

Improved Bounds for Finger Search on a RAM *

Alexis Kaporis [†] Christos Makris [†] Spyros Sioutas [‡] Athanasios Tsakalidis [§]
Kostas Tsihclas [¶] Christos Zaroliagis [§]

July 11, 2009

Abstract

We present a new finger search tree with $O(1)$ worst-case update time and $O(\log \log d)$ expected search time in the Random Access Machine (RAM) model of computation for a large class of input distributions. The parameter d represents the number of elements (distance) between the search element and an element pointed to by a finger, in a finger search tree that stores n elements. Our data structure improves upon a previous result by Andersson and Mattsson that exhibits expected $O(\log \log n)$ time and $O(1)$ worst-case update time by incorporating the distance d into the search time complexity, and thus removing the dependence on n . We are also able to show that we can achieve an expected search time of $O(\log \log d + \phi(n))$ with high probability, where $\phi(n)$ is *any* slowly growing function of n . For the need of the analysis we model the updates by a “balls and bins” combinatorial game that is interesting in its own right as it involves insertions and deletions of balls according to an unknown distribution.

Keywords: data structures, dictionary problem, balls and bins problem, interpolation search, expected analysis.

1 Introduction

Search trees and in particular finger search trees are fundamental data structures that have been extensively studied and used. Applications of finger search trees include optimal algorithms for the basic operations of union, intersection and difference on sets [24], efficient list splitting [24], efficient implementation of priority queues [16], efficient sorting of nearly ordered files [16] and sorting of Jordan sequences in linear time [19]. They also find applications in computational geometry, for example in constructing the visibility graph of a polygon [18, 15], in deriving optimal algorithms for

*This work was partially supported by the Future and Emerging Technologies Unit of EC (IST priority – 6th FP), under contracts no. IST-2002-001907 (integrated project DELIS) and no. FP6-021235-2 (project ARRIVAL), and by the Carathéodory project of the University of Patras.

[†]Department of Computer Engineering and Informatics, University of Patras, 26500 Patras, Greece. Email: {kaporis,makri}@ceid.upatras.gr

[‡]Department of Informatics, Ionian University, Corfu, Greece. Email: sioutas@ionio.gr

[§]Department of Computer Engineering and Informatics, University of Patras, 26500 Patras, Greece; and R.A. Computer Technology Institute, N. Kazantzaki Str, Patras University Campus, 26504 Patras, Greece. Email: {tsak,zaro}@ceid.upatras.gr

[¶]Corresponding Author. Department of Informatics, Aristotle University of Thessaloniki, Greece; and R.A. Computer Technology Institute, N. Kazantzaki Str, Patras University Campus, 26504 Patras, Greece. Email: tsichlas@csd.auth.gr

the 3-dimensional layers-of-maxima problem [4], and in obtaining improved methods for dynamic point location [4].

A *finger search tree* is a leaf-oriented search tree storing n elements, in which the search procedure can start from an arbitrary element pointed to by a *finger* f (for simplicity, we shall not distinguish throughout the paper between f and the element pointed to by f). The goal is: (i) to find another element x stored in the tree in a time complexity that is a function of the “distance” (number of leaves) d between f and x ; and (ii) to update the data structure after the deletion of f or after the insertion of a new element next to f .

Previous Work. Several results for finger search trees have been achieved on the Pointer Machine (PM) and the Random Access Machine (RAM) models of computation. The PM model [32] is assumed to have a memory consisting of an unbounded collection of locations (registers) connected by pointers. Each location may contain an arbitrary amount of additional information, and no arithmetic is allowed in order to compute the address of a location. The only way to access a location is by following pointers. The RAM model comes in two main variants. The first variant is the classic, comparison-based, unit-cost RAM [8], which is assumed to have a memory consisting of an infinite collection of addressable locations (with addresses $0, 1, 2, \dots$). Each location has the capacity of storing a number of arbitrary (theoretically infinite) precision. Comparisons are assumed to take constant time. Arithmetic operations are allowed for computing memory addresses. For an input of n elements, it is tacitly assumed that arithmetic and Boolean operations as well as operations for indexing an n -element array are carried out in constant time on $O(\log n)$ -bit integers. The second variant is the so-called word RAM [12, 17] and constitutes a variation of the classic unit-cost RAM. In particular, the memory is divided into addressable locations or words, each having a word length of w bits, and these addresses are themselves stored in memory words. For an input of size n , it should hold that $w \geq \log n$ (since otherwise n is not representable), and the memory locations store integers in the range $[0, 2^w - 1]$. The restriction to integers is not crucial. Real numbers of finite precision can also be handled [2, 3, 17, 33, 35, 36], as for example numbers following the IEEE 754 floating-point standard. It is also assumed that the word RAM can perform the standard AC^0 operations of addition, subtraction, comparison, bitwise Boolean operations and shifts, as well as multiplications in constant worst-case time on $O(w)$ -bit operands.

In the classic RAM model, finger search trees with $O(1)$ update time and $O(\log d)$ search time have already been devised by Dietz and Raman [10]. Recently, for the word RAM model, Andersson and Thorup [2, 3] presented a new data structure for finger search trees with $O(1)$ update time and $O(\sqrt{\log d / \log \log d})$ search time, which is optimal since there exists a matching lower bound for searching on a word RAM [5]. In the PM model, Brodal et al. [6] presented very recently a finger search tree with $O(1)$ update time and $O(\log d)$ search time, which is optimal for this model due to the lower bound on sorting [23]. All these bounds are worst-case time complexities and since they have matching lower bounds, there is no room for improvement.

However, simpler data structures and/or improvements regarding the search complexities on the classic RAM model can be obtained if randomization is allowed, or if certain classes of input distributions are considered. An example for the former is the simple and elegant finger search tree developed by Seidel and Aragon [30] that achieves $O(1)$ expected update time when decisions for rebalancing operations are guided by tosses of coins, while the search operation is carried out in $O(\log d)$ expected time. A notorious example for the latter is the method of *interpolation search*, first suggested by Peterson [29], which for random data generated according to the *uniform*

distribution achieves $\Theta(\log \log n)$ expected search time. This was shown in [14, 27, 37]. Willard in [34] showed that this time bound holds for an extended class of distributions, called *regular*¹.

A natural extension is to adapt interpolation search into dynamic data structures, that is, data structures which support insertion and deletion of elements in addition to interpolation search. Their study was started with the works of [11, 20] for insertions and deletions performed according to the uniform distribution, and continued by Mehlhorn and Tsakalidis [25], and Andersson and Mattsson [1] for μ -random insertions and random deletions, where μ is a so-called *smooth* density. An insertion is μ -random if the element to be inserted is drawn randomly with density function μ ; a deletion is random if every element present in the data structure is equally likely to be deleted (these notions of randomness are also described in [22]).

The notion of *smooth* input distributions that determine insertions of elements in the update sequence were introduced in [25], and were further generalized and refined in [1]. Informally, a distribution defined over an interval I is smooth if the probability density over any subinterval of I does not exceed a specific bound, however small this subinterval is (i.e., the distribution does not contain sharp peaks). More precisely, given two functions f_1 and f_2 , a density function $\mu = \mu[a, b](x)$ is (f_1, f_2) -smooth [1] if there exists a constant β , such that for all c_1, c_2, c_3 , $a \leq c_1 < c_2 < c_3 \leq b$, and all integers n , it holds that

$$\int_{c_2 - \frac{c_3 - c_1}{f_1(n)}}^{c_2} \mu[c_1, c_3](x) dx \leq \frac{\beta \cdot f_2(n)}{n}$$

where $\mu[c_1, c_3](x) = 0$ for $x < c_1$ or $x > c_3$, and $\mu[c_1, c_3](x) = \mu(x)/p$ for $c_1 \leq x \leq c_3$ where $p = \int_{c_1}^{c_3} \mu(x) dx$.

Intuitively, function f_1 partitions an arbitrary subinterval $[c_1, c_3] \subseteq [a, b]$ into f_1 equal parts, each of length $\frac{c_3 - c_1}{f_1} = O(\frac{1}{f_1})$; that is, f_1 measures how fine is the partitioning of an arbitrary subinterval. Function f_2 guarantees that no part, of the f_1 possible, gets more probability mass than $\frac{\beta f_2}{n}$; that is, f_2 measures the sparseness of any subinterval $[c_2 - \frac{c_3 - c_1}{f_1}, c_2] \subseteq [c_1, c_3]$. The class of (f_1, f_2) -smooth distributions (for appropriate choices of f_1 and f_2) is a superset of both regular and uniform classes of distributions, as well as of several non-uniform classes [1, 24]. Actually, any probability distribution is $(f_1, \Theta(n))$ -smooth, for a suitable choice of β .

In [25] a dynamic interpolation search data structure was introduced, called *Interpolation Search Tree (IST)*. This data structure requires $O(n)$ space for storing n elements. The amortized insertion and deletion cost is $O(\log n)$, while the expected amortized insertion and deletion cost is $O(\log \log n)$. The worst case search time is $O(\log^2 n)$, while the expected search time is $O(\log \log n)$ on sets generated by μ -random insertions and random deletions, where μ is a $(\lceil n^a \rceil, \sqrt{n})$ -smooth density function and $\frac{1}{2} \leq a < 1$. An IST is a multi-way tree, where the degree of a node u depends on the number of leaves of the subtree rooted at u (in the ideal case the degree of u is the square root of this number). Each node of the tree is associated with two arrays: a REP array which stores a set of sample elements, one element from each subtree, and an ID array that stores a set of sample elements approximating the inverse distribution function. In particular, for a node with degree \sqrt{m} and having m leaves in its subtree, the ID array divides the interval covered by the node in m^a subintervals, where $1/2 \leq a < 1$. Each interval is associated with a pointer to a proper subtree. The search algorithm for the IST uses the ID array in each visited node to interpolate REP and locate the element, and consequently the subtree where the search is to be continued.

¹A density μ is regular if there are constants b_1, b_2, b_3, b_4 such that $\mu(x) = 0$ for $x < b_1$ or $x > b_2$, and $\mu(x) \geq b_3 > 0$ and $|\mu'(x)| \leq b_4$ for $b_1 \leq x \leq b_2$.

In [1], Andersson and Mattsson explored further the idea of dynamic interpolation search by observing that: (i) the larger the ID array the bigger becomes the class of input distributions that can be efficiently handled with an IST-like construction; and (ii) the IST update algorithms may be simplified by the use of a static, implicit search tree whose leaves are associated with binary search trees and by applying the incremental global rebuilding technique of [26]. The resulting new data structure in [1] is called the *Augmented Sampled Forest (ASF)*. Assuming that $H(n)$ is a non-decreasing, invertible and $o(\log n)$ function (whose full details will be given in Section 2) denoting the height of the static implicit tree, Andersson and Mattsson [1] showed that an expected search and update time of $\Theta(H(n))$ can be achieved for μ -random insertions and random deletions where μ is $(n \cdot g(H(n)), H^{-1}(H(n)-1))$ -smooth and g is a function satisfying $\sum_{i=1}^{\infty} g(i) = \Theta(1)$. In particular, for $H(n) = \Theta(\log \log n)$ and $g(x) = x^{-(1+\varepsilon)}$ ($\varepsilon > 0$), they get $\Theta(\log \log n)$ expected search and update time for any $(n/(\log \log n)^{1+\varepsilon}, n^{1-\delta})$ -smooth density, where $\varepsilon > 0$ and $0 < \delta < 1$ (note that $(\lceil n^a \rceil, \sqrt{n})$ -smooth $\subset (n/(\log \log n)^{1+\varepsilon}, n^{1-\delta})$ -smooth). The worst-case search and update time is $O(\log n)$, while the worst-case update time can be reduced to $O(1)$ if the update position is given by a finger. Moreover, for several but more restricted than the above smooth densities they can achieve $o(\log \log n)$ expected search and update time complexities; in particular, for the uniform and any bounded distribution the expected search and update time becomes $O(1)$.

The above are the best results so far in both the realm of dynamic interpolation structures and the realm of dynamic search tree data structures for μ -random insertions and random deletions on the classic RAM model.

