above description for the Hand can be generalized to handle any degree-balanced search trees. Furthermore, during insertions and deletions in T , the Hand can also be updated in time proportional to the amount of structural changes (node split, node fusion and children sharing) of T . From a theoretical perspective, this allows us maintain the Hand for supporting finger search without introducing any asymptotic overhead to the update of the underlying search tree. The missing detail in this Section has been reported in [4].

⁶A slight difference is that we have deliberately chosen to use only the green nodes to get to w in the first phase, even though a simpler algorithm could be have if we used the blue nodes as well. Our admittedly strange choice will be explained when we get to Section 4.4.1.

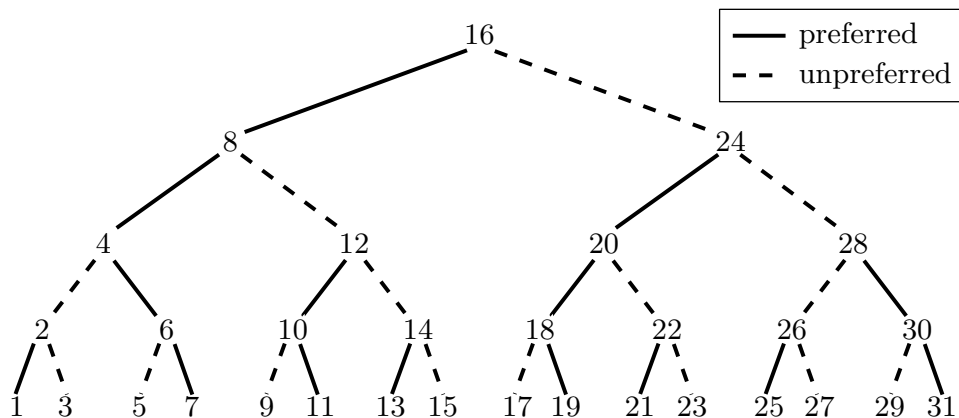


Figure 3: An Example Reference Tree of 31 Nodes

4 Finger Search on Multi-Splay Trees

The present document only provides an overview of Multi-Splay Trees. The construction is described with reasonable details but the proofs will not be presented. The latter can be found in [23].

4.1 The Interleave Lowerbound

The fact that Multi-Splay Trees are $O(\log \log n)$ -competitive is actually inherent in their design, with the competitiveness proof built into the data structure. As in many cases in competitive analysis, this proof involves a lowerbound on the cost of the optimal algorithm. It will be convenient to state this lowerbound before we describe the construction of Multi-Splay Trees since they are closely related.

Consider a fixed binary search tree S of n nodes and a fixed length- m search sequence $\sigma = \sigma_1, \sigma_2, \dots, \sigma_m$. Using the terminology of online algorithms, we say that S “serves” σ by performing a search for each σ_i . Each search starts from the root of S and there is no restructuring in S after each search.

Preferences. Supposed S has just served σ_i for some i . At this point, we say that a node x “prefers left” if the most recent search in $\sigma_1, \sigma_2, \dots, \sigma_i$ that involves the subtree rooted at x is made either to x itself, or to a node that is in the left subtree of x . In other words, in the very last time a search goes through x , the search either stops at x , or it continues to the left. If neither of these two cases applies, then we say that x “prefers right”.

Initially, all nodes are considered to prefer left. This choice is arbitrary and can be shown to be inconsequential when σ is sufficiently long. Notice that a search ending at the node x can actually change its preference to “left” had its preference been “right” prior to this search.

Figure 3 shows a reference tree of 31 nodes organized into a complete binary search

tree. Notice that S does not have to be of any particular shape as long as it is fixed. The shape in Figure 3 is merely our choice. The tree edges have been drawn to reflect the preferences of the nodes at that moment. Each internal node prefers to the side where its solid child pointer resides.

Switch. A node may change its preference when a search path goes through it. Each time when a node x changes its preference, we call that a “switch” at x . Consider the reference tree in Figure 3. If the next search target is 17, then a left-to-right switch at 16 and a right-to-left switch at 18 will be made.

