Algorithmica (2016) 76:1264–1275
DOI 10.1007/s00453-016-0224-x

The Power and Limitations of Static Binary Search Trees with Lazy Finger

Prosenjit Bose[1] · **Karim Douïeb**[4] · **John Iacono**[2] · **Stefan Langerman**[3]

Received: 29 June 2015 / Accepted: 24 September 2016 / Published online: 6 October 2016

Abstract A static binary search tree where every search starts from where the previous one ends (*lazy finger*) is considered. Such a search method is more powerful than that of the classic optimal static trees, where every search starts from the root (*root finger*), and less powerful than when rotations are allowed—where finding the best rotation based tree is the topic of the dynamic optimality conjecture of Sleator and Tarjan. The runtime of the classic root-finger tree can be expressed in terms of the entropy of the distribution of the searches, but we show that this is not the case for the optimal lazy finger tree. A non-entropy based asymptotically-tight expression for the runtime of the optimal lazy finger trees is derived, and a dynamic programming-based method is presented to compute the optimal tree.

P. Bose: Research supported in part by NSERC.
J. Iacono: Research partially completed at NYU School of Engineering with support from NSF Grants 1319648, 1229185 and 1533564. Research partially completed at Université Libre de Bruxelles with support from FNRS and the the Commission for Educational Exchange between the United States of America, Belgium, and Luxembourg. Research partially completed at and supported by MADALGO, a center of the Danish National Research Foundation.

✉ John Iacono
iacono@nyu.edu

1 Carleton University, Ottawa, ON, Canada

2 New York University, Brooklyn, NY, USA

3 Université Libre de Bruxelles, Brussels, Belgium

4 Forest, Belgium

1 Introduction

1.1 Static Trees

A binary search tree is one of the most fundamental data structures in computer science. In response to a search operation, some binary trees perform changes in the data structure, while others do not. For example, the splay tree [18] data structure performs a sequence of rotations that moves the searched item to the root. Other binary search tree data structures do not change at all during a search, for example, red-black trees [13] and AVL trees [1]. We will call BSTs that do not perform changes in the structure during searches *static* and call trees that perform changes *BSTs with rotations*. In this work we do not consider insertions and deletions, only searches in the comparison model, and thus can assume without loss of generality that all structures under consideration store the integers from 1 to n and that all searches are to these items.

We consider two variants of static BSTs: root finger and lazy finger. In the classic method, the *root finger* method, the first search proceeds from the root to the item being searched. In the second and subsequent searches, a root finger BST executes the searches in the same manner, always starting each search from the root. Here we consider *lazy finger* BSTs to be those which start each search at the destination of the previous search and move to the item being searched. In general, this movement involves going up to the least common ancestor (LCA) of the previous and current items being searched, and then moving down from the LCA to the current item being searched. In order to facilitate such a search, each node of the tree needs to be augmented with the minimal and maximal elements in its subtree. (In fact, this can be reduced by observing that if one is willing to look at the data in the parent of the LCA, only the minimum or maximum needs to be stored depending on whether a node is the left or right child of its parent).

1.2 Notation and Definitions

A static tree T is a fixed binary search tree containing n elements. No rotations are allowed. The data structure must process a sequence of searches, by moving a single pointer in the tree. Let $r(T, i, j)$ be the time to move the pointer in the tree T from node i to j. If $d_T(i)$ represents the depth of node i, with the root defined as having depth zero, then

$$\begin{aligned} r(T, i, j) &= d_T(i) - d_T(\mathrm{LCA}_T(i, j)) + d_T(j) - d_T(\mathrm{LCA}_T(i, j)) \\ &= d_T(i) + d_T(j) - 2d_T(\mathrm{LCA}_T(i, j)). \end{aligned}$$

Observe that if i is the root, $r(T, i, j)$ is simply $d_T(j)$, as the other two terms are the depth of the root, which is zero.

The runtime to execute a sequence $X = x_1, x_2, \ldots x_m$ of searches on a tree T using the root finger method is

$$R_{\text{root}}(T, X) = \sum_{i=1}^{m} r(T, root(T), x_i) = \sum_{i=1}^{m} d_T(x_i)$$

and the runtime to execute the same sequence on a tree T using the lazy finger method is

$$\begin{aligned} R_{\text{lazy}}(T, X) &= \sum_{i=1}^{m} r(T, x_{i-1}, x_i) \\ &= \left(2 \sum_{i=1}^{m} (d_T(x_i) - d_T(\text{LCA}_T(x_i, x_{i-1})))\right) - d_T(x_m) \end{aligned}$$

where x_0 is defined to be the root of T, which is where the first search starts.