New Results. Based upon dynamic interpolation search, we present in this paper a new finger search tree which, for μ -random insertions and random deletions, achieves $O(1)$ worst-case update time and $O(\log \log d)$ expected search time on the classic RAM model of computation for the same class of smooth density functions μ considered in [1], thus improving upon the dynamic search structure of Andersson and Mattsson with respect to the expected search time complexity. We can also show that the expected search time becomes $O(\log \log d + \phi(n))$ *with high probability*², where $\phi(n)$ is any slowly growing function of n (e.g., the inverse Ackermann function [31]). Moreover, for the same classes of restricted smooth densities considered in [1], we can achieve $o(\log \log d)$ expected search and update time complexities (e.g., $O(1)$ times for the uniform and any bounded distribution). We would like to note that: (i) the expected bounds in [1, 25] have not been proved to hold with high probability; (ii) this is the first work (to the best of our knowledge) that uses the dynamic interpolation search paradigm in the framework of finger search trees.

Our data structure is based on a rather simple idea. It consists of two levels: the top level is a tree structure, called *static interpolation search tree* (cf. Section 2). The elements (unlike in [25]) are not stored in the leaves, but (similarly to [1]) in a family of buckets, which comprises the bottom level of our data structure. These buckets store a truncated version, up to a sufficiently large precision, of the real elements along with pointers to a sorted list containing the real elements. Buckets are treated as a kind of “indexing structure” to the real elements and are implemented using the q^* -heap machinery [12, 35, 36]. This can be seen as a small trick to accelerate the execution of the search and update operations. We also show that the mapping from fixed precision elements to the (arbitrary precision) real ones does not affect the efficiency of our operations.

Note that it is not at all obvious how a combination of the aforementioned top and bottom level data structures can give better bounds, since deletions of elements may create chains of

²Throughout the paper, we say that an event E occurs *with high probability* if $\Pr[E] = 1 - o(1)$.

empty buckets. To alleviate this problem and prove the expected search bound, we use an idea of independent interest. We model the insertions and deletions as a combinatorial game of bins and balls. This combinatorial game is innovative in the sense that it is not used in a load-balancing context, but it is used to model the behaviour of a dynamic data structure as the one we describe in this paper. We provide upper and lower bounds on the number of elements in a bucket and show that, with high probability, a bucket never gets empty. This fact implies that with high probability there cannot exist chains of empty buckets, which in turn allows us to express the search time bound in terms of the parameter d . Note that the combinatorial game presented here is different from the known approaches for balls and bins games (see e.g., [7]), since in those approaches the bins are considered static and the distribution of balls uniform. On the contrary, the bins in our game are random variables since the distribution of balls is unknown. This also makes the initialization of the game a non-trivial task which is tackled by firstly sampling a number of balls and then determining appropriate bins which allow the almost uniform distribution of balls into them.

Our data structure is designed for the classic (unit-cost) RAM. This is a direct consequence of modeling the elements of the structure as being generated by a continuous distribution, a characteristic common to *all* previous results on interpolation search [1, 13, 14, 25, 27, 28, 29, 34, 37]. Note that we do not use the arbitrary precision for any hidden costly calculation, it is just an artifact of the modeling of the source of the elements and the preservation of their nice statistical properties. Except for comparisons, we assume that the operation of truncating a real number of arbitrary precision to a number of an appropriately large (but fixed) precision takes constant time, and that fixed precision numbers are stored in memory words of $O(\log n)$ bits. We also assume that multiplication and the standard AC^0 operations (addition, subtraction, comparison, bitwise Boolean operations and shifts), required for the manipulation of elements within buckets, can be performed in constant worst-case time on $O(\log n)$ -bit operands.

The remainder of the paper is organized as follows. In Section 2, we discuss preliminary notions and results that are used throughout the paper, and define the static interpolation search tree. Our data structure is presented in Section 3, while the analysis of the time complexities of its operations is discussed in Section 4. The analysis of the combinatorial game, upon which our expected search time is based, is given in Section 5. We conclude in Section 6. A preliminary version of this work appeared in [21].

2 Preliminaries

The searching problem is fundamental in data structures. For our purposes, it is defined as follows. Consider a random file $F = \{X_1, \dots, X_n\}$, where each element $X_i \in [a, b] \subset \mathbb{R}$, $1 \leq i \leq n$, obeys an unknown distribution μ , and let $S = \{X_{(1)}, \dots, X_{(n)}\}$ be an increasing ordering of F . The goal is to find the largest element $X_{(j)} \in S$ that precedes (i.e., is less than or equal to) a *target* element x , starting the searching procedure from the entry point of the data structure representing S (e.g., if S is represented by a tree, then its entry point is the root of the tree). We call this the *classical searching* problem. In this paper, we will mainly deal with the *finger search* variation of the classical searching problem, where the searching procedure does not necessarily start from the entry point of the data structure, but from an arbitrary element pointed to by a finger f .

One crucial component of our design is a search tree data structure, which we call *static interpolation search tree*. The *static interpolation search tree* (SIST) is a static, explicit, and refined version of the search trees used in [1, 25] that both address the classical searching problem. Follow-

ing [1, 25], a static interpolation search tree corresponding to the ordered file S of n elements is fully characterized by three functions $H(n) : \mathcal{N} \rightarrow \mathbb{R}_0^+$, $R(n) : \mathcal{N} \rightarrow \mathbb{R}_0^+$ and $I(n) : \mathcal{N} \rightarrow \mathbb{R}_0^+$, which are non-decreasing and invertible with a second derivative less than or equal to zero. $H(n)$ denotes the height of the tree³, $R(n)$ denotes the out-degree of the root, and $I(n)$ denotes how fine is the partition of the set of elements. Achieving a height of $H(n)$ dictates that $R(n) = n/H^{-1}(H(n) - 1)$. Moreover, $H(n)$ should be $o(\log n)$ and not $O(1)$, and $H^{-1}(i) \neq 0$, for $1 \leq i \leq H(n) - 1$. In order to handle an as large as possible class of distributions μ , the approximation of the sample density should be as fine as possible, implying that $I(n)$ should be as large as possible. Since $I(n)$ affects space, it is chosen as $I(n) = n \cdot g(H(n))$, where $\sum_{i=1}^{\infty} g(i) = \Theta(1)$, so that the space of SIST remains linear. The following lemma characterizes the time and space complexity of SIST.

Lemma 1 *A Static Interpolation Search Tree on an ordered file of n elements drawn from a $(n \cdot g(H(n)), H^{-1}(H(n) - 1))$ -smooth distribution can be built in $O(n)$ time and uses $O(n)$ space.*

Proof. The proof is similar to the proof of [1, Theorem 6] and we provide it here for completeness.

Let $C(n)$ be the time to build the SIST for n elements. The time needed to build the ID and REP arrays is linear to their sizes. Thus the following recurrence relation for the build time of the SIST holds:

$$C(n) = I(n) + R(n) + R(n)C\left(\frac{n}{R(n)}\right)$$

Let $C(n) = nP(n)$. Then,

$$nP(n) = I(n) + R(n) + R(n)\frac{n}{R(n)}P\left(\frac{n}{R(n)}\right)$$

or

$$P(n) = \frac{I(n)}{n} + \frac{R(n)}{n} + P\left(\frac{n}{R(n)}\right) \quad (1)$$

Since $R(n) = \frac{n}{H^{-1}(H(n)-1)}$ and $I(n) = n \cdot g(H(n))$, by substituting in Equation (1) we get:

$$\begin{aligned} P(n) &= g(H(n)) + \frac{1}{H^{-1}(H(n) - 1)} + P(H^{-1}(H(n) - 1)) \\ &= g(H(n)) + \frac{1}{H^{-1}(H(n) - 1)} + g(H(n) - 1) + \frac{1}{H^{-1}(H(n) - 2)} + \dots + P(\Theta(1)) \end{aligned} \quad (2)$$

Since the height is $H(n)$ and $P(\Theta(1)) = \Theta(1)$ (since $\frac{C(\Theta(1))}{\Theta(1)} = \Theta(1)$), we get from Equation (2):

$$\begin{aligned} P(n) &= \sum_{i=1}^{H(n)-1} \left(g(H(n) - i + 1) + \frac{1}{H^{-1}(H(n) - i)} \right) + \Theta(1) \\ &= \sum_{i=2}^{H(n)} g(i) + \sum_{i=1}^{H(n)-1} \frac{1}{H^{-1}(i)} + \Theta(1) \end{aligned} \quad (3)$$

³Whenever $H(n)$ refers to height, we mean $\lceil H(n) \rceil$.

The first sum of Equation (3) is $\Theta(1)$ by the properties of function g . Since it always holds that $H(n) = O(\log n)$ (and as a result $H^{-1}(n) = \Omega(c^n)$, for some $c > 1$) the second sum is also $\Theta(1)$. Thus, $P(n) = \Theta(1)$ and consequently $C(n) = \Theta(n)$. As the time to build the structure is linear, the space cannot be larger and the lemma follows. \blacksquare

In general, consider an internal node v of SIST at depth i and assume that n_i leaves (elements of S) are stored in the subtree rooted at v , whose elements take values in $[\ell, u]$. Then, we have that v has degree $R(n_i) = \Theta(H^{-1}(H(n) - i + 1)/H^{-1}(H(n) - i))$, and $I(n_i) = n_i \cdot g(H(n) - i)$. The internal node v at depth i is associated with an array $\text{REP}[1..R(n_i)]$ of sample elements, containing one sample element for each of its subtrees, and an array $\text{ID}[1..I(n_i)]$ that stores a set of sample elements approximating the inverse distribution function. The role of the ID array is to partition the interval $[\ell, u]$ into $I(n_i)$ equal parts, each of length $\frac{u-\ell}{I(n_i)}$. The role of the REP array is to partition its associated ordered sub-file of S into $R(n_i)$ equal subfiles, each of size $\frac{n_i}{R(n_i)}$. By using the ID array, we can interpolate the REP array to determine the subtree in which the search procedure will continue. In particular, for the $\text{ID}[1..I(n_i)]$ array associated with node v , it holds that $\text{ID}[i] = j$ iff $\text{REP}[j] < \ell + i(u - \ell)/I(n_i) \leq \text{REP}[j + 1]$. Recall that x is the element we seek. To interpolate REP, compute the index $j = \text{ID}[\lfloor (I(n_i)(x - \ell)/(u - \ell)) \rfloor]$, and then search the REP array from $\text{REP}[j + 1]$ until the appropriate subtree is located. Furthermore, for each node we explicitly maintain parent, child, and sibling pointers. Pointers to sibling nodes will be alternatively referred to as *level links*. The required pointer information can be easily incorporated in the construction of the static interpolation search tree.

Another crucial component of our design is a search tree data structure called q^* -heap [35, 36], implemented on a unit-cost RAM with a word length of w bits – which assumes that multiplication and the standard AC^0 operations (addition, subtraction, comparison, bitwise Boolean operations and shifts) are performed in constant time on $O(w)$ -bit operands. This data structure is similar to fusion trees [12, 35, 36], but it uses q -heaps instead of an ad-hoc static table in each node of the tree and achieves the same bounds with those of the fusion tree. Let M be the current number of elements in the q^* -heap and let N be an upper bound on the maximum number of elements ever stored in the q^* -heap, imposing that $w \geq \log N$. Then, insertion, deletion and search operations are carried out in $O(1 + \log M / \log \log N)$ worst-case time after an $O(N)$ preprocessing overhead. Choosing $M = \text{polylog}(N)$, all operations are performed in $O(1)$ time. We will use this structure to guarantee constant worst-case update operations.

3 The Data Structure

For clarity we divide the description of the data structure into two parts. In the first part we provide a high level description, while in the second part we get into the details as well as the implementation of the operations on the data structure.