Interleave Bound. For any given σ and any given S , we can compute the total number of switches that occurred in S while serving σ . That is, summing over each node in S , the total number of times a node changes its preference in the duration of serving σ . This sum is known as the Interleave Bound, denoted $IB(S, \sigma)$. Wilber [24] showed that the $IB(S, \sigma)$ is in fact a lowerbound, up to constant factors, on any binary search tree⁷ T to serve σ even when T can restructure itself in-between searches.

4.2 Defining Multi-Splay Trees

The design of Multi-Splay Trees, as well as Tango, is closely related to the above construction in the following sense: Both of them are concrete representations of an imaginary complete binary search tree S , which is called the *reference tree*. In other words, given a Multi-Splay Tree, we may reconstruct the corresponding reference tree from the state of the Multi-Splay Tree.

Observe that the preferences of the nodes in S naturally define a path decomposition of S . The root of S has a solid path all the way to a leaf, where every internal node on this path prefers the direction of the next node on this path. Each of the internal nodes on this path also has a non-preferred child, which is the root of its own subtree. We decompose each of these subtrees recursively.

Of the two child pointers at a node, the one that points to the preferred child is called “solid”. The other one is called “dashed”. Using this terminology, S has now been decomposed into solid paths connected by dashed edges. This is clearly illustrated in Figure 3.

In a Multi-Splay Tree, each of these solid paths are simply represented as a Splay Tree of the same shape and the same keys. Observe that when the collection of these totally solid Splay Trees is connected with the dashed pointers, it has exactly the same shape as S and is furthermore a valid binary search tree containing exactly the same set of keys with S . This is perhaps self-evident in Figure 3, which can now be viewed as a possible shape of a Multi-Splay Tree containing 31 keys.⁸ Let this “tree of Splay Trees” be denoted by T .

⁷Care must be taken to define what are valid binary search trees. In our model, a node in a binary search tree can only contain a key, a left child pointer, a right child pointer and an $O(1)$ extra storage that do not contain pointers to any other nodes in the tree.

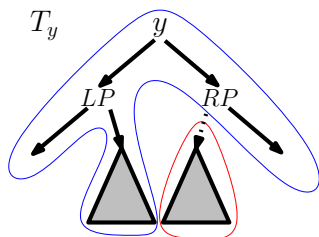
⁸The shape of the Multi-Splay Tree in Figure 3 is indeed possible according to the structural

We need a method to distinguish the dashed pointers so that we can operate on each of the Splay Trees individually. So, an extra “root bit” will be introduced to each node in T . The root bit of a node is set if the node is the root node of a Splay Tree in T .

To search for a key in T , we simply proceed as in a normal binary search tree without making any distinction to whether a pointer is solid or dashed. Since we may have to traverse $O(\log n)$ Splay Trees, each with $O(\log n)$ nodes, the worst-case search time would be $O(\log^2 n)$. This is of course an over-estimate of the time required. In fact, it has been shown that a search takes only amortized $O(\log n)$ time [23] using a proof similar to Sleator and Tarjan’s analysis of Link-Cut Trees [18].

Restructuring. After each search, we will have to restructure T to reflect the preferences of the nodes in S . Every dashed pointer that we traversed in the search path in T corresponds to a switch in S . Indeed, the fact that a dashed pointer is traversed implies that the same search, when executed in S , will have to leave a solid path and continue in another. The key observation is that each switch can be implemented by several splits and joins of the Splay Trees involved.

Suppose a left-to-right switch is needed on a node y in S and it is contained in the Splay Tree T_y , which represents the solid path in S that contains y . This means the solid subpath below y in S has to be split off and the solid path that starts at the previously unpreferred right child of y has to be joined in. This is accomplished in five steps as follows.



First, we splay y the root of T_y . Then, if the left parent of y is present in T_y , we splay the left parent to become the left child of y . Similarly we splay the right parent of y to become the right child of y if it is present. At this point, it can be shown that the right subtree of the left child (LP) of y in T_y contains precisely the subpath that we want to cut off and that the left child pointer of the right child (RP) of y in T_y is dashed and points to the Splay Tree corresponding to the solid path that we want to join into T_y . By marking and unmarking the root bits of the root of these two subtrees respectively, T_y has now been updated to contain the desired path.