1.3 History of Optimal Static Trees with Root Finger

For the root finger method, once the tree T is fixed, the cost of any single search in tree T depends only on the search and the tree, not on any of the search history. Thus, the optimal search tree for the root finger method is a function only of the frequency of the searches for each item. Let $f_X(a)$ denote the number of searches in X to a. Given f_X, computing the optimal static BST with root finger has a long history. In 1971, Knuth gave a $O(n^2)$ dynamic programming solution that finds the optimum tree [15]. More interesting is the discovery of a connection between the runtime of the optimal tree and the entropy of the frequencies:

$$H(f_X) = \sum_{a=1}^{n} \frac{f_X(a)}{m} \lg \frac{m}{f_X(a)}.$$

Melhorn [16] showed that a simple greedy heuristic proposed by Knuth [15] and shown to have a linear-time implementation by Fredman [11] produced a static tree where an average search takes time $2 + \frac{1}{1-\lg(\sqrt{5}-1)} H(f_X)$. Furthermore, Melhorn demonstrated a lower bound of $\frac{1}{\lg 3} H(f_X)$ for an average search in an optimal static tree, and showed this bound was tight for infinitely many distributions. (The lg 3 come from the fact that the comparisons at the node of a BST are in fact three-way and thus yield at most lg 3 bits of information). Thus, by 1975, it was established that the runtime for an average search in an optimal search tree with root finger was $\Theta(H(f_X))$, and that such a tree could easily be computed in linear time.

1.4 Our Results

We wish to study the natural problem of what we call search with a lazy finger in a static tree, i.e. have each search start where the last one ended. We seek to characterize the optimal tree for this search strategy, and describe how to build it.

The lazy finger method is asymptotically clearly no worse then the root finger method; moving up to the LCA and back down is better than moving to the root and back down, which is exactly double the cost of the root finger method. But, in general, is the lazy finger method better? For the lazy finger method, the cost of a single search in a static tree depends only on the current search and the previous search. Thus the optimal search tree for the lazy finger method only depends on the frequency of each search transition; let $f_X(a, b)$ be the number of searches in X to b where the previous search was to a. This is between the power of root finger, which only depends on the current search, and trees with rotations where the runtime can depend on a superconstant number of the previous searches. Given these pairwise frequencies (from which the frequencies $f_X(a)$ can easily be computed), is there a nice closed form for the runtime of the optimal BST with lazy finger? One candidate runtime to consider is the conditional entropy, which takes into account the entropy of one event given another:

$$H_c(f_X) = \sum_{a=1}^{n} \sum_{b=1}^{n} \frac{f_X(a, b)}{m} \lg \frac{f_X(a)}{f_X(a, b)}$$

This is of interest as information theory gives this as an expected lower bound[1] if the search sequence is derived from a Markov chain where n states represents searching each item; this is because the conditional entropy gives the expected information (and thus a lower bound on binary decisions) in a search given the previous search.

While a runtime related to the conditional entropy is the best achievable by any algorithm parameterized solely on the pairwise frequencies, however, we will show in Lemma 5 that the conditional entropy is impossible to be asymptotically achieved for any BST, static or dynamic, within any $o(\log n)$ factor. Thus, for the root finger, the lower bound given by information theory is achievable, yet for lazy finger it is not related to the runtime of the optimal tree. In Sect. 7 we will present a simple static non-tree structure whose runtime is related to the conditional entropy.

This still leaves us with the question: is there a simple closed form for the runtime of the optimal BST with lazy finger? We answer this in the affirmative by showing an equivalence between the runtime of BSTs with lazy finger and something known as the *weighted dynamic finger runtime*. In the weighted dynamic finger runtime, if item i is given weight w_i, then the time to execute search x_i is

$$\lg \frac{\sum_{k=\min(x_i, x_{i-1})}^{\max(x_i, x_{i-1})} w_k}{\min(w_{x_{i-1}}, w_{x_i})}.$$

Our main theorem is that the runtime of the best static tree with lazy finger, $LF(X)$, is given by the weighted dynamic finger runtime bound with the best choice of weights:

[1] When multiplied by $\frac{1}{\lg 3}$, as the information theory lower bound holds for binary decisions and as observed in [16] needs to be adjusted to the ternary decisions that occur at each node when traversing a BST.

$$LF(X) = \min_T R_{\text{lazy}}(T, X) = \Theta\left(\min_W \left\{ \sum_{i=1}^{m} \lg \frac{\sum_{k=\min(x_i, x_{i-1})}^{\max(x_i, x_{i-1})} w_k}{\min(w_{x_{i-1}}, w_{x_i})} \right\}\right)$$

To prove this, we first state the result of Seidel and Aragon [17] in Sect. 2 of how to construct a tree with the weighted dynamic finger runtime given a set of weights. Then, in Sect. 3, we show how, given any static tree T, there exist weights such that the runtime of T on a sequence using a lazy finger can be lower bounded using the weighted dynamic finger runtime with these weights. These results are combined in Sect. 4 to give the main theorem.

While a nice closed-form formula for the runtime of splay trees is not known, there are several different bounds on their runtime: working set, static finger, dynamic finger, and static optimality [7,8,18]. One implication of our result is that the runtime of the optimal lazy finger tree is asymptotically as good as that of all of the aforementioned bounds with the exception of the working set bound (see Theorem 3 for why the working set bound does not hold on a lazy finger static structure).

While these results have served to characterize the best runtime for the optimal BST, a concrete method is needed to compute the best tree given the pairwise frequencies. We present a dynamic programming solution in Sect. 6; this solution takes time $O(n^3)$ to compute the optimal tree for lazy finger, given a table of size n^2 with the frequency of each pair of searches occurring adjacently. This method could be extended using the ideas of Iacono and Mulzer [14] into one which periodically rebuilds the static structure using the observed frequencies so far; the result would be an online structure that for sufficiently long search sequences achieves a runtime that is within a constant factor of the optimal tree without needing to be initialized with the pairwise frequencies.

1.5 Relation to Finger Search Structures

The results here have a relation to the various finger search structures that have been proposed. We note, first of all, that the trees we are considering are not level linked; the only pointers are to the parent and children. Secondly, while the basic finger search runtime of $O(\sum_{i=2}^{m} \log |x_i - x_{i-1}|)$ (recall that we are assuming the x_i are integers from 1 to n) is long known to be easily achievable in a static tree, it is easily shown that there are some search sequences X for which the optimal tree performs far better. For example, the search sequence $x_i = i\sqrt{n} \bmod n$ where n is a perfect square can be easily executed in time $O(m)$ on the best static tree with lazy finger, which is much better than the $\Theta(m \log n)$ of dynamic finger.

But this limitation of the $O(\sum_{i=2}^{m} \log |x_i - x_{i-1}|)$ runtime has been long known, which is why the weighted version of finger search was proposed. Our main contribution is to realize that the weighted dynamic finger runtime bound, which was not proposed in the context of lazy finger, is the asymptotically tight characterization of BSTs with lazy finger when used with the best choice of weights.

1.6 Why Static Trees?

Static trees are less powerful than dynamic ones in terms of the classes of search sequence distributions that can be executed quickly, so why are we studying them?

One should use the simplest structure with the least overhead that gets the job done. By completely categorizing the runtime of the optimal tree with lazy finger, one can know if such a structure is appropriate for a particular application or whether one should instead use the more powerful dynamic trees, or simpler root-finger trees.

Rotation-based trees have horrible cache performance. However, there are methods to map the nodes of a static tree to memory so as to have optimal performance in the disk-access model and cache-oblivious models of the memory hierarchy [6,9,12,19]. One leading cache oblivious predecessor query data structure that supports insertion and deletion works by having a static tree and moves the data around in the fixed static tree in response to insertions and deletions and only periodically rebuilds the static structure [4]—in such a structure an efficient static structure is the key to obtaining good performance even with insertions and deletions.

Also, concurrency becomes a real issue in dynamic trees, which requires another layer of complexity to resolve (see, for example Bronson et al. [5]), while static trees trivially support concurrent operations.

2 Weights Give a Tree

We use the following theorem:

Theorem 1 (Seidel and Aragon [3]) *Given a set of positive weights* $W = w_1, w_2, \ldots w_n$, *there is a randomized method to choose a tree* T_W *such that the expected runtime (with repect to the choice of tree) of search with lazy finger is* $r(T_W, i, j) = O\left(\lg \frac{\sum_{k=\min(i,j)}^{\max(i,j)} w_k}{\min(w_i, w_j)}\right)$.