3.1 High-Level Description of our Data Structure

Our data structure T is designed for the classic (unit-cost) RAM. This is a direct consequence of modeling the elements of the structure as being generated by a continuous distribution, a characteristic common to *all* previous results on interpolation search [1, 13, 14, 25, 27, 28, 29, 34, 37]. Note that we do not use the arbitrary precision for any hidden costly calculation, it is just an artifact of the modeling of the source of the elements and the preservation of their nice statistical

properties captured by Lemma 2 (Section 3.2). However, in order to speed-up update and search operations, we will use at some point q^* -heaps, implying that these numbers must be truncated to numbers of adequate precision. Thus, we have to map the arbitrary precision representation of a number to its fixed precision representation and vice-versa. Of course, there are some implications of this mapping which however can be easily tackled as shown in Sections 3.2 and 4. This can be seen as a small trick to accelerate the execution of the operations supported by our data structure. In the following, we represent the truncated version of a real x to any degree of accuracy by \tilde{x} .

From a high-level point of view, our data structure T consists of two levels. The top level is the static interpolation search tree (SIST). The elements (unlike in [25]) are not stored in the leaves, but (similarly to [1]) in a set of buckets, which comprises the bottom level of our data structure. These buckets store a truncated version of the real elements along with pointers to a sorted list L containing the real elements. In particular, for each real element x , its truncated version \tilde{x} , up to a sufficiently large precision, is stored in some bucket along with a pointer to x in L . This has two advantages: (i) We treat buckets as a kind of “indexing structure” to the real elements. Since buckets store elements of fixed precision, we can employ the q^* -heap machinery to implement them, and hence accelerate the execution of the search and update operations within the buckets. (ii) We can show that the mapping from fixed precision elements to the real ones introduces only a $O(1)$ expected overhead in searching time. In particular, we show in Lemma 5 (Section 4) that for any fixed-precision number \tilde{x} , which is the truncated version of a real $x \in [a, b]$, only an expected number of $O(1)$ such real elements are mapped to \tilde{x} during a sequence of update operations. Consequently, the searching time for an element x is dominated by the time required to search SIST and the appropriate bucket in order to locate \tilde{x} . Moreover, we also show that $O(\log n)$ bits suffice in order to represent the truncated elements.

3.2 The Details of Our Data Structure

As mentioned earlier, T is a two-level data structure. The top level is a static interpolation search tree (SIST), while the bottom level consists of buckets of elements. T is maintained by incrementally performing global reconstructions [26]. Assume that S_0 is the set of stored elements at the latest reconstruction, and assume that $S_0 = \{x_1, \dots, x_{n_0}\}$ in sorted order, where $x_i \in [a, b]$, $1 \leq i \leq n_0$. The elements x_i are considered reals of arbitrary precision as to guarantee important statistical properties and also to guarantee the expected search time complexity as in [1, 25]. The top level of T is a SIST on all elements of S_0 .

The bottom level of T is a set of ρ buckets implemented as q^* -heaps [35, 36]. Each bucket \mathcal{B}_i , $1 \leq i \leq \rho$, stores a subset of (the truncated versions of) elements in S_0 and is represented by the element $rep(i) = \max\{x : \tilde{x} \in \mathcal{B}_i\}$. The set of elements stored in the buckets constitute an ordered collection $\mathcal{B}_1, \dots, \mathcal{B}_\rho$ such that $\max\{x : \tilde{x} \in \mathcal{B}_i\} < \min\{y : \tilde{y} \in \mathcal{B}_{i+1}\}$ for all $1 \leq i \leq \rho - 1$. In other words, $\mathcal{B}_i = \{\tilde{x} : x \in (rep(i-1), rep(i))\}$, for $2 \leq i \leq \rho$, and $\mathcal{B}_1 = \{\tilde{x} : x \in [rep(0), rep(1))\}$, where $rep(0) = a$ and $rep(\rho) = b$.

To be more precise, in the reconstruction stage, the set S_0 is partitioned into two sets S_1 and S_2 , where $S_1 = \{x_{i \cdot \frac{n_0}{\ln n_0}} : i = 1, \dots, \frac{n_0}{\ln n_0} - 1\} \cup \{b\}$, and $S_2 = S_0 - S_1$. The i -th element of S_1 is the representative $rep(i)$ of the i -th bucket \mathcal{B}_i , where $1 \leq i \leq \rho$ and $\rho = |S_1| = \frac{n_0}{\ln n_0}$, and is stored as a leaf of the SIST. An element $x \in S_2$ is stored twice:

1. As a leaf of the SIST in T containing x .

2. In the appropriate bucket \mathcal{B}_i is stored as \tilde{x} , iff $\text{rep}(i-1) < x \leq \text{rep}(i)$, for $2 \leq i \leq \frac{n_0}{\ln n_0}$; otherwise ($x \leq \text{rep}(1)$), \tilde{x} is stored in \mathcal{B}_1 .

This leaf in the SIST is marked as redundant and is equipped with a pointer to the representative of the bucket to which it belongs. We also mark as redundant all internal nodes of the SIST that span redundant leaves belonging to the same bucket \mathcal{B}_i and equip them with a pointer to the representative $\text{rep}(i)$ of the bucket. All redundant nodes pointing to the bucket \mathcal{B}_i constitute a subtree. This can be forced during the reconstruction stage. The bucket \mathcal{B}_i is equipped with a pointer to the root of this redundant subtree.

This redundancy may seem curious, but it has a critical role in the analysis of the expected performance of T . First, the elements of S_0 are stored in the bottom level (buckets), because they guarantee that it is highly unlikely that a bucket will become empty due to random deletions (see Section 5). Second, the elements of S_2 are stored in the top level of T (SIST), because they guarantee the expected performance of the search procedure in a similar manner to the interpolation search trees presented in [1, 25]. This is captured by the following lemma.

Lemma 2 *Let T be a SIST on a set S of n elements generated by a μ -random distribution and let T' be any subtree of T which spans a consecutive subset $S' \subset S$. Then, the elements of S' are also μ -randomly distributed.*

Proof. Similar to the proof of Lemma 4 in [25]. ■

In conjunction with T , a sorted linked list L of all the real elements in the data structure is also maintained. L is used mainly to map truncated elements, stored in the buckets, to their real counterparts and it is crucial for the reconstruction stage, since it provides all real elements sorted. To achieve this, an arbitrary element x in L maintains a pointer to its truncated version \tilde{x} in the bottom level of T and vice versa. Note that L does not need to be modified during the reconstruction stage. Finally, search operations will always conclude at list L , either by locating the element we search or its predecessor. Consequently, fingers will always point to elements of list L , since they can only be updated as a result of a search operation.

Each time the number of updates exceeds rn_0 , where r is an arbitrary constant, the whole data structure is reconstructed. Let n be the number of stored elements at this time. After the reconstruction, the number of buckets is equal to $\frac{n}{\ln n}$ and the value of the parameter N , used for the implementation of \mathcal{B}_i with a q^* -heap, is n .

Having concluded the description of the data structure and its reconstruction we move to the discussion of the update and search operations supported by T . First, we discuss update operations and then we move to the discussion of the search operation. Note that the SIST is not affected by any update operation between two consecutive reconstructions. The SIST will change only during the reconstruction stage.

Deletions can be handled quite easily. We are provided with a finger to the element, let it be ψ , subject to deletion in list L . We remove ψ from L in $O(1)$ worst-case time, since we have to manipulate only a constant number of pointers, and look at its predecessor and successor elements in L . If any of these two elements point to $\tilde{\psi}$ in bucket \mathcal{B}_i (i.e., if any of these elements points to the same truncated version as ψ did), then we do nothing. Otherwise, we follow the pointer to $\tilde{\psi}$ and remove $\tilde{\psi}$ from \mathcal{B}_i in $O(1)$ worst-case time, since the buckets are organized as q^* -heaps. This concludes the deletion operation that takes $O(1)$ worst-case time.

The *insertion* of a new element ψ follows a similar methodology. Assume that ψ is to be inserted next to an element y of L pointed to by a finger f . Initially, ψ is inserted in L next to y in $O(1)$ worst-case time, since only a constant number of pointers need to be manipulated. Then, by using the pointer of y to \tilde{y} , we determine the bucket into which $\tilde{\psi}$ will be inserted. This insertion is carried out in $O(1)$ worst-case time, due to the q^* -heap organization of the buckets. Finally, a pointer is established from $\tilde{\psi}$ to ψ . This concludes the insertion operation that takes $O(1)$ worst-case time.

Now, we turn to the description of the *search operation* which is slightly more involved than the update operations, due to the interplay between the bottom level of T and the list L . The search procedure for locating an element ψ in T , provided that the finger f points to element y in list L , is carried out as follows.

Initially, the search procedure in T involves a check as to whether ψ is to the left or to the right of y . Assume, without loss of generality, that ψ is to the right of y . The pointer from y to \tilde{y} is followed in order to determine the bucket \mathcal{B}_i in which \tilde{y} belongs. Then, two cases are considered:

1. Both elements belong to the same bucket \mathcal{B}_i . In this case, we just retrieve from the q^* -heap that implements \mathcal{B}_i the element \tilde{z}_0 which is equal to element $\tilde{\psi}$ or its predecessor in $O(1)$ worst-case time. Note that there may be many elements which have value equal to \tilde{z}_0 , let them be $\tilde{z}_1 = \tilde{z}_2 = \dots = \tilde{z}_j = \tilde{z}_0$. All these numbers correspond to real elements for which it holds that $z_0 \neq z_1 \neq \dots \neq z_j$. This is an effect of the truncation of the real elements in q^* -heaps. We call all elements \tilde{z}_i , $0 \leq i \leq j$, the **chain** of ψ . The elements z_i can be located in L by following the pointers from \tilde{z}_i to z_i . For each \tilde{z}_i , $0 \leq i \leq j$, z_i is compared to ψ and either a match is found or the largest element less than ψ is returned (predecessor).
2. The elements are stored in different buckets \mathcal{B}_i and \mathcal{B}_j containing \tilde{y} and $\tilde{\psi}$ respectively. In this case, the search starts from the root of the subtree containing the redundant elements of \mathcal{B}_i (by following the respective pointer from \mathcal{B}_i) and continues towards the root of the SIST. Assuming that node v is reached, it is checked whether ψ is stored in a descendant of v or in the right neighbour z of v . This can be easily accomplished by checking the boundaries of the REP arrays of both nodes. If they are not stored in the subtrees of v and z , then the search proceeds to the parent of v , otherwise it continues in the particular subtree using the ID and REP arrays. When a redundant node is reached, its associated pointer is followed to the appropriate bucket \mathcal{B}_k and Case 1 is invoked for this bucket.

The time complexity of the search operation can be affected by the expected length of the chain. If the length of the chain of ψ is p , then there will be an additive term of $O(p)$ in the time complexity of the search operation. Thus, it would be best to bound the length of the chains. We can provide such a bound as shown in the next section (Lemma 5).

4 Analysis of Time and Space Complexity

In this section we analyze the time complexities of the operations of our data structure. We start with the preprocessing and update bounds.

Let $n = O(n_0)$ be the number of elements in the latest reconstruction, which are stored in the sorted list L , and are drawn from a $(n \cdot g(H(n)), H^{-1}(H(n) - 1))$ -smooth distribution.

Lemma 3 *The preprocessing time and the space usage of our data structure is $\Theta(n)$. The update operations are performed in $O(1)$ worst-case time.*

Proof. The time and space bounds regarding the top level (SIST) follow from Lemma 1. The other components of our structure are built in $O(n)$ time by simply traversing the proper subtrees of the top level. The time bound of the update operations follow from the discussion in Section 3.2 and the results in [35, 36] (see also Section 2). ■

We now turn to the time complexity of the search operation. As it was mentioned earlier, the search time is affected (among others) by the expected length of the chain of an element, and we start our search time investigation by providing a bound for this length. Lemma 5 below implies that the expected length of a chain is $O(1)$. This is crucial in efficiently locating – via q^* -heaps – an arbitrary target real ψ . To see this, suppose that our q^* -heaps work with fixed precision numbers. Assume that when searching for a real ψ , the appropriate q^* -heap locates a fixed precision number \tilde{z} that coincides with or it is the predecessor of our target element. Then, Lemma 5 implies that the chain of ψ has $O(1)$ expected length, which are real elements inserted during the rn_0 update operations. Therefore, within $O(1)$ expected search time we can determine if the real ψ appears in the linked list L . Moreover, Lemma 5 guarantees that $O(\log n)$ bits suffice to represent all chains. In other words, $O(\log n)$ bits suffice to create indices as well as to represent and manipulate the truncated versions of the real elements in the buckets.