Intuitively, since each Splay Tree has only $O(\log n)$ nodes, a split or join—both implemented by a splay operation—should only take $O(\log \log n)$ time. Now recall that the number of switches is a lowerbound on the running time of the optimal algorithm. As the restructuring of T due to each switch can be implemented in $O(\log \log n)$ time, and the restructuring cost in each of the Splay Trees dominates

invariants of Multi-Splay Trees. At the very least, it could be the initial configuration. However, while serving a sequence of searches, not all configurations can be produced by the restructuring. While we did not work out the details to verify, the author’s intuition is that it is unlikely, if not impossible, to occur.

over the search cost, T should be $O(\log \log n)$ -competitive and this concludes our sketch.

4.3 The Dynamic Finger Property

Before we go on to discuss our ideas on the Dynamic Finger Conjecture on Multi-Splay Trees, let us remark on a major difference between Multi-Splay Trees (and also Splay Trees) and other search trees that support finger search. For readers who are familiar with Splay Trees and The Dynamic Finger Conjecture, please skip this Section.

In traditional search trees, the concept of finger search arises because we do not use a “classical” binary tree search that starts at the root of the tree. Instead, a finger search starts at the node that contains the key of the most recent search. Hence, the finger path and the access path to a key are generally different.

In Splay Trees and Multi-Splay Trees, however, a search always starts at the root and ends at the target, making the finger path and the access path exactly the same. This cannot be a good idea in classical search trees. The trick in Splay Trees and Multi-Splay Trees is to perform restructuring after each search. In particular, both of these designs will bring the target to the root using their own method of restructuring. So in a way, while it’s true that we start the search at the node that contains the most recent target, the concept of “finger search” is merely referring to a property of the running time incurred in a sequence of searches and the restructuring that follows. Again, there is no operation known as “finger search” on Splay Trees or Multi-Splay Trees.

From the perspective of finger search, we may interpret the restructuring as an attempt to bring the keys that are in the vicinity of the targets of a sequence of clustered⁹ searches closer to the root. Consequently, the time incurred in any sequence of searches may possibly be bounded by the sum of the logarithm of the differences in the target ranks. The fact that Splaying has this curious property in no way obvious, although the Dynamic Finger Theorem by Cole [8] does assert its truthfulness. Our question is whether Multi-Splaying has this very same property or not.

4.4 Current State of Affairs

First let us report on two simple cases.

Repeated Searches. If one wants to know if Multi-Splaying has the Dynamic Finger property, perhaps the first thing that comes to mind is whether repeated searches to the same key takes $O(1)$ time per search. As a Multi-Splay Tree always restructure itself so that the last search target is at the root, this does not bother us.

⁹Observe that if the search targets are always far apart in rank, say, $\Omega(n^\epsilon)$ for some $\epsilon > 0$, then any balanced search tree that provides logarithmic time searches, including Splay Trees, can be claimed to have the Dynamic Finger property.

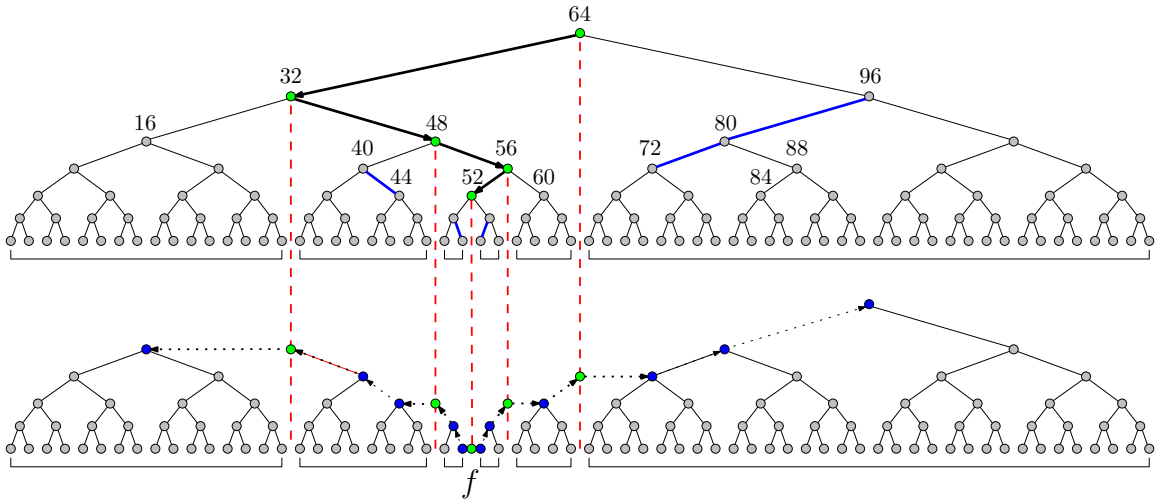


Figure 4: Figure 2 reprinted here for convenience

(This, by the way, is a straightforward way to show that Tango does not have the Dynamic Finger property and hence also not Dynamically Optimal.)