The method to randomly create T_W is a straightforward random tree construction using the weights: recursively pick the root using the normalized weights of all nodes as probabilities. Thus, by the probabilistic method [2], there is a deterministic tree, call it T_W whose runtime over the sequence X is at most the runtime bound of Seidel and Aragon for the sequence X on the best possible choice of weights.

Corollary 1 *For any set of positive weights* $W = w_1, w_2, \ldots w_n$ *there is a tree* $T_W(X)$ *such that*

$$\sum_{i=1}^{m} r(T_W(X), x_{i-1}, x_i) = O\left(\sum_{i=1}^{m} \lg \frac{\sum_{k=\min(x_i, x_{i-1})}^{\max(x_i, x_{i-1})} w_k}{\min(w_{x_{i-1}}, w_{x_i})}\right)$$

Proof This follows directly from Seidel and Aragon, where $T_W(X)$ is a tree that achieves the expected runtime of their randomized method for the best choice of weights. □

3 Trees Can Be Represented by Weights

We show that a tree can be represented by weights:

Lemma 1 *For each tree T there is a set of weights $W^T = w_1^T, w_2^T, \ldots w_n^T$ such that for all i, j $r(T, i, j) = \Theta\left(\lg \frac{\sum_{k=\min(i,j)}^{\max(i,j)} w_k^T}{\min(w_i^T, w_j^T)}\right)$.*

Proof These weights are simple: give a node at depth d in T a weight of 4^{-d}. Consider a search that starts at node i and goes to node j. Such a path goes up from i to $\mathrm{LCA}_T(i, j)$ and down to j. A lower bound on $\sum_{k=\min(i,j)}^{\max(i,j)} w_k^T$ is the weight of $\mathrm{LCA}_T(i, j)$ which is included in this sum and is $4^{-d_T(\mathrm{LCA}_T(i,j))}$. Thus we can bound $\lg \frac{\sum_{k=\min(i,j)}^{\max(i,j)} w_k^T}{\min(w_i^T, w_j^T)}$ as follows: $\lg \frac{\sum_{k=\min(i,j)}^{\max(i,j)} w_k^T}{\min(w_i^T, w_j^T)} \geq \lg \frac{4^{-d_T(\mathrm{LCA}_T(i,j))}}{\min(4^{-d_T(i)}, 4^{-d_T(j)})} = 2\max(d_T(i), d_T(j)) - 2d_T(\mathrm{LCA}_T(i, j)) \geq d_T(i) + d_T(j) - 2d_T(\mathrm{LCA}_T(i, j)) = r(T, i, j)$.

Similarly, an upper bound on $\sum_{k=\min(i,j)}^{\max(i,j)} w_k^T$ is twice the weight of $\mathrm{LCA}_T(i, j)$: $2 \cdot 4^{-d_T(\mathrm{LCA}_T(i,j))}$. This is because each of the two paths down from the LCA have weights that when summed are geometric and sum to less than half that of the LCA: $\lg \frac{\sum_{k=\min(i,j)}^{\max(i,j)} w_k^T}{\min(w_i^T, w_j^T)} \leq \lg \frac{2 \cdot 4^{-d_T(\mathrm{LCA}_T(i,j))}}{\min(4^{-d_T(i)}, 4^{-d_T(j)})} = 1 + \lg \frac{4^{-d_T(\mathrm{LCA}_T(i,j))}}{\min(4^{-d_T(i)}, 4^{-d_T(j)})}$ which is $1 + 2r(T, i, j)$ using the same math as in the previous paragraph. □

4 Proof of Main Theorem

Here we combine the results of the previous two sections to show that the runtime of the optimal tree with lazy finger is asymptotically the weighted dynamic finger bound for the best choice of weights.