To prove Lemma 5, we will first define the minimum intervals handled by our data structure, determine an upper bound on these intervals (which is an upper bound on the number of chains), and then prove that the number of elements is probabilistically bounded.

Assume an (f_1, f_2) -smooth distribution μ , where $f_1(n) = \frac{n}{\log^{1+\epsilon} \log n}$ and $f_2(n) = n^{1-\delta} = n^\alpha$. Let the elements in S_0 ordered increasingly as x_1, x_2, \dots, x_{n_0} , that is, these n_0 real numbers are stored in our data structure at the end of the latest reconstruction. These reals belong to $[a, b] \subset \mathbb{R}$ and are drawn according to the unknown input distribution μ . For convenience, let $x_0 = a$ and $x_{n_0+1} = b$.

By the definition of the smooth distribution, the range $[a, b]$ is divided initially to $f_1(n)$ equally sized subranges each of which gets at most $\frac{\beta \cdot f_2(n)}{n}$ mass probability and expectedly $\frac{\beta \cdot f_2(n)}{n} \times n = \beta \cdot f_2(n) = \beta n^\alpha$ elements. For simplicity and without loss of generality we will not take into account β , since it is a constant. This procedure is applied recursively until we reach a sufficiently small subinterval with probability mass as low as possible in order to get $C = O(1)$ elements in expectation. Thus, if h is the number of recursions it suffices:

$$n^{\alpha^h} = C \implies h = O(\log_{1/\alpha} \log n) \quad (4)$$

Note that in the 1-st recursion the number of subranges is $f_1(n) = f_1(n^{\alpha^0}) = \frac{n}{\log^{1+\epsilon} \log n}$. In the 2-nd recursive division of the range, each subrange will be split into $f_1(n^\alpha) = f_1(n^{\alpha^1}) = \frac{n^{\alpha^1}}{\log^{1+\epsilon} \log n^{\alpha^1}}$ further subranges. In general, in the $(i+1)$ -th recursive division, each subrange produced during the i -th recursion will be divided into $f_1(n^{\alpha^i}) = \frac{n^{\alpha^i}}{\log^{1+\epsilon} \log n^{\alpha^i}}$ further subranges. We call the subranges at the final level of recursion as *non-dividable subranges*. In the following, the number of elements falling in a subrange is denoted as the *length of the subrange*.

Lemma 4 *The expected length of a non-dividable subrange is $O(1)$ and $O(\log n)$ bits suffice to represent all non-dividable subranges.*

Proof. Taking into account Eq. (4), in the final level of recursion the total number of non-dividable

subranges is

$$\prod_{i=0}^{O(\log_{1/\alpha} \log n)} f_1(n^{\alpha^i}) = \prod_{i=0}^{O(\log_{1/\alpha} \log n)} \frac{n^{\alpha^i}}{\log^{1+\epsilon} \log n^{\alpha^i}} < \prod_{i=0}^{O(\log_{1/\alpha} \log n)} n^{\alpha^i} \quad (5)$$

It follows that the total number of bits needed to represent all these non-dividable subranges is at most

$$\log \left(\prod_{i=0}^{O(\log_{1/\alpha} \log n)} n^{\alpha^i} \right) \quad (6)$$

which is

$$\sum_{i=0}^{O(\log_{1/\alpha} \log n)} \log n^{\alpha^i} = \log n \sum_{i=0}^{O(\log_{1/\alpha} \log n)} \alpha^i < \log n \sum_{i=0}^{\infty} \alpha^i = \log n \cdot \frac{1}{1-\alpha} = O(\log n) \quad (7)$$

Thus, $O(\log n)$ bits are sufficient to represent all non-dividable subranges, and as a result of the aforementioned recursive division each such non-dividable subrange is expected to contain $O(1)$ elements. \blacksquare

We are now ready to prove the following.

Lemma 5 *The expected length of an arbitrary chain is $O(1)$ or $O(\phi(n))$ with high probability, where $\phi(n)$ is any slowly growing function of n . Moreover, $O(\log n)$ bits suffice to represent the truncated version of any real element.*

Proof. First note that the number of non-dividable subranges determined by the aforementioned recursive process is an upper bound on the number of chains. Hence, by Lemma 4, $O(\log n)$ bits are sufficient to represent all chains and every chain is expected to contain $O(1)$ elements. Lemma 4 is also applied in the SIST structure which guarantees a deterministic upper bound on the total number of recursions. As a result, the height h determined by Eq. (4) is deterministic.

We now turn to proving that with high probability no chain gets more than $\phi(n)$ elements, where $\phi(n)$ is any slowly increasing function of n .

Let I_i denote the subrange at the $(i+1)$ -th recursive division, and for some integer ℓ , let E_i denote the event $E_i = \{|I_i| < \ell n^{\alpha^i}\}$. To prove our high probability claim, it suffices to prove that

$$\Pr \left[\bigcap_{i=1}^h E_i \right] \rightarrow 1 \quad (\text{as } n \rightarrow \infty).$$

We start with the following.

Claim 1 *For all integer ℓ , $\Pr [\overline{E_i}] = \Pr [|I_i| \geq \ell n^{\alpha^i}] \leq \left(\frac{c}{\ell}\right)^{\ell n^{\alpha^i}}$, where c is a constant.*

Proof. It follows by a straightforward adaptation of the proof of the claim in [25, p. 630]. \blacksquare

Set $\ell = c \cdot \phi(n)$. Then,

$$\Pr \left[\bigcup_{i=1}^h \overline{E_i} \right] \leq \sum_{i=1}^h \Pr [\overline{E_i}] \leq \sum_{i=1}^h \left(\frac{c}{\ell}\right)^{\ell n^{\alpha^i}} = \sum_{i=1}^h \left(\frac{1}{\phi(n)}\right)^{c\phi(n)n^{\alpha^i}} \quad (8)$$

where the second inequality follows by Claim 1. The next claim bounds the above summation.

Claim 2 $\sum_{i=1}^h \left(\frac{1}{\phi(n)}\right)^{c\phi(n)n^{\alpha^i}} \leq \sum_{i=1}^h \frac{1}{\phi^i(n)}$.

Proof. We rearrange the order of the summations by setting $k = h - i + 1$ (thus k runs from h to 1). Let

$$a_k = \left(\frac{1}{\phi(n)}\right)^{c\phi(n)n^{\alpha^{h-k+1}}}$$

and

$$b_k = \frac{1}{\phi^k(n)}$$

Let also $A(n) = \sum_{k=1}^h a_k$ and $B(n) = \sum_{k=1}^h b_k$. Hence, it suffices to prove that $A(n) < B(n)$. This boils down in proving that $a_k < b_k, \forall 1 \leq k \leq h$. We shall prove this using induction.

Base case ($k = 1$): It holds that $a_1 = \left(\frac{1}{\phi(n)}\right)^{c\phi(n)n^{\alpha^h}} < \frac{1}{\phi(n)} = b_1$, since $c\phi(n)n^{\alpha^h} > 1$.

Induction hypothesis ($k = m$): We assume that $a_m = \left(\frac{1}{\phi(n)}\right)^{c\phi(n)n^{\alpha^{h-m+1}}} < \frac{1}{\phi^m(n)} = b_m$.

Induction step ($k = m + 1$): We would like to prove that

$$\begin{aligned} \left(\frac{1}{\phi(n)}\right)^{c\phi(n)n^{\alpha^{h-m}}} &< \frac{1}{\phi^{m+1}(n)} \iff \\ \phi(n) \left(\frac{1}{\phi(n)}\right)^{c\phi(n)n^{\alpha^{h-m}}} &< \frac{1}{\phi^m(n)} \iff \\ \left(\frac{1}{\phi(n)}\right)^{c\phi(n)n^{\alpha^{h-m}}-1} &< \frac{1}{\phi^m(n)} \end{aligned}$$

Since by the induction hypothesis,

$$\left(\frac{1}{\phi(n)}\right)^{c\phi(n)n^{\alpha^{h-m+1}}} < \frac{1}{\phi^m(n)}$$

to complete the proof of the induction step it suffices to prove that

$$\begin{aligned} \left(\frac{1}{\phi(n)}\right)^{c\phi(n)n^{\alpha^{h-m}}-1} &< \left(\frac{1}{\phi(n)}\right)^{c\phi(n)n^{\alpha^{h-m+1}}} \iff \\ (\phi(n))^{1-c\phi(n)n^{\alpha^{h-m}}} &< (\phi(n))^{-c\phi(n)n^{\alpha^{h-m+1}}} \end{aligned}$$

Since $\phi(n) \geq 1$, the above inequality holds if and only if

$$\begin{aligned} 1 - c\phi(n)n^{\alpha^{h-m}} &< -c\phi(n)n^{\alpha^{h-m+1}} \iff \\ \phi(n) &> \frac{1}{c(n^{\alpha^{h-m}} - n^{\alpha^{h-m+1}})} \end{aligned} \tag{9}$$

In order for (9) to hold, it suffices to prove that the best lower bound of its righthand side is $O(1)$, since in such a case $\phi(n) = \omega(1)$, which holds for any arbitrarily slowly increasing function of n .

Consider the function $g(x) = n^{\alpha^{h-x}} - n^{\alpha^{h-x+1}}$, $x \in [1, h]$, which is strictly increasing in this range and thus it takes its minimum value when $x = 1$. Thus, to find the best lower bound for the righthand side of (9), we only need to consider the case where the denominator takes its minimum value, i.e., when $m = 1$. In this case the denominator becomes

$$n^{\alpha^{h-1}} - n^{\alpha^h} = n^{\alpha^h/\alpha} - n^{\alpha^h} = C^{\frac{1}{\alpha}} - C = O(1)$$

exactly as we wanted. This completes the proof of the claim. \blacksquare

Now, using (8) and Claim 2, we have that

$$\Pr \left[\bigcap_{i=1}^h E_i \right] = 1 - \Pr \left[\bigcup_{i=1}^h \overline{E}_i \right] \geq 1 - \sum_{i=1}^h \Pr \left[\overline{E}_i \right] \geq 1 - \sum_{i=1}^h \frac{1}{\phi^i(n)} = 1 - \frac{1}{\phi(n)} \cdot \frac{1 - \left(\frac{1}{\phi(n)}\right)^h}{1 - \frac{1}{\phi(n)}}$$

Consequently,

$$\lim_{n \rightarrow \infty} \Pr \left[\bigcap_{i=1}^h E_i \right] = 1 - \lim_{n \rightarrow \infty} \frac{1}{\phi(n)} \cdot \frac{1 - \lim_{n \rightarrow \infty} \left(\frac{1}{\phi(n)}\right)^h}{1 - \lim_{n \rightarrow \infty} \frac{1}{\phi(n)}} = 1 - 0 \cdot 1 = 1$$

This concludes the proof of the lemma. \blacksquare

To complete our investigation on the search time complexity, we distinguish between two cases for the sake of simplicity. First, we study the case of $(n/(\log \log n)^{1+\epsilon}, n^{1-\delta})$ -smooth densities, and then we discuss how our result can be extended to the general case of $(n \cdot g(H(n)), H^{-1}(H(n) - 1))$ -smooth densities.

4.1 The case of $(n/(\log \log n)^{1+\epsilon}, n^{1-\delta})$ -smooth densities

The complexity of our search operation is captured by the following theorem.