Scanning. Perhaps the second thing to check is whether Multi-Splay Trees support scanning. More precisely, on a Multi-Splay Tree that contains n keys, perform a sequence of n searches targeting the n keys in their sorted order. As the rank of each successive search target only increases by one, the Dynamic Finger property would imply that this sequence takes $O(n)$ time.

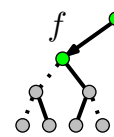
The fact that scanning takes linear time has already been shown in the original paper that proposes Multi-Splay Trees. [23] Their proof seems straightforward when compared to the three proofs of the scanning theorem on Splay Trees (Tarjan [20], Sundar [19] and Elmasry [10], with Sundar’s being the simplest of the three in our opinion). This is arguably a good sign for us since it suggests that Multi-Splay Trees may be easier to analyze in general.

4.4.1 Multi-Splay Trees and the Hand

In view of the complexity of Cole’s proof, a reasonable question to ask is if there is any reason that Multi-Splay Trees can afford a simpler proof. The punchline of this Section is going to be that, due to the tremendous structure in Multi-Splay Trees, the proof may be quite similar to what we did in [4]. More specifically, we will consider the heterogeneous decomposition of the reference tree and use the Hand to guide our amortization. We will now attempt to develop this connection in the rest of this Section.

Let us again refer to Figure 2 to gain some intuition. (Figure 4 is a reprint of Figure 2.) For this Section, the upper part of the Figure should be regarded as the

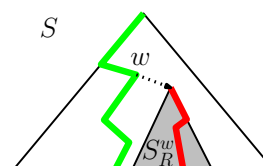
reference tree S and the lower part is the heterogeneous decomposition of S with respect to the last search target f , which is at rank 52. The preferences in most of the nodes in S will not matter for this presentation, but to be consistent with the fact that the last search target is f , the ancestral path of f must be solid. To complete the solid path that contains f all the way to a leaf, let us assume that f 's current preference is to its right, and this right child prefers to the left. Furthermore, let the left child of f prefers right.



Let T be a Multi-Splay Tree that implements S . Consider the case when the next search target is to a key x that is d rank away to the right of f . For the sake of illustration, let us consider the running example with x being 84. The reader may recall that we have already considered this in the context of the Hand in Section 3.1. In fact, we will again break down the discussion into the two phases by analyzing the finger path in the heterogeneous decomposition of S . But first, we will present how to understand the finger path in the heterogeneous decomposition from the perspective of preference switching in S .

The Switching Perspective. Recall that the number of switches due to a search for x is the number of preference changes among the nodes on the access path to x on the reference tree S . This path always starts at the root but generally the root is not the finger position. To see the relationship between the finger path and the access path, we bring in the heterogeneous decomposition.

Suppose we are following the access path from the root of S to x before any switch has been done. Consider the solid path decomposition of S and in particular, the current solid path of the root. If x is on this solid path, then no switches will be necessary to reach x . (There can still be one switch at x due to the invariant that an access to a node will cause it to prefer left.) But if x is not on this solid path, then consider the lowest node w on this solid path that is also an ancestor of x . We call w the *exit node*. It is easy to see that the exit node must exist because the root of S is always a candidate if no lower node suffices. It should also be noted that the exit node can be f itself. This can happen if f prefers left at the moment and x is in the right subtree of f in S .



In the case when the finger is going forward, as in our running example, the switch on w always goes from left to right. The trick is to observe that S_R^w , the right subtree of w in S , follows w in the heterogeneous decomposition. Once the access path reaches the root of S_R^w , all the remaining switches occur inside S_R^w . What we will do is to analyze the switch at w , which is the highest, and the rest of the switches in S_R^w as two groups.