Theorem 2

$$\min_T \left\{\sum_{i=1}^m r(T, x_{i-1}, x_i)\right\} = \Theta\left(\min_W \left\{\sum_{i=1}^m \lg \frac{\sum_{k=\min(x_i, x_{i-1})}^{\max(x_i, x_{i-1})} w_k}{\min(w_{x_{i-1}}, w_{x_i})}\right\}\right)$$

Proof Start by setting $T^{\min}$, to be the optimal tree. That is, $T^{\min} = \mathrm{argmin}_T \left\{\sum_{i=1}^m r(T, x_{i-1}, x_i)\right\}$:

$$\min_T \left\{\sum_{i=1}^m r(T, x_{i-1}, x_i)\right\} = \sum_{i=1}^m r(T^{\min}, x_{i-1}, x_i)$$

Using Lemma 1 there is a constant c such and a set of weights $w^{T^{\min}}$ such that:

$$\geq c \sum_{i=1}^m \lg \frac{\sum_{k=\min(x_i, x_{i-1})}^{\max(x_i, x_{i-1})} w_k^{T^{\min}}}{\min(w_{x_{i-1}}^{T^{\min}}, w_{x_i}^{T^{\min}})}$$

The weights $w^{T^{\min}}$ are a lower bound on the sum with the optimal weights

$$\geq c \min_W \sum_{i=1}^m \lg \frac{\sum_{k=\min(x_i, x_{i-1})}^{\max(x_i, x_{i-1})} w_k}{\min(w_{x_{i-1}}, w_{x_i})}$$

Using Theorem 1, there is a constant c' such that:

$$\geq c' \sum_{i=1}^{m} r(T_W, x_{i-1}, x_i)$$

The sum with T_w is at most the sum for optimal T:

$$\geq c' \min_T \left\{ \sum_{i=1}^{m} r(T, x_{i-1}, x_i) \right\}$$

□

5 Hierarchy and Limitations of Models

In this section we show there is a strict hierarchy of runtimes from the root finger static BST model to the lazy finger static BST model to the rotation-based BST model. Let $OPT(X)$ be the fastest any binary search with rotations can execute X.

Theorem 3 *For any sequence* X, $\min_T R_{root}(T, X) = \Omega\left(\min_T R_{lazy}(T, X)\right) = \Omega(OPT(X))$. *Furthermore there exist classes of search sequences of any length* m, X'_m *and* X''_m *such that* $\min_T R_{root}(T, X'_m) = \omega\left(\min_T R_{lazy}(T, X'_m)\right)$ *and* $\min_T R_{lazy}(T, X''_m) = \omega(OPT(X''_m))$.

Proof We address each of the claims of this theorem separately.

Root finger can be simulated with lazy finger $\min_T R_{\text{root}}(T, X) = \Omega(\min_T R_{\text{lazy}}(T, X))$. For lazy finger, moving up to the LCA and back down is no more work than than moving to the root and back down, which is exactly the double of the cost of the root finger method.

Lazy finger can be simulated with a rotation-based tree $\min_T R_{\text{lazy}}(T, X) = \Omega(OPT(X))$. The normal definition of a tree allowing rotations has a finger that starts at the root at every operation and can move around the tree performing rotations, where following pointers and performing rotations can be done at unit cost. The work of Demaine et al. [10] shows how to simulate with constant-factor overhead any number of lazy fingers in a tree that allows rotations in the normal tree with rotations and one single pointer that starts at the root. This transformation can be used on a static tree with lazy finger to get the result.

Some sequences can be executed quickly with lazy finger but not with root finger: There is a X'_m *such that* $\min_T R_{\text{root}}(T, X'_m) = \omega\left(\min_T R_{\text{lazy}}(T, X'_m)\right)$. One choice of X'_m is the sequential search sequence $1, 2, \ldots n, 1, 2, \ldots$ repeated until a search sequence of length m is created. So long as $m \geq n$, this takes time $O(m)$ to execute on any tree using lazy finger, but takes $\Omega(m \lg n)$ time to execute on every tree using root finger.

Some sequences can be executed quickly using a BST with rotations, but not with lazy finger Pick some small k, say $k = \lg n$. Create the sequence X''_m in rounds as follows: In each round pick k random elements from $1, \ldots, n$, search each of them once, and then perform n random searches on these k elements. Continue with more rounds until a total of m searches are performed. A splay tree can perform this in time $O(m \lg k)$. This is because splay trees have the working-set bound, which states that

the amortized time to search an item is at most big-O of the logarithm of the number of different things searched since the last time that item was searched. For the sequence X''_m, the n random searches in each round have been constructed to have a working set bound of $O(\lg k)$ amortized, while the k other searches in each round have a working set bound of $O(\lg n)$ amortized. Thus the total cost to execute X''_m on a splay tree is $O\left(\frac{m}{n+k}(n \lg k + k \lg n)\right)$ which is $O(m \lg \lg n)$ since $k = \lg n$.