Theorem 1 *Suppose that the top level of T is a static interpolation search tree with parameters $R(n_0) = (n_0)^\delta$, $I(n_0) = n_0/(\log \log n_0)^{1+\epsilon}$, where $\epsilon > 0$, $0 < \delta < 1$. Let d be the number of elements between the finger f and the search element y in list L , let \mathcal{B}_i and \mathcal{B}_j be the buckets containing \tilde{f} and \tilde{y} respectively, and let n denote the current number of elements drawn from a $(n/(\log \log n)^{1+\epsilon}, n^{1-\delta})$ -smooth distribution. Then, the expected time complexity of a search operation is equal to: (a) $O\left(\frac{\log |\mathcal{B}_i|}{\log \log n} + \frac{\log |\mathcal{B}_j|}{\log \log n} + \log \log d\right)$; (b) $O\left(\frac{\log |\mathcal{B}_i|}{\log \log n} + \frac{\log |\mathcal{B}_j|}{\log \log n} + \log \log d + \phi(n)\right)$ with high probability, where $\phi(n)$ is any slowly growing function of n .*

Proof. The search operation in T can be decomposed into four basic steps (see also Figure 1): (i) the search for \tilde{y} in the q^* -heap implementing \mathcal{B}_i , (ii) the traversal of internal nodes of the static interpolation search tree, using ancestor pointers, level links and interpolation search in order to find \mathcal{B}_j containing \tilde{y} , (iii) the search for \tilde{y} in the q^* -heap implementing \mathcal{B}_j and (iv) the search in L for y .

From the results in [35, 36] (see also Section 2), the time complexity for the execution of steps (i) and (iii) is equal to $O\left(\frac{\log |\mathcal{B}_i|}{\log \log n_0} + \frac{\log |\mathcal{B}_j|}{\log \log n_0}\right)$, where n_0 is the number of stored elements at the latest reconstruction. Since $\log \log n_0 = \Theta(\log \log n)$, this time complexity is equal to $O\left(\frac{\log |\mathcal{B}_i|}{\log \log n} + \frac{\log |\mathcal{B}_j|}{\log \log n}\right)$.

Once locating \tilde{y} in \mathcal{B}_j , we get from Lemma 5 that step (iv) can be accomplished in $O(1)$ expected time, or in $O(\phi(n))$ expected time with high probability.

Step (ii) introduces the distance in the time complexity. The construction of the subtrees of redundant elements as well as the fact that no bucket gets empty with high probability (as we prove in Theorem 6), has the result that d (the distance between f and y in L) is an upper bound on the distance between \mathcal{B}_i and \mathcal{B}_j .

Suppose that for step (ii) we stop the ascension of the search procedure at node u , coming from child v and descending to child w (cf. Fig.1). It is clear that between v and w there must exist at least one separating node, call it z , otherwise we should stop the traversal at a lower height. Let u_l and z_l be the number of buckets of the SIST in the subtrees rooted at u and z respectively and let t be the height of u (without taking into account the redundant nodes). It is easy to see that $z_l \leq d \leq u_l$.

Since $t = O(\log \log u_l)$ and $z_l = (u_l)^\delta$, we conclude that $(O(2^{2^t}))^\delta \leq d \leq O(2^{2^t})$. Since the exponent δ , when considered in the double logarithmic time complexity, becomes an additive term, and since in Theorem 6 we will show that with high probability no bucket gets empty we deduce that the time complexity in the ascent phase of step (ii) is $O(\log \log d)$ with high probability. In the following, we will prove (by exploiting the probabilistic analysis in [25]) that the time complexity of the descent phase in step (ii) is $O(\log \log d)$ with high probability and the theorem will follow.

Consider the descent phase of step (ii). During the descent the algorithm visits a path P of t nodes with the last node being the root of a subtree with $\log n$ redundant elements. Let v_1, \dots, v_t be the nodes in the path listed in order of visit and consider a node v_i randomly selected in the path. By Lemma 2, the leaves (elements) of the subtree rooted at v_i are μ -random, and let n_i be their number. It is clear that for every i , $n_i \geq \log n$. In [25, Lemma 7] it was proven that, for the special case where $\delta = 1/2$ there is a constant c such that the probability that the interpolation procedure takes in v_i more than l steps is bounded from above by $(\frac{c}{l})^{l\sqrt{n_i}}$. Their analysis can be immediately extended in order to prove that for arbitrary δ there is a constant c such that the probability that the interpolation procedure takes in v_i more than l steps is bounded from above by $(\frac{c}{l})^{ln_i^{1-\delta}}$ (actually, the proof is based on Claim 1). For $l = 2c$ the above bound becomes $(\frac{1}{2})^{2cn_i^{1-\delta}}$. Let q be the probability that there is a node in P for which interpolating takes more than $2c$ steps. Then, it follows that $q \leq \sum_{i=1}^t (\frac{1}{2})^{2cn_i^{1-\delta}} \leq t(\frac{1}{2})^{2c(\log n)^{1-\delta}}$. Hence, for the probability p that the descent phase takes less than $2ct$ steps we have $p \geq 1 - t(\frac{1}{2})^{2c(\log n)^{1-\delta}}$. Since $t = O(\log \log d) = O(\log \log n)$ we get: $t(\frac{1}{2})^{2c(\log n)^{1-\delta}} \rightarrow 0$, as n grows, and thus we conclude that the descent phase in step (ii) takes $2ct = O(\log \log d)$ expected time with high probability. ■

In order to prove that the data structure has a small expected search time, we introduce a combinatorial game of balls and bins with deletions (Section 5). To obtain the desirable time complexities with high probability, we provide upper and lower bounds on the number of elements in a bucket and we show that no bucket gets empty (see Theorem 6). In particular, we show that $|\mathcal{B}_i| = O(\log n)$ with high probability for a bucket \mathcal{B}_i . Plugging this into Theorem 1, we get the main result of the paper.

Theorem 2 *Let μ be a $(n/(\log \log n)^{1+\varepsilon}, n^{1-\delta})$ -smooth density for $\varepsilon > 0$ and $0 < \delta < 1$. Then, there exists a finger search tree on n elements that for μ -random insertions and random deletions achieves an expected search time of: (i) $O(\log \log d)$; (ii) $O(\log \log d + \phi(n))$ with high probability. Here, $\phi(n)$ is any slowly growing function of n , and d is the distance between the finger and the*

search element. The space usage of the data structure is $\Theta(n)$ and the worst-case update time is $O(1)$.

4.2 Other Smooth Densities

Our analysis so far focused on the particular class of $(n/(\log \log n)^{1+\epsilon}, n^{1-\delta})$ -smooth densities where $\epsilon > 0$ and $0 < \delta < 1$. However, we can generalize our results to hold for the general class of $(n \cdot g(H(n)), H^{-1}(H(n) - 1))$ -smooth densities considered in [1], where $\sum_{i=1}^{\infty} g(i) = \Theta(1)$, and $H(n)$ is as defined in Section 2, thus being able to achieve $o(\log \log d)$ expected time complexity for several distributions.

As it is proved in [1], this class of smooth densities defines a natural hierarchy in the sense that the class of $(n \cdot g(H(n)), H^{-1}(H(n) - 1))$ -smooth densities contains the class of $(n \cdot g(F(n)), F^{-1}(F(n) - 1))$ -smooth densities as long as $H(n)$, $F(n)$, and $H(n)/F(n)$ are non-decreasing functions. Moreover, if $H(n)$ is also $o(\log n)$ (but not $O(1)$), then any member of this class is not zero on an interval.

By examining the proof of Theorem 1, we can see that the specific choice of the class of smooth densities comes into play when analyzing step (ii) of the search procedure in T . Hence, in the rest of this section we sketch the proof for the general class of densities described previously.

Let t denote the time complexity of step (ii). We can generalize the proof by applying the following argument: let $h(z)$ be the height of z (without taken into account the redundant nodes in its subtree) and $h(u)$ be the height of u (without taken into account again redundant nodes). Let z_l be the number of buckets in the subtree rooted at z and let u_l be the number of buckets in the subtree rooted at u . Then the following hold:

(i) $t = O(h(u))$; (ii) $z_l \leq d \leq u_l \Rightarrow h(z) \leq H(d) \leq h(u)$; and (iii) $h(u) = h(z) + 1$.

From (i) and (iii) we get that $t = O(h(z))$ and from (ii) we get $t = O(H(d))$. The above discussion establishes the following theorem.

Theorem 3 *Let μ be a $(n \cdot g(H(n)), H^{-1}(H(n) - 1))$ -smooth density, where $\sum_{i=1}^{\infty} g(i) = \Theta(1)$, $H(n) : \mathbb{N} \rightarrow \mathbb{R}_0^+$ is non-decreasing, invertible, with a second derivative less than or equal to zero, $o(\log n)$ and not $O(1)$, and $H^{-1}(i) \neq 0$ for $1 \leq i \leq H(n) - 1$. Then, there exists a finger search tree on n elements that for μ -random insertions and random deletions achieves an expected search time of: (i) $\Theta(H(d))$; (ii) $\Theta(H(d) + \phi(n))$ with high probability. Here, $\phi(n)$ is any slowly growing function of n , and d is the distance between the finger and the search element. The space usage of the data structure is $\Theta(n)$ and the worst-case update time is $O(1)$.*

For example, the density $\mu[0, 1](x) = -\ln x$ is $(n/(\log^* n)^{1+\epsilon}, \log^2 n)$ -smooth, and for this density $R(n) = n/\log^2 n$. This means that the height of the tree with n elements is $H(n) = \Theta(\log^* n)$ and the method of [1] gives an expected search time complexity of $\Theta(\log^* n)$. However, by applying Theorem 3, we can reduce the expected time complexity for the search operation to $\Theta(\log^* d)$, or to $\Theta(\log^* d + \phi(n))$ with high probability.

Note that there are smooth densities that do not belong to the aforementioned hierarchy of smooth densities [1]. For instance, the class μ of bounded densities, which are $(n, 1)$ -smooth and hence may be zero on an interval, achieve $H(n) = O(1)$. This implies the same expected search time with [1].

4.3 Worst-case Guarantees

To achieve worst-case complexities, we can follow the standard approach by maintaining, except for our data structure T , another structure W . The latter structure guarantees worst-case time bounds, while the T structure guarantees expected time complexities. The structure T is attached a flag *active* denoting whether this structure is valid subject to searches and updates, or invalid. In this way, between two global reconstructions of the structure, W stores all available elements while T either stores all elements (*active*=TRUE) or a past instance of the set of elements (*active*=FALSE). W can be any worst-case finger search data structure, e.g., like the one in [6]. Note, that when *active*=FALSE then only W is active for the next $O(n)$ update operations until a new version of T is constructed and thus *active*=TRUE again.

5 A Combinatorial Game of Balls and Bins with Deletions

In this section we describe a balls and bins random process that models each update operation in the structure T presented in Section 3. Consider the structure T immediately after the latest reconstruction. It contains the set S_0 of n elements (we shall use n for notational simplicity) which are drawn randomly according to the distribution $\mu(\cdot)$ from the interval $[a, b]$. The next reconstruction is performed after rn update operations on T , where r is a constant. Each update operation is either a uniformly at random deletion of an existing element from T , or a μ -random insertion of a new element from $[a, b]$ into T . To model the update operations as a balls and bins random process, we do the following.

We represent each selected element from $[a, b]$ as a *ball*. We partition the interval $[a, b]$ into $\rho = \frac{n}{\ln n}$ parts $[rep(0), rep(1)] \cup (rep(1), rep(2)] \cup \dots \cup (rep(\rho - 1), rep(\rho)]$, where $rep(0) = a$, $rep(\rho) = b$, and $\forall i = 1, \dots, \rho - 1$, the elements $rep(i) \in [a, b]$ are those defined in Section 3. We represent each of these ρ parts as a distinct *bin*.

During each of the rn insertion/deletion operations in T , a μ -random ball $x \in [a, b]$ is inserted in (deleted from) the i -th bin \mathcal{B}_i iff $rep(i - 1) < x \leq rep(i)$, $i = 2, \dots, \rho$, otherwise x is inserted in (deleted from) \mathcal{B}_1 .

