Algorithmica (2013) 66:249–286
DOI 10.1007/s00453-012-9636-4

Improved Bounds for Finger Search on a RAM

Alexis Kaporis · Christos Makris · Spyros Sioutas · Athanasios Tsakalidis · Kostas Tsichlas · Christos Zaroliagis

Received: 16 December 2011 / Accepted: 23 February 2012 / Published online: 17 March 2012

Abstract We present a new finger search tree with $O(\log\log d)$ 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 Anders-

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.

A. Kaporis
Department of Information and Communication Systems Engineering, University of the Aegean, Karlovassi, Samos, 83200, Greece
e-mail: kaporisa@aegean.gr

C. Makris · A. Tsakalidis · C. Zaroliagis
Department of Computer Engineering and Informatics, University of Patras, 26500 Patras, Greece

C. Makris
e-mail: makri@ceid.upatras.gr

A. Tsakalidis
e-mail: tsak@ceid.upatras.gr

C. Zaroliagis
e-mail: zaro@ceid.upatras.gr

C. Makris · A. Tsakalidis · K. Tsichlas · C. Zaroliagis
Computer Technology Institute, N. Kazantzaki Str, Patras University Campus, 26504 Patras, Greece

K. Tsichlas (✉)
Department of Informatics, Aristotle University of Thessaloniki, Thessaloniki, Greece
e-mail: tsichlas@csd.auth.gr

S. Sioutas
Department of Informatics, Ionian University, Corfu, Greece
e-mail: sioutas@ionio.gr

son and Mattsson that exhibits expected $O(\log\log n)$ search 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 the search time is $O(\log\log d + \phi(n))$ 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 [25], efficient list splitting [25], efficient implementation of priority queues [17], efficient sorting of nearly ordered files [17] and sorting of Jordan sequences in linear time [20]. They also find applications in computational geometry, for example in constructing the visibility graph of a polygon [16, 19], in deriving optimal algorithms for the 3-dimensional layers-of-maxima problem [5], and in obtaining improved methods for dynamic point location [5].

A *finger search tree* is a leaf-oriented search tree storing n elements, in which the search procedure for a target element x can start from an arbitrary element (leaf) pointed to by a *finger* f (for simplicity, we shall not distinguish throughout the paper between an element and its key, as well as between f and the element pointed to by f). The goal is twofold: (i) to find x in a time complexity that is a function of the "distance" d, defined as the number of leaves 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.

Several results for finger search trees have been achieved on the Pointer Machine and the Random Access Machine models of computation. Before discussing the results, we review these models.

1.1 Models of Computation

A Random Access Machine (RAM) [1, 9, 37] consists of a finite program, a finite collection of registers, each of which can store a number of arbitrary (theoretically infinite) precision, and a memory consisting of a (theoretically infinite) collection of addressable locations or words (with addresses $0, 1, 2, \ldots$), where each location has the capacity of storing a number of arbitrary (theoretically infinite) precision. Arithmetic or logical operations on the contents of registers as well as reading (fetching the contents of a location into a register) and writing (storing the contents of a register in a location) operations are assumed to take one unit of time. Arithmetic operations are allowed for computing memory addresses. This model is known as the *unit-cost RAM*.

Since the manipulation of numbers of arbitrary size in unit time can result in an unreasonably powerful model (by encoding several numbers in one), a standard assumption to prevent this is to set a limit on the size of representable integers and to restrict the operations allowed on reals [37, Chap. 1]. In particular, 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; on real numbers, the typical operations allowed are comparison, addition and sometimes multiplication with no clever encoding allowed on such numbers. This RAM variant is known as the *unit-cost RAM with logarithmic word size*.

An extension of this unit-cost RAM variant is the so-called unit-cost *real RAM* [31, 34] that has become the standard model in computational geometry. The extension concerns additional operations allowed on real numbers which, apart from comparison and addition, include subtraction, multiplication, division, and analytic functions (k-root, trigonometric, exponential, logarithmic, etc.). A floor function can also be supported provided that the resulting integer has $O(\log n)$ bits (this is crucial since otherwise, we again run in an unreasonably powerful model that is able to solve in polynomial time PSPACE-complete problems [32]).

Yet another variant of the unit-cost RAM is the so-called *word RAM* [13, 18]. In this variant, 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 $[0, 2^w - 1]$. In other words, the word RAM is a unit-cost RAM with word size at least $\log n$. The restriction to integers is not crucial. Real numbers of finite precision can also be handled [3, 4, 18, 38, 40, 41], 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.