However, for a static tree with lazy finger, X''_m is basically indistinguishable from a random sequence and takes $\Omega(m \lg n)$ expected time. This is because the majority of the searches are random searches where the previous item was a random search, and in any static tree the expected distance between two random items is $\Omega(\lg n)$. □

Lemma 2 *The runtime of a BST in any model cannot be related to the conditional entropy of the search sequence.*

Proof Wilber [20] proved that there is a particular sequence, known as the bit reversal sequence, that if one searches the items in the sequence it takes $\Omega(n \lg n)$ time in an optimal dynamic BST. This sequence is a precise permutation of all elements in the tree. However, any single permutation repeated over and over has a conditional entropy of 0, since every search is completely determined by the previous one. □

6 Constructing the Optimal Lazy Finger BST

Recall that $f_{a,b} = f_X(a, b)$ is the number of searches in X where the current search is to b and the previous search is to a, and $f_X(a)$ is the number of searches to a in X. We will first describe one method to compute the cost to execute X on some tree T. Suppose the nodes in $[a, b]$ constitute the nodes of some subtree of T, call it $T_{a,b}$ and denote the root of the subtree as $r(T_{a,b})$. We now present a recursive formula for computing the expected cost of a single search in T. Let $R_{\text{lazy}}(T, X, a, b)$ be the number of edges traversed in $T_{a,b}$ when executing X. Thus, $R_{\text{lazy}}(T, X, 1, n)$ equals the runtime $R_{\text{lazy}}(T, X)$. There is a recursive formula for $R_{\text{lazy}}(T, X, a, b)$:

$$R_{\text{lazy}}(T, X, a, b) = \begin{cases} 0 \text{ if } b < a \\ \overbrace{R_{\text{lazy}}(T, X, a, r(T_{a,b}) - 1)}^{(a)} \\ \quad + \overbrace{R_{\text{lazy}}(T, X, r(T_{a,b}) + 1, b)}^{(b)} \\ \quad + \overbrace{2 \sum_{\substack{i \in [a, r(T_{a,b})-1] \\ j \in [r(T_{a,b})+1, b]}} (f_{i,j} + f_{j,i})}^{(c)} \\ \quad + \overbrace{\sum_{i \neq r(T_{a,b})} (f_{i,r(T_{a,b})} + f_{r(T_{a,b}),i})}^{(d)} \\ \quad + \overbrace{\sum_{\substack{i \in [a,b] \\ i \neq r(T_{a,b}) \\ j \notin [a,b]}} (f_{i,j} + f_{j,i})}^{(e)} \end{cases} \text{otherwise}$$

The formula is long but straightforward. First we recursively include the number of edges traversed in the left (a) and right (b) subtrees of the root $r(T_{a,b})$. Thus, all that is left to account for is traversing the edges between the root of the subtree and its up to two children. Both edges to its children are traversed when a search moves from the left to right subtree of $r_{a,b}$ or vice-versa (c). A single edge to a child of the $r(T_{a,b})$ traversed if a search moves from either the left or right subtrees of $r(T_{a,b})$ to $r(T_{a,b})$ itself or vice-versa (d), or if a search moves from any node but the root in the current subtree containing the nodes $[a, b]$ out to the rest of T or vice-versa (e).

This formula can easily be adjusted into one to determine the optimal cost over all trees—since at each step the only dependence on the tree was is root of the current subtree, the minimum can be obtained by trying all possible roots. Here is the resultant recursive formulation for the minimum number of edges traversed in and among all subtrees containing $[a, b]$:

$$\min_T R_{\text{lazy}}(T, X, a, b) = \begin{cases} 0 \text{ if } b < a \\ \min_{r\in[a,b]} \left\{ \begin{array}{l} \min_T R_{\text{lazy}}(T, X, a, r-1) \\ +\min_T R_{\text{lazy}}(T, X, r+1, b) \\ +2\sum_{\substack{i\in[a,r-1]\\ j\in[r+1,b]}}(f_{i,j}+f_{j,i}) \\ +\sum_{i\neq r}(f_{i,r}+f_{r,i}) \\ +\sum_{\substack{i\in[a,b]\\ i\neq r(T_{a,b})\\ j\notin[a,b]}}(f_{i,j}+f_{j,i}) \end{array} \right\} \text{otherwise} \end{cases}$$