5.1 Almost Uniform Bins

Our aim is to prove that with high probability (w.h.p.) the maximum load of any bin is $O(\ln n)$, and that no bin remains empty as $n \rightarrow \infty$. If we knew the distribution $\mu(\cdot)$, then we could partition the interval $[a, b]$ into $\rho = \frac{n}{\ln n}$ distinct bins (parts), $[rep_\mu(0), rep_\mu(1)] \cup (rep_\mu(1), rep_\mu(2)] \cup \dots \cup (rep_\mu(\rho - 1), rep_\mu(\rho)]$, with $rep_\mu(0) = a$ and $rep_\mu(\rho) = b$, such that a μ -random ball x would be equally likely to belong into any of the ρ corresponding bins. In other words, since these ρ bins have equal probability to receive ball x , we have that $\forall x \in [a, b]$ it holds:

$$\Pr[x \in (rep_\mu(i - 1), rep_\mu(i)]] = \int_{rep_\mu(i-1)}^{rep_\mu(i)} \mu(t) dt = \frac{1}{\rho} = \frac{\ln n}{n}, \quad i = 1, \dots, \rho = \frac{n}{\ln n}.$$

Remark 1 *The above expression implies that the unknown sequence $rep_\mu(0), \dots, rep_\mu(\rho)$ makes the event “insert (delete) a μ -random (random) element x into (from) the structure” equivalent to the event “throw (delete) a ball uniformly at random into (from) one of ρ distinct bins”. Such a uniform distribution of balls into bins is well understood and it is folklore to find conditions such that no bin remains empty and no bin gets more than $O(\ln n)$ balls.*

Unfortunately, the probability density $\mu(\cdot)$ is unknown. Consequently, our goal is to *approximate* the unknown sequence $rep_\mu(0), \dots, rep_\mu(\rho)$ with a sequence $rep(0), \dots, rep(\rho)$, that is, to partition the interval $[a, b]$ into ρ parts $[rep(0), rep(1)] \cup (rep(1), rep(2)] \cup \dots \cup (rep(\rho-1), rep(\rho)]$, aiming to prove that each bin (part) will have the element property:

$$\Pr[x \in (rep(i-1), rep(i))] = \int_{rep(i-1)}^{rep(i)} \mu(t) dt = \Theta\left(\frac{1}{\rho}\right) = \Theta\left(\frac{\ln n}{n}\right), \quad i = 1, \dots, \rho.$$

Remark 2 *The sequence $rep(0), \dots, rep(\rho)$ makes the event “insert (delete) a μ -random (random) element x into (from) the structure” equivalent to the event “throw (delete) a ball almost uniformly at random into one of ρ distinct bins”. This fact will become the cornerstone in our subsequent proof that no bin remains empty and almost no bin gets more than $\Theta(\ln n)$ balls.*

An illustration of Remarks 1 and 2 is given in Fig. 2.

The basic insight of our approach is illustrated by the following random game. Consider the part of the horizontal axis spanned by $[a, b]$, which will be referred to as the $[a, b]$ axis. Suppose that only a wise man knows the positions on the $[a, b]$ axis of the sequence $rep_\mu(0), \dots, rep_\mu(\rho)$, referred to as the *red dots*. Next, perform n independent insertions of μ -random elements from $[a, b]$ (this is the role of the set S_0). In each insertion of an element x , we add a *blue dot* in its position on the $[a, b]$ axis. At the end of this random game we have a total of n blue dots in this axis. Now, the wise man reveals the red dots on the $[a, b]$ axis, i.e., the sequence $rep_\mu(0), \dots, rep_\mu(\rho)$. If we start counting the blue dots *between* any two consecutive red dots $rep_\mu(i-1)$ and $rep_\mu(i)$, we almost always find that there are $\ln n + o(1)$ blue dots. This is because the number X_i^μ of μ -random elements (blue dots) selected from $[a, b]$ that belong in $(rep_\mu(i-1), rep_\mu(i))$, $i = 1, \dots, \rho$, is a Binomial random variable, $X_i^\mu \sim B(n, \frac{1}{\rho} = \frac{\ln n}{n})$, which is sharply concentrated to its expectation $E[X_i^\mu] = \ln n$.

The above discussion suggests the following procedure for constructing the sequence $rep(0), \dots, rep(\rho)$. Partition the sequence of n blue dots on the $[a, b]$ axis into $\rho = \frac{n}{\ln n}$ parts, each of size $\ln n$. Set $rep(0) = a$, $rep(\rho) = b$, and set as $rep(i)$ the $i \cdot \ln n$ -th blue dot, $i = 1, \dots, \rho - 1$. Call this procedure **Red-Dots**.

Remark 3 *The above intuitive argument does not imply that $\lim_{n \rightarrow \infty} rep(i) = rep_\mu(i)$, $\forall i = 0, \dots, \rho$. Clearly, since $rep_\mu(i)$, $i = 0, \dots, \rho$, is a real number, the probability that at least one blue dot hits an invisible red dot is insignificant. The above argument stresses on the crucial fact that the probability measure enclosed in the random interval $(rep(i-1), rep(i))$, $i = 1, \dots, \rho$, must be of order $\Theta\left(\frac{1}{\rho}\right) = \Theta\left(\frac{\ln n}{n}\right)$, regardless of the particular distribution density $\mu(\cdot)$.*

Theorem 4 *Let $rep(0), rep(1), \dots, rep(\rho)$ be the output of procedure **Red-Dots**, and let $p_i(n) = \int_{rep(i-1)}^{rep(i)} \mu(t) dt$. Then:*

$$\Pr\left[\exists i \in \{1, \dots, \rho\} : p_i(n) \neq \Theta\left(\frac{1}{\rho}\right) = \Theta\left(\frac{\ln n}{n}\right)\right] \rightarrow 0.$$

Proof. Let $\alpha(n) = \ln n/n$. Without loss of generality, we compute the probability that a block of $\ln n = \alpha(n)n$ consecutive blue dots is spread into a sub-interval (part) of the $[a, b]$ axis of probability measure $q(n)$. This probability equals

$$\binom{n}{\alpha(n)n} q(n)^{\alpha(n)n} (1 - q(n))^{(1 - \alpha(n))n} \sim \left[\left(\frac{q(n)}{\alpha(n)}\right)^{\alpha(n)} \left(\frac{1 - q(n)}{1 - \alpha(n)}\right)^{1 - \alpha(n)} \right]^n \quad (10)$$

where the expression on the right is asymptotically equal to the expression on the left if we use Stirling's approximation $n! \sim \left(\frac{n}{e}\right)^n \sqrt{2\pi n}$ and ignore inverse polynomial multiplicative terms. Expression (10) is a convex function of two variables ($q(n)$ and $\alpha(n)$) and achieves its maximum when $q(n) = \alpha(n)$. Hence, the expression vanishes exponentially with n , when either $q(n) = o(\alpha(n))$ or $q(n) = \omega(\alpha(n))$. There are $\rho = \frac{n}{\ln n}$ blocks of $\ln n$ consecutive blue dots. Applying the first moment method, we get that the probability that at least one block has its $\ln n$ blue dots spread into a sub-interval of $[a, b]$ axis of measure $q(n)$ is at most

$$\frac{n}{\ln n} \cdot \left[\left(\frac{q(n)}{\alpha(n)} \right)^{\alpha(n)} \left(\frac{1 - q(n)}{1 - \alpha(n)} \right)^{1 - \alpha(n)} \right]^n \rightarrow 0, \quad (11)$$

as n approaches infinity. We conclude that it is very unlikely that the probability measure $p_i(n)$ of each part $(rep(i-1), rep(i)]$, $i = 1, \dots, \rho$, defined by the sequence $rep(0), rep(1), \dots, rep(\rho)$, will be different from $\Theta(1/\rho) = \Theta(\ln n/n)$. ■

The above discussion and Theorem 4 imply the following.

Corollary 1 *If n elements are μ -randomly selected from $[a, b]$, and the sequence $rep(0), \dots, rep(\rho)$ from those elements is produced by procedure Red-Dots, then this sequence partitions the interval $[a, b]$ into ρ distinct bins (parts) $[rep(0), rep(1)] \cup (rep(1), rep(2)] \cup \dots \cup (rep(\rho-1), rep(\rho)]$ such that a ball $x \in [a, b]$ can be thrown (deleted) independently of any other ball in $[a, b]$ into (from) any of the bins with probability $p_i(n) = \Pr[x \in (rep(i-1), rep(i)]] = \frac{c_i \ln n}{n}$, where $i = 1, \dots, \rho$ and c_i is a positive constant.*

Definition 1 *Let $c = \min_i \{c_i\}$ and $C = \max_i \{c_i\}$, $i = 1, \dots, \rho$, where $c_i = \frac{np_i(n)}{\ln n}$.*

5.2 Randomness Invariance

In this section, we study the randomness properties in each of the rn subsequent insertion/deletion operations on the structure T (r is a constant).

Observe that before the process of rn insertions/deletions starts, each bin \mathcal{B}_i (i.e., part $(rep(i-1), rep(i)]$) contains exactly $\ln n$ balls (blue dots on the $[a, b]$ axis) of the n initial balls of the set S_0 . For convenience, we analyze a slightly different process of the subsequent rn insertions/deletions. Delete all elements (balls) of S_0 except for the representatives $rep(0), rep(1), \dots, rep(\rho)$ of the ρ bins. Then, insert μ -randomly n/c (see Definition 1) new elements (balls) and subsequently start performing the rn insertions/deletions. Since the n/c new balls are thrown μ -randomly into the ρ bins $[rep(0), rep(1)] \cup (rep(1), rep(2)] \cup \dots \cup (rep(\rho-1), rep(\rho)]$, by Corollary 1 the initial number of balls into \mathcal{B}_i is a Binomial random variable that obeys $B(n/c, p_i(n))$, $i = 1, \dots, \rho$, instead of being fixed to the value $\ln n$. Clearly, if we prove that for this process no bin remains empty and does not contain more than $O(\ln n)$ balls, then this also holds for the initial process.

Definition 2 *Let the random variable $M(j)$ denote the number of balls existing in structure T at the end of the j -th insertion/deletion operation, $j = 0, \dots, rn$. Initially, $M(0) = n/c$.*

The next useful lemma allows us to keep track of the statistics of an arbitrary bin.

Lemma 6 *Suppose that at the end of j -th insertion/deletion operation there exist $M(j)$ distinct balls that are μ -randomly distributed into the ρ distinct bins. Then, after the $(j+1)$ -th insertion/deletion operation the $M(j+1)$ distinct balls are also μ -randomly distributed into the ρ distinct bins.*

Proof. We use induction on j . The lemma trivially holds for $j = 0$. That is, independently (by Corollary 1) each ball $x \in [a, b]$ of the initial $M(0) = n/c$ balls belongs into \mathcal{B}_i with probability $p_i(n) = c_i/\rho = c_i \ln n/n$, $i = 1, \dots, \rho$. Suppose that it holds for the $M(j)$ balls when $j = k$. That is, each ball x , of the $M(k)$ existing balls, belongs into \mathcal{B}_i with probability $p_i(n)$, $i = 1, \dots, \rho$. We prove the lemma for $j = k + 1$.

If the $(k + 1)$ -th operation is insertion then the current number of balls is $M(k + 1) = M(k) + 1$. By Corollary 1, the *new* inserted ball x' belongs into \mathcal{B}_i independently with probability $p_i(n)$, $i = 1, \dots, \rho$. For the same reason, each ball x , of the $M(k)$ *old* balls, belongs into \mathcal{B}_i independently with probability $p_i(n)$, $i = 1, \dots, \rho$. We conclude that at the end of the $(k + 1)$ -th operation, each ball x of the total $M(k + 1) = M(k) + 1$ balls belongs into \mathcal{B}_i independently with probability $p_i(n)$.

If the $(k + 1)$ -th operation is deletion then the current number of balls is $M(k + 1) = M(k) - 1$. Due to Corollary 1, each ball x , of the $M(k) - 1$ remaining balls, belongs into \mathcal{B}_i independently with probability $p_i(n)$, $i = 1, \dots, \rho$.

We conclude that at the end of the $(k + 1)$ -th operation, each ball x of the current $M(k + 1)$ balls, belongs into \mathcal{B}_i independently with probability $p_i(n)$, $i = 1, \dots, \rho$. ■

An immediate consequence of Lemma 6 is the following lemma.