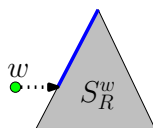
Reaching w . In the first phase, the finger goes up the green nodes on the inverted spine of the heterogeneous decomposition to identify the node w whose right subtree S_R^w contains x . Observe that we only use the green nodes, just like our algorithm in Section 3.1. In our example, w is 64 and x is 84.

By construction, the subtree containing the root of a Multi-Splay Tree T actually represents the solid path that contains the most recent search target f . Let this subtree in T be denoted $Sp(S)$, the Splay Tree that contains the solid path of S . Given our assumption about the node preferences, these are exactly the green nodes together with the right child of f and its left child in Figure 2.

Observe that regardless of the choice of f , the nodes in $Sp(S)$ must “spread” across the ranks, both to the left and to the right of f . In particular, the distances between these nodes are in essence geometrically distributed, modulo the blue nodes on the spine prefixes that are missing from $Sp(S)$. In fact, $Sp(S)$ contains both the left and right parent stacks of f , both of which are the nodes of the inverted spines that we use to move the finger up. In other words, it suffices to touch only the green nodes to get to w .

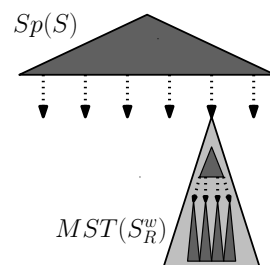
Our plan is to assign the weights to the nodes in $Sp(T)$ such that splaying a node that is far away in rank has a cost proportional to the logarithm of the rank difference. Not surprisingly, our weight assignment is based on the heterogeneous height of the nodes. In general, for a node y at heterogeneous height $hh(y)$, its weight will be $1/2^{hh(y)}$. Therefore, $O(hh(w))$ credits will be sufficient to pay for this phase.

Going back to our example, we have to make a left-to-right switch at 64. This amounts to splaying 64 to the root of $Sp(S)$, at a cost of 4.



Entering S_R^w . At the beginning of the second phase, the finger is on w and in the context of the Hand, it will enter S_R^w at the entry node on the left spine of S_R^w . Note, however, that this is *not* what happened in the reference tree S where the switches are counted. In particular, the access path entered from the root of S_R^w . The good news is that all the switches that occur within S_R^w can be analyzed as a group.

We will rely on a structural property of Multi-Splay Trees that is inherent in their recursive construction. In particular, the connected part of T that represents S_R^w can be considered, in isolation, as a valid Multi-Splay Tree containing exactly the same keys as in S_R^w . It has been proven in [23] that a search in a Multi-Splay Tree containing n keys takes amortized $O(\log n)$ time. Therefore, if we can show that we can view this connected part in isolation, then intuitively the cost to access x in S_R^w can be bounded by the height h of S_R^w .



As we have already noted before, the difference between h and $hh(w)$ is in fact bounded by the length of the spine prefix associated with w .

In the context of the Hand, this difference has already been deamortized in the sense that the spine prefix is already built by the time we get to f . In other words, we showed a careful scheduling that will incrementally build this spine prefix prior to

the finger’s arrival at f . Therefore, on its way to x , the finger can make use of this spine prefix.

In the context of Multi-Splay Trees, the same scheduling will be used. However, the time we have scheduled for building this spine prefix will instead be saved as credits. These credits will be spent the first time the access path enters S_R^w to pay for this difference between h (the “actual amortized” cost) and $\text{hh}(w)$ (the “amortized amortized” cost).¹⁰ Therefore, the $O(\text{hh}(w))$ allocated credits can pay for the cost of accessing x within S_R^w , which can then be shown to dominate the cost of this phase.

Future Work. The above sketch contains all the necessary intuition for handling the situation when the finger is constantly moving only in one direction. However, it does not cover the case when the finger is moving back and forth, especially in a small rank space. We are still working on a careful argument to handle this case.

Here we also provide a short list of details that we have suppressed in the sketch above, all of which are necessary even just for the forward-only case.

- The two phases in our description do not have a direct correspondence to the actual restructuring done in a Multi-Splay Tree. The notion that we can view the second phase in isolation is merely an intuition. Fortunately, this conceptual difference only affects two rotations during the restructuring.
- We did not mention the issue of node reweighting because it requires detailed description of the structure of a Multi-Splay Tree and we did not provide such a description.