This formula can trivially be evaluated using dynamic programming in $O(n^5)$ time as there are $O(n^3)$ choices for a, b, and r and evaluating the summations in the brute-force way takes time $O(n^2)$. The dynamic programming gives not only the cost of the best tree, but the minimum roots chosen at each step gives the tree itself. The runtime can be improved to $O(n^3)$ by observing that when f is viewed as a 2-D array, each of the sums is simply a constant number of partial sum queries (queries that ask for the sum of a contiguous block in the array) on the array f, each of which can be answered in $O(1)$ time after $O(n^2)$ preprocessing. (The folklore method of doing this is to store all the 2-D partial sums from the origin; a generic partial sum can be computed from these with a constant number of additions and subtractions).

We summarize this result in the following theorem:

Theorem 4 *Given the pairwise frequencies f_X finding the tree that minimizes the execution time of search sequence X using lazy finger takes time $O(n^3)$.*

This algorithm computes an optimal tree. Computing f from X can be done in $O(m)$ time, for a total runtime of $O(m + n^3)$. It remains open if there is any approach to speed up the computation of the optimal tree, or an approximation thereof. Note that although our closed form expression of the asymptotic runtime of the best tree was stated in terms of an optimal choice of weights, the dynamic program presented here in no way attempts to compute these weights. It would be interesting if some weight-based method were to be discovered.

7 Multiple Trees Structure

Here we present a static data structure in the comparison model on a pointer machine that guarantees an average search time of $O(H_c(f_X)\log_d n)$ for any fixed value $1 \leq d \leq n$, a runtime which we have shown to be impossible for any BST algorithm, static or dynamic. This data structure requires $O(dn)$ space. In particular, setting $d = n^\epsilon$ gives a search time of $O(H_c(f_X))$ with space $O(n^{1+\epsilon})$ for any $\epsilon > 0$. The purpose of this structure is to demonstrate that while no tree can have a runtime related to the conditional entropy, pointer based structures can.

As a first attempt, a structure could be made of n binary search trees $T_1, T_2, \ldots T_n$ where each tree T_i is an optimal static tree given the previous search was to i. By using tree $T_{x_{i-1}}$ to execute search T_i, the asymptotic conditional entropy can be easily obtained. However the space of this structure is $O(n^2)$. Thus space can be reduced by observing the nodes not near the root of every tree are being executed slowly and thus need not be stored in every tree.

The *multiple trees structure* has two main parts. It is composed first by a complete binary search tree T' containing all of set of keys to be stored, $S = [1, \ldots, n]$. Thus the height of T' is $O(\lg n)$. The second part is n binary search trees $\{T_1, T_2, \ldots, T_n\}$. A tree T_i contains the d elements j that have the greatest frequencies $f_X(i, j)$; these are the j elements most frequently searched after that i has been searched. The depth of an element j in T_i is $O(\lg \frac{f_X(i)}{f_X(i,j)})$. For each element j in the entire structure we add a pointer linking j to the root of T_j. The tree T' uses $O(n)$ space and every tree T_j uses $O(d)$ space. Thus the space used by the entire structure is $O(dn)$.

Suppose we have just searched the element i and our finger search is located on the root of T_i. Now we proceed to the next search to the element j in the following way: Search j in T_i. If j is in T_i then we are done, otherwise search j in T'. After we found j either in T_j or T' we move the finger to the root of T_j by following the aforementioned pointer.

If j is in T_i then it is found in time $O(\lg \frac{f_X(i)}{f_X(i,j)})$. Otherwise if j is found in T', then it is found in $O(\lg n)$ time. We know that if j is not in T_x this means that optimally it requires $\Omega(\lg d)$ comparisons to be found since T_x contains the d elements that have the greatest probability to be searched after that x has been accessed. Hence every search is at most $O(\lg n/\lg d)$ times the optimal search time of $O(\lg \frac{f_X(i)}{f_X(i,j)})$. Thus a search for x_i in X takes time $O\left(\log_d n \lg \frac{f_X(x_i)}{f_X(x_{i-1},x_i)}\right)$. Summing this up over all m searches x_i in X gives the runtime to execute X:

$$\begin{aligned} O\left(\sum_{i=1}^{m} \log_d n \lg \frac{f_X(x_i)}{f_X(x_{i-1},x_i)}\right) &= O\left(\sum_{a=1}^{n}\sum_{b=1}^{n} f_X(a,b)\log_d n \lg \frac{f_X(a)}{f_X(x_a,x_b)}\right) \\ &= O(mH_c(f_X)\log_d n) \end{aligned}$$

We summarize this result in the following theorem:

Theorem 5 *Given the pairwise frequencies f_X and a constant d, $1 \leq d \leq n$, the multiple trees structure executes X in time $O(mH_c(f_X)\log_d n)$ and uses space $O(nd)$.*

8 Open Problems

We conjecture that no pointer-model structure has space $O(n)$ and search cost $O(H_c(f_X))$.