Lemma 7 *Let the random variable $Y_i(j)$ with $(i, j) \in \{1, \dots, \rho\} \times \{0, \dots, rn\}$ denote the number of balls that the i -th bin contains at the end of the j -th operation. Then, $Y_i(j) \sim B(M(j), p_i(n))$.*

5.3 Dynamics of $M(j)$

We want to study the dynamics of the current number of balls $M(j)$ existing in the structure T at the end of j -th operation, $j = 0, \dots, rn$; that is, we wish to approximate this number with high probability, for each insertion/deletion operation $j = 0, \dots, rn$. In each operation, a ball is either inserted with probability $p > 1/2$, or is deleted with probability $1 - p$. $M(j)$ is a discrete random variable which has the nice property of sharp concentration to its expected value, i.e., it has small deviation from its mean compared to the total number of operations.

In the following, instead of working with the actual values of j and $M(j)$, we shall use their *scaled* (divided by n) values t and $m(t)$, resp., that is, $t = \frac{j}{n}$, $m(t) = \frac{M(j)}{n}$, with range $(t, m(t)) \in [0, r] \times [1, m(r)]$. The following theorem provides an estimation on $m(t)$.

Theorem 5 *For each operation $0 \leq t \leq r$, the scaled number of balls that are distributed into the $\frac{n}{\ln(n)}$ bins at the end of the t -th operation equals $m(t) = (2p - 1)t + o(1)$, with high probability.*

Proof. Let the random variable γ^+tn , $\gamma^+ \in [0, 1]$, denote the current fraction of insertion operations among the currently performed tn operations, and let γ^-tn denote the remaining fraction of deletion operations. Clearly, $\gamma^+ + \gamma^- = 1$. The number $M(j) = M(tn)$ of balls that are distributed into the bins at the end of j -th operation equals the number γ^+tn of insertions minus the number γ^-tn of deletions. Consequently, $M(j) = M(tn) = (\gamma^+ - \gamma^-)tn = (\gamma^+ - (1 - \gamma^+))tn = (2\gamma^+ - 1)tn$. Therefore, $M(tn)$ depends solely on the random variable γ^+tn . Since in each of the tn operations a ball is inserted with probability p , the random variable γ^+tn of currently inserted balls obeys the binomial $B(tn, p)$ distribution. As a result, the random variable γ^+tn is sharply concentrated to its expected value ptn . That is, $\gamma^+tn \rightarrow ptn$ with high probability, as $n \rightarrow \infty$. Equivalently, $\gamma^+ \rightarrow p$ with high probability, as $n \rightarrow \infty$. ■

Remark 4 Observe that for $p > 1/2$, $m(t)$ is an increasing positive function of the scaled number t of operations, that is, $\forall t \geq 0$, $M(tn) = m(t)n \geq M(0) = m(0)n = n/c$.

Remark 4 implies that if no bin remains empty before the process of rn operations starts, since for $p > 1/2$ the balls accumulate as the process evolve, then no bin will remain empty in each subsequent operation. This is important on proving part (i) of Theorem 6.

5.4 Statistics of the Bins

In this section, we prove that before the first operation, and for all subsequent operations, with high probability, no bin remains empty. Furthermore, we prove that during each step the maximum load of any bin is $\Theta(\ln(n))$ with high probability. For the analysis below we make use of the Lambert function $LW(x)$, which is the analytic at zero solution with respect to y of the equation: $ye^y = x$ (see [9]). Recall also that during each operation $j = 0, \dots, rn$ with probability $p > 1/2$ we insert a μ -random ball $x \in [a, b]$, and with probability $1 - p$ we delete an existing ball from the current $M(j)$ balls that are stored in the structure T .

Theorem 6 (i) For each operation $0 \leq t \leq r$, let the random variable $X(t)$ denote the current number of empty bins. If $p > 1/2$, then for each operation t , $E[X(t)] \rightarrow 0$.

(ii) At the end of operation t , let the random variable $Z_\kappa(t)$ denote the number of bins with load at least $\kappa \ln(n)$, where $\kappa = \kappa(t)$ satisfies $\kappa \geq (-Cm(t) + 2)/(C \cdot LW(-\frac{Cm(t)-2}{Cm(t)\epsilon})) = O(1)$, and C is the positive constant defined in Definition 1. If $p > 1/2$, then for each operation t , $E[Z_\kappa(t)] \rightarrow 0$.

Proof. (i) Recall the definitions of the positive constants c and C (Definition 1, at the end of Section 5.1). From Lemmata 6 and 7, $\forall i = 1, \dots, \rho = \frac{n}{\ln(n)}$, it holds:

$$Pr[Y_i(t) = 0] \leq \left(1 - c \frac{\ln(n)}{n}\right)^{m(t)n} \sim e^{-cm(t)\ln(n)} = \frac{1}{n^{cm(t)}}. \quad (12)$$

From Eq. (12), by linearity of expectation, we obtain:

$$E[X(t) \mid m(t)] \leq \sum_{i=1}^{\rho} Pr[Y_i(t) = 0] \leq \frac{n}{\ln(n)} \cdot \frac{1}{n^{cm(t)}}. \quad (13)$$

From Theorem 5 and Remark 4 it holds:

$$\forall t \geq 0, \frac{1}{n^{cm(t)}} \leq \frac{1}{n^{cm(0)}} = \frac{1}{n}.$$

This inequality implies that in order to show for each operation t that the expected number $E[X(t) \mid m(t)]$ of empty bins vanishes, it suffices to show that before the process starts, the expected number $E[X(0) \mid m(0)]$ of empty bins vanishes. In this line of thought, from Theorem 5, Eq. (13) becomes,

$$E[X(0) \mid m(0)] \leq \frac{n}{\ln(n)} \cdot \frac{1}{n^{cm(0)}} = \frac{n}{\ln(n)} \cdot \frac{1}{n} = \frac{1}{\ln(n)} \rightarrow 0.$$

Finally, from Markov's inequality, we obtain

$$Pr[X(t) > 0 \mid m(t)] \leq E[X(t) \mid m(t)] \leq E[X(0) \mid m(0)] \rightarrow 0.$$

(ii) At the end of t -th operation, with high probability $m(t)n$ balls are distributed amongst the ρ distinct bins. By Lemma 7, an arbitrary \mathcal{B}_i is expected to contain $Y_i(t) = \Theta(m(t) \ln(n))$ balls, $i = 1, \dots, \rho$. Let $\mathcal{B}_{i'}$ be one of the bins that attains the maximum probability $p_{i'}(n) = C \frac{\ln n}{n}$ to receive a ball per insertion operation.

To prove that the expected number $E[Z_\kappa(t) \mid m(t)]$ of bins containing more than $\kappa \ln(n)$ balls converges to 0, it suffices to prove that for all $i = 1, \dots, \rho$, the probability of any \mathcal{B}_i to contain $Y_i(t) \geq \kappa \ln(n) > m(t) \ln(n)$ is exponentially small, for $\kappa \geq -\frac{Cm(t)-2}{C \cdot LW(-\frac{Cm(t)-2}{Cm(t)e})}$. It suffices to prove this for $\mathcal{B}_{i'}$ which is the most likely to receive balls. To this end,

$$Pr[Y_{i'}(t) \geq \kappa \ln(n) \mid m(t)] = \sum_{j=\kappa \ln(n)}^{m(t)n} Pr[Y_{i'}(t) = j \mid m(t)]. \quad (14)$$

From Lemma 7, $Y_{i'}(t)$ is a Binomial random variable. Introducing the deviation function $\delta = \delta(n)$ with range in $(0, 1)$, we get:

$$Pr[Y_{i'}(t) = \delta m(t)n \mid m(t)n] = \binom{m(t)n}{\delta m(t)n} \left(C \frac{\ln(n)}{n}\right)^{\delta m(t)n} \left(1 - C \frac{\ln(n)}{n}\right)^{(1-\delta)m(t)n}.$$

Applying Stirling's approximation: $n! \sim \sqrt{2\pi n} e^{-n} n^n$ and ignoring inverse polynomial multiplicative terms we obtain:

$$Pr[Y_{i'}(t) = \delta m(t)n \mid m(t)n] \sim \left[\left(C \frac{\ln(n)}{\delta n}\right)^\delta \left(\frac{n - C \ln(n)}{(1-\delta)n}\right)^{(1-\delta)} \right]^{m(t)n}.$$

Since we want to study the deviation $\kappa \ln(n)$ of $Y_{i'}(t)$ from its expected value $m(t)C \ln(n)$ we set the function $\delta = \delta(n) = \frac{\kappa C \ln(n)}{m(t)n}$. In this way, we have that

$$\begin{aligned} Pr[Y_{i'}(t) = \delta m(t)n \mid m(t)n] &= Pr[Y_{i'}(t) = \kappa \ln(n) \mid m(t)n] \\ &\sim \left[\left(\frac{m(t)}{\kappa}\right)^{\frac{\kappa C \ln(n)}{m(t)n}} \left(\frac{n - C \ln(n)}{n - \kappa C \ln(n)/m(t)}\right)^{\frac{m(t)n - \kappa C \ln(n)}{m(t)n}} \right]^{m(t)n} \\ &\sim \left(\frac{m(t)}{\kappa}\right)^{\kappa C \ln(n)} e^{(\kappa - m(t))C \ln(n)} e^{\frac{\ln^2(n)}{n} C(\kappa - \kappa^2/m(t))} \\ &\sim \left(\left(\frac{m(t)}{\kappa}\right)^{\kappa C} e^{C(\kappa - m(t))}\right)^{\ln(n)}. \end{aligned}$$

Therefore, for $\kappa > Cm(t)$, and by noticing that the probability density function of $Y_{i'}(t)$ has a unique maximum at the point $E[Y_{i'}(t) \mid m(t)] = m(t)C \ln(n)$ and is strictly decreasing for all points greater than $m(t)C \ln(n)$, Eq. (14) becomes:

$$\begin{aligned} Pr[Y_{i'}(t) \geq \kappa \ln(n) \mid m(t)] &= \sum_{j=\kappa \ln(n)}^{m(t)n} Pr[Y_{i'}(t) = j \mid m(t)] \\ &\leq m(t)n \cdot \left(\left(\frac{m(t)}{\kappa}\right)^{\kappa C} e^{C(\kappa - m(t))}\right)^{\ln(n)} \\ &= m(t) \left(\left(\frac{m(t)}{\kappa}\right)^{\kappa C} e^{C(\kappa - m(t)) + 1}\right)^{\ln(n)}. \end{aligned}$$

Since there are $\frac{n}{\ln(n)}$ bins, by linearity of expectation and by applying Markov's inequality, we conclude that the number $Z_\kappa(t)$ of bins with load at least $\kappa \ln(n)$, vanishes with high probability:

$$\begin{aligned}
Pr[Z_\kappa(t) > 0 \mid m(t)] &\leq E[Z_\kappa(t) \mid m(t)] \\
&\leq \sum_{i=1}^{\rho} Pr[Y_i(t) \geq \kappa \ln(n) \mid m(t)] \\
&\leq \frac{n}{\ln(n)} Pr[Y_{i'}(t) \geq \kappa \ln(n) \mid m(t)] \\
&\leq \frac{n}{\ln(n)} m(t) \left(\left(\frac{m(t)}{\kappa} \right)^{\kappa C} e^{(C(\kappa-m(t))+1)} \right)^{\ln(n)} \\
&= \frac{m(t)}{\ln(n)} \left(\left(\frac{m(t)}{\kappa} \right)^{\kappa C} e^{(C(\kappa-m(t))+2)} \right)^{\ln(n)}.
\end{aligned}$$

From the above inequality, in order to have $Pr[Z_\kappa(t) > 0 \mid m(t)] \leq E[Z_\kappa(t) \mid m(t)] \rightarrow 0$ it suffices to solve with respect to κ the following inequality:

$$\left(\frac{m(t)}{\kappa} \right)^{\kappa C} e^{(C(\kappa-m(t))+2)} \leq 1 \Leftrightarrow \kappa \geq -\frac{Cm(t) - 2}{C \cdot LW\left(-\frac{Cm(t)-2}{Cm(t)e}\right)}.$$

■

6 Conclusions

In this paper we presented a new finger search tree with $O(1)$ worst-case update time and linear space that supports finger searching queries in $O(\log \log d)$ expected time, or in $O(\log \log d + \phi(n))$ expected time with high probability, where $\phi(n)$ is any slowly growing function of n . The insertions of elements in our finger search tree are considered μ -random, where μ is $(n/(\log \log n)^{1+\epsilon}, n^{1-\delta})$ -smooth, and the deletions are random. In general, we can support $O(1)$ worst-case update time and expected search time of $O(H(d))$, or $O(H(d) + \phi(n))$ with high probability, for μ -random insertions and random deletions, where μ is $(n \cdot g(H(n)), H^{-1}(H(n) - 1))$ -smooth, g is a function satisfying $\sum_{i=1}^{\infty} g(i) = \Theta(1)$, and $H(n)$ is non-decreasing and $o(\log n)$. For several other restricted smooth densities, we can also achieve $o(\log \log d)$ expected search time, or $o(\log \log d) + O(\phi(n))$ expected search time with high probability. Our result is an improvement over the general searching problem considered in [1], since we can achieve better search bounds with high probability.