A Pointer Machine (PM) [36, 37] is similar to RAM with the exception of memory organization. In a PM, the memory consists of an unbounded collection of locations connected by pointers. Each location is divided into a fixed number of fields, and each field can hold a pointer to another location or a number of arbitrary (theoretically infinite) precision. Reading from or writing into location fields, creating or destroying a location, and operations on register contents are carried out in unit time. Contrary to RAMs, arithmetic is not allowed in order to compute the address of a location. The only way to access a location in a PM is by following pointers. The aforementioned discussion in the RAM context regarding the representation of integers and the allowed operations on reals applies also to the numbers stored in the registers and location fields of a PM [37].

1.2 Previous Work

Finger search trees with $O(1)$ update time and $O(\log d)$ search time have already been devised by Dietz and Raman [11] in the unit-cost RAM model with logarithmic word size and in which the only operation allowed on reals is comparison. Recently, for the word RAM model, Andersson and Thorup [3, 4] 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 [6]. In the PM model, Brodal et al. [7] presented 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 [24]. The only operation allowed on the numbers (reals) stored in location fields of PM is comparison. 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 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 [33] on the PM model, 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 famous example for the latter, on the RAM model, is the method of *interpolation search*, first suggested by Peterson [30]. Unlike classical search methods, which use an arbitrary rule to select a splitting element (e.g., the middle element in binary search) to split the input into two subfiles aiming to reduce the size of the subfile to be searched, the main idea of interpolation search is to select the splitting element by taking advantage of the statistical properties of the input elements. In this way, the consecutive splitting elements are spread closer and closer to the target element x, thus gradually eliminating the size of the subfile to be searched for x and improving the search time. The interpolation search method for random data generated according to the *uniform* distribution achieves $\Theta(\log\log n)$ expected search time. This was shown in [15, 28, 42]. Willard in [39] showed that this time bound holds for an extended class of distributions, called *regular*.[1]

A remark is in place w.r.t. the variant of the RAM model used in the aforementioned results concerning interpolation search. The input elements are real numbers, since they are produced by a continuous probability distribution. This assumption does not aim to illegally facilitate searching via any hidden-cost operations, it only aims to simplify the probabilistic analysis of the algorithms, since the conditional input distribution of the subfile remains unaffected per recursive application of interpolation steps. Specific operations on these reals are required in order to carry out the interpolation step (see Sect. 2 for details); namely, these operations include (except for comparison) subtraction, multiplication, division and the floor function.

Hence, although not explicitly stated, all the aforementioned papers [15, 28, 30, 39, 42] as well as the ones [2, 14, 26, 29] that will be discussed in the remainder of this subsection use the unit-cost real RAM model, without supporting analytic functions, but enhanced with the floor function where it is implicitly assumed that the resulting integer has $O(\log n)$ bits, since it indexes a particular array of size n.

[1] A 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 $b_1 \leq x \leq b_2$.

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 [12, 21] for insertions and deletions performed according to the uniform distribution, and continued by Mehlhorn and Tsakalidis [26], and Andersson and Mattsson [2] 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 [23]).

The notion of *smooth* input distributions that determine insertions of elements in the update sequence were introduced in [26], and were further generalized and refined in [2]. 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). Formally:

Definition 1 [2] Given two functions f_1 and f_2, a density function $\mu = \mu[a,b](x)$ is (f_1, f_2)-*smooth* if there exists a constant β, such that for all c_1, c_2, c_3, $a \le c_1 < c_2 < c_3 \le b$, and all integers n, it holds that

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

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 $c_1 \le x \le c_3$ 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 [2, 25]. Actually, any probability distribution is $(f_1, \Theta(n))$-smooth, for a suitable choice of β.

In [26] a dynamic interpolation search data structure was introduced, called *Interpolation Search Tree* (IST). This data structure requires $O(n)$ space for storing n real 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 $(\lceil n^\alpha \rceil, \sqrt{n})$-smooth density function and $\frac{1}{2} \le \alpha < 1$. 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^α subintervals, where $1/2 \le \alpha < 1$. 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.

In [2], 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 [27]. The resulting new data structure in [2] is called the *Augmented Sampled Forest* (ASF). Assuming that $H(n)$ is an non-decreasing, invertible and $o(\log n)$ function (whose full details are given in Sects. 2 and 4.2) denoting the height of the static implicit tree, Andersson and Mattsson [2] 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 