One interesting implication of this result is that any search tree which is dynamically optimal must have the weighted dynamic finger runtime for any choice of weights including the minimizing one. However, no rotation-based binary search tree algorithms are known which have this runtime bound. The closest thing that is known is the dynamic finger property of spay trees, which is equivalent to the weighted dynamic finger with unit weights [7,8]. However, the proof found in [7,8] does not lend itself to an obvious extension to non-unit weights.

References

1. Adelson-Velskij, G.M., Landis, E.M.: An algorithm for the organization of information. Doklady Akademii Nauk USSR **146**(2), 263–266 (1962)
2. Alon, N., Spencer, J.: The Probabilistic Method. Wiley, London (1992)
3. Aragon, C.R., Seidel, R.: Randomized search trees. In: FOCS, pp. 540–545. IEEE Computer Society (1989)
4. Bender, M.A., Duan, Z., Iacono, J., Jing, W.: A locality-preserving cache-oblivious dynamic dictionary. J. Algorithms **53**(2), 115–136 (2004)
5. Bronson, N.G., Casper, J., Chafi, H., Olukotun, K.: A practical concurrent binary search tree. In: Govindarajan, R., Padua, D.A., Hall, M.W. (eds.) PPOPP, pp. 257–268. ACM (2010)
6. Clark, D.R., Munro, J.I.: Efficient suffix trees on secondary storage (extended abstract). In: Tardos, E. (ed.) SODA, pp. 383–391. ACM/SIAM (1996)
7. Cole, R.: On the dynamic finger conjecture for splay trees. part ii: the proof. SIAM J. Comput. **30**(1), 44–85 (2000)
8. Cole, R., Mishra, B., Schmidt, J.P., Siegel, A.: On the dynamic finger conjecture for splay trees. part i: splay sorting log n-block sequences. SIAM J. Comput. **30**(1), 1–43 (2000)
9. Demaine, E.D., Iacono, J., Langerman, S.: Worst-case optimal tree layout in a memory hierarchy. CoRR, cs.DS/0410048 (2004)
10. Demaine, E.D., Iacono, J., Langerman, S., Özkan, O.: Combining binary search trees. In: Proceedings of the 40th International Colloquium on Automata, Languages and Programming (ICALP 2013), Riga, Latvia, 8–12 July 2013, pp. 388–399 (2013)
11. Fredman, M.L.: Two applications of a probabilistic search technique: sorting x + y and building balanced search trees. In: Rounds, W.C., Martin, N., Carlyle, J.W., Harrison, M.A. (eds.) STOC, pp. 240–244. ACM (1975)
12. Gil, J., Itai, A.: How to pack trees. J. Algorithms **32**(2), 108–132 (1999)
13. Guibas, L.J., Sedgewick, R.: A dichromatic framework for balanced trees. In: FOCS, pp. 8–21. IEEE Computer Society (1978)
14. Iacono, J., Mulzer, W.: A static optimality transformation with applications to planar point location. Int. J. Comput. Geom. Appl. **22**(4), 327–340 (2012)
15. Knuth, D.E.: Optimum binary search trees. Acta Inf. **1**, 14–25 (1971)
16. Mehlhorn, K.: Nearly optimal binary search trees. Acta Inf. **5**, 287–295 (1975)
17. Seidel, R., Aragon, C.R.: Randomized search trees. Algorithmica **16**(4/5), 464–497 (1996)
18. Sleator, D.D., Tarjan, R.E.: Self-adjusting binary search trees. J. ACM **32**(3), 652–686 (1985)
19. Peter van Emde Boas: Preserving order in a forest in less than logarithmic time and linear space. Inf. Process. Lett. **6**(3), 80–82 (1977)
20. Wilber, R.E.: Lower bounds for accessing binary search trees with rotations. SIAM J. Comput. **18**(1), 56–67 (1989)