Since the techniques of Section 5 can reduce an arbitrary unknown distribution to an *almost* uniform distribution, it would be interesting to establish high probability search bounds for even larger classes than smooth distributions.

Acknowledgment: We are indebted to the anonymous referees for their valuable comments that improved considerably the presentation of the paper. In particular, we express our gratitude to one referee who helped us to resolve several subtle points.

References

- [1] A. Andersson and C. Mattsson. Dynamic Interpolation Search in $o(\log \log n)$ Time. In *Proc. 20th Coll. on Automata, Languages and Programming – ICALP'93*, LNCS Vol. 700 (Springer 1993), pp. 15-27.

- [2] A. Anderson and M. Thorup. Tight(er) Worst-case Bounds on Dynamic Searching and Priority Queues. In *Proc. 32nd ACM Symposium on Theory of Computing – STOC 2001*, pp.335-342. ACM, 2000. See also <http://arxiv.org/abs/cs.DS/0210006>.
- [3] A. Anderson and M. Thorup. Dynamic Ordered Sets with Exponential Search Trees. *Journal of the ACM* 54(3), Article 13, 2007, pp. 1-40.
- [4] M.J. Atallah, M. Goodrich and K. Ramaiyer. Biased Finger Trees and Three-Dimensional Layers of Maxima. In *Proc. 10th ACM Symposium on Computational Geometry*, pp. 150-159, 1994.
- [5] P. Beame and F. Fich. Optimal Bounds for the Predecessor problem and related problems. *Journal of Computer and Systems Sciences*, 65(1):38-72, 2002.
- [6] G.S. Brodal, G. Lagogiannis, C. Makris, A. Tsakalidis, and K. Tsihclas. Optimal Finger Search Trees in the Pointer Machine. In *Proc. 34th ACM Symposium on Theory of Computing –STOC 2002*, pp.583-592, 2002.
- [7] R. Cole, A. Frieze, B. Maggs, M. Mitzenmacher, A. Richa, R. Sitaraman, and E. Upfal. On Balls and Bins with Deletions. In *Randomization and Approximation Techniques in Computer Science – RANDOM’98*, Lecture Notes in Computer Science Vol. 1518 (Springer-Verlag, 1998), pp.145-158.
- [8] S.A. Cook and R.A. Reckhow. Time bounded random access machines. *Journal of Computer and System Sciences* 7:354–375, 1973.
- [9] R.M. Corless, G.H. Gonnet, D.E.G. Hare, D.J. Jeffrey, and D.E. Knuth. On the Lambert W Function. *Advances in Computational Mathematics* 5:329-359, 1996.
- [10] P. Dietz and R. Raman. A Constant Update Time Finger Search Tree. *Information Processing Letters*, 52:147-154, 1994.
- [11] G. Frederickson. Implicit Data Structures for the Dictionary Problem. *Journal of the ACM* 30(1):80-94, 1983.
- [12] M.L. Fredman and D.E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47:424-436, 1993.
- [13] G. Gonnet. Interpolation and Interpolation-Hash Searching. PhD Thesis. Waterloo: University of Waterloo 1977.
- [14] G. Gonnet, L. Rogers, and J. George. An Algorithmic and Complexity Analysis of Interpolation Search. *Acta Informatica* 13(1):39-52, 1980.
- [15] L. Guibas, J. Hershberger, D. Leven, M. Sharir, and R.E. Tarjan. Linear Time Algorithms for Visibility and Shortest Path Problems Inside Simple Polygons. *Algorithmica*, 2:209-233, 1987.
- [16] L. Guibas, E. McCreight, M. Plass, and J. Roberts. A new representation for linear lists. In *Proc. 9th Annual ACM Symposium on Theory of Computing – STOC’77*, pp.49-60, 1977.
- [17] T. Hagerup. Sorting and searching on the word RAM. In *Theoretical Aspects of Computer Science – STACS’98*, Lecture Notes in Computer Science Vol. 1373 (Springer-Verlag, 1998), pp. 366-398.
- [18] J. Hershberger. Finding the Visibility Graph of a Simple Polygon in Time Proportional to its Size. In *Proc. 3rd ACM Symposium on Computational Geometry*, pp. 11-20, 1987.
- [19] K. Hoffman, K. Mehlhorn, P. Rosenstiehl and R.E. Tarjan. Sorting Jordan sequences in linear time using level-linked search trees. *Information and Control*, 68(1-3):170-184, 1986.
- [20] A. Itai, A. Konheim, and M. Rodeh. A Sparse Table Implementation of Priority Queues. In *Proc. ICALP’81*, Lecture Notes in Computer Science Vol. 115 (Springer-Verlag 1981), pp. 417-431.

- [21] A. Kaporis, C. Makris, S. Sioutas, A. Tsakalidis, K. Tsihclas, and C. Zaroliagis, “Improved Bounds for Finger Search on a RAM”, in *Algorithms – ESA 2003*, Lecture Notes in Computer Science Vol. 2832 (Springer-Verlag, 2003), pp. 325-336.
- [22] D.E. Knuth. Deletions that preserve randomness. *IEEE Trans. Softw. Eng.* 3:351-359, 1977.
- [23] D.E. Knuth. *The Art of Computer Programming: Sorting and Searching*. Addison-Wesley, 1997.
- [24] K. Mehlhorn and A. Tsakalidis. *Handbook of Theoretical Computer Science – Vol I: Algorithms and Complexity*, Chapter 6: Data Structures, pp.303-341, The MIT Press, 1990.
- [25] K. Mehlhorn and A. Tsakalidis. Dynamic Interpolation Search. *Journal of the ACM*, 40(3):621-634, July 1993.
- [26] M. Overmars, J. Leeuwen. Worst Case Optimal Insertion and Deletion Methods for Decomposable Searching Problems. *Information Processing Letters*, 12(4):168-173.
- [27] Y. Pearl, A. Itai, and H. Avni. Interpolation Search – A $\log \log N$ Search. *Communications of the ACM* 21(7):550-554, 1978.
- [28] Y. Perl, E. M. Reingold. Understanding the Complexity of the Interpolation Search. *Information Processing Letters* 6(6):219-222, December 1977.
- [29] W.W. Peterson. Addressing for Random Storage. *IBM Journal of Research and Development* 1(4):130-146, 1957.
- [30] R. Seidel and C.R. Aragon. Randomized Search Trees. *Algorithmica*, 16:464-497, 1996.
- [31] R.E. Tarjan. Efficiency of a good but not linear set-union algorithm. *Journal of the ACM* 22:215225, 1975.
- [32] R.E. Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *Journal of Computer and System Sciences* 18(2):110-127, 1979.
- [33] M. Thorup. On RAM Priority Queues. *SIAM Journal on Computing*, 30(1):86-109, 2000.
- [34] D.E. Willard. Searching Unindexed and Nonuniformly Generated Files in $\log \log N$ Time. *SIAM Journal of Computing* 14(4):1013-1029, 1985.
- [35] D.E. Willard. Applications of the Fusion Tree Method to Computational Geometry and Searching. In *Proc. 3rd ACM-SIAM Symposium on Discrete Algorithms – SODA’92*, pp. 286-295, 1992.
- [36] D.E. Willard. Examining Computational Geometry, van Emde Boas Trees, and Hashing from the Perspective of the Fusion Tree. *SIAM Journal of Computing* 29(3):1030-1049, 2000.
- [37] A.C. Yao and F.F. Yao. The Complexity of Searching an Ordered Random Table. In *Proc. 17th IEEE Symp. on Foundations of Computer Science – FOCS’76*, pp. 173-177, 1976.

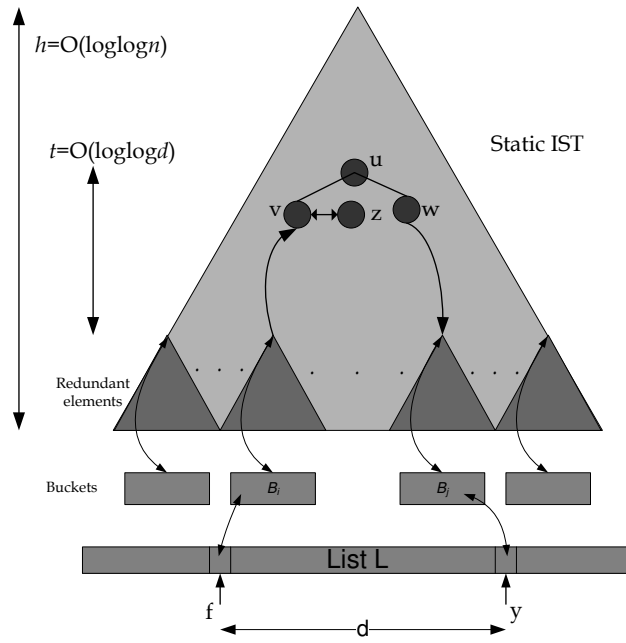


Figure 1: An overview of the tree structure as well as the search path from a finger f to an element y .

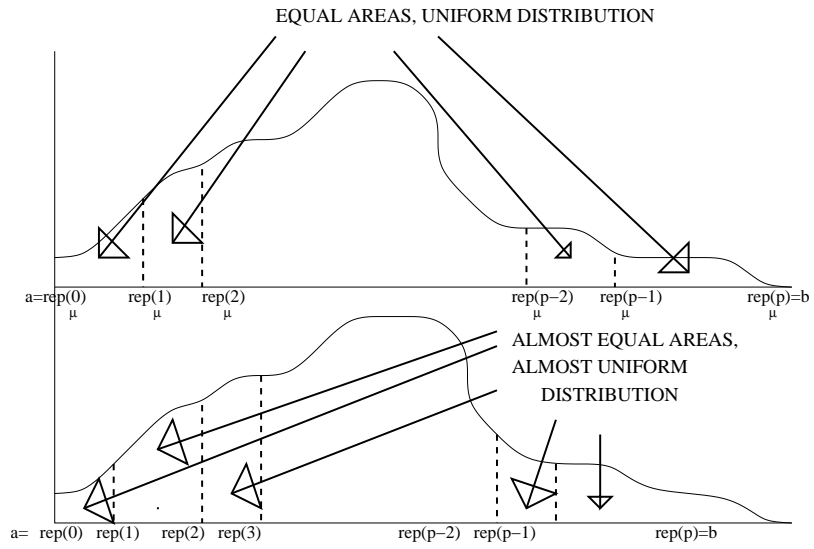


Figure 2: Plot of an unknown probability density $\mu(x), x \in [a, b]$. The upper graphic represents the uniform bins defined by: $rep_\mu(0), rep_\mu(1), \dots, rep_\mu(\rho)$. The lower graphic represents the *almost* uniform bins defined by: $rep(0), rep(1), \dots, rep(\rho)$.