On a high level, note that the weight assignment we use on the nodes in the inverted spines is such that deeper nodes are cheaper to access. However, in the proof of the logarithmic time access in [23], the weight assignment used will cause an access to a deeper node to be more expensive.

Fortunately, these two weight assignments do not conflict with each other because they apply to two disjoint groups of nodes in the heterogeneous decomposition. In particular, only nodes on the inverted spines require our “inverted” weight assignment. But as nodes join and leave the inverted spines, their weight assignments have to be changed accordingly. This will require a careful argument and corresponds to restoring the two invariants of the Hand after a finger search.

5 Time Line

As we have already noted, the results in Section 3 regarding the Hand has already been reported in [4] back in 2003. As it turns out, by introducing the Heterogeneous

¹⁰We dub this the concept of reamortization but we will not expand on it in this document.

Decomposition into the description of the finger search algorithm based on the Hand, we managed to conceptually simplify it by quite a bit. Therefore, we plan to update the write-up in the thesis to reflect our updated understanding of the Hand.

Our on-going work in the Dynamic Finger property on Multi-Splay Trees is new and it is our top priority to finish the proof as soon as possible. As of this writing, together with Jon Derryberry, Danny Sleator and Chris Wang, we are preparing a conference submission about the various newly-proven properties of Multi-Splay Trees, one of which we hope would be the Dynamic Finger property. Our hope is to submit it to SODA 2007.

Our current plan, which is admittedly optimistic, is to finish the proof and also the write-up of the thesis by the end of July 2006.

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](#)
- [3] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972. [1.1](#)
- [4] G. E. Blelloch, B. M. Maggs, and S. L. M. Woo. Space-efficient finger search on degree-balanced search trees. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithm*, pages 374–383, 2003. [3.1](#), [4.4.1](#), [5](#)
- [5] G. S. Brodal, G. Lagogiannis, C. Makris, A. K. Tsakalidis, and K. Tsihlias. Optimal finger search trees in the pointer machine. *Journal of Computer and System Sciences*, 67(2):381–418, 2003. [1.1](#)
- [6] M. R. Brown and R. E. Tarjan. A fast merging algorithm. *Journal of the ACM*, 26(2):211–226, 1979. [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](#)
- [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. [1.1](#), [4.3](#)
- [9] E. D. Demaine, D. Harmon, J. Iacono, and M. Pătraşcu. Dynamic optimality—almost. In *Proc. 45th IEEE Symposium on Foundations of Computer Science*, pages 484–490, 2004. [1.2](#)

- [10] A. Elmasry. On the sequential access theorem and deque conjecture for splay trees. *Theoretical Computer Science*, 314(3):459–466, 2004. [4.4](#)
- [11] 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. [1.1](#)
- [12] 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. [1.2](#)
- [13] S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982. [1.1](#)
- [14] 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](#), [4](#)
- [15] S. R. Kosaraju. Localized search in sorted lists. In *Proc. 13th Annual ACM Symposium on Theory of Computing*, pages 62–69, 1981. [1.1](#)
- [16] W. Pugh. A skip list cookbook. Technical Report CS-TR-2286.1, University of Maryland, College Park, 1990. [1.1](#)
- [17] R. Seidel and C. R. Aragon. Randomized search trees. *Algorithmica*, 16:464–497, 1996. [1.1](#)
- [18] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32:652–686, 1985. [1.1](#), [1.2](#), [4.2](#)
- [19] R. Sundar. On the deque conjecture for the splay algorithm. *Combinatorica*, 12(1):95–124, 1992. [4.4](#)
- [20] R. E. Tarjan. Sequential access in splay trees takes linear time. *Combinatorica*, 5:367–378, 1985. [4.4](#)
- [21] 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](#), [1.2](#)
- [22] A. K. Tsakalidis. AVL-trees for localized search. *Information and Control*, 67:173–194, 1985. [1.1](#)
- [23] C. C. Wang, J. Derryberry, and D. D. Sleator. $O(\log \log n)$ -competitive dynamic binary search trees. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithm*, pages 374–383, 2006. [1.2](#), [4](#), [4.2](#), [4.4](#), [4.4.1](#), [4.4.1](#)

- [24] R. E. Wilber. Lower bounds for accessing binary search trees with rotations. *SIAM Journal on Computing*, 18(1):56–67, 1989. [4.1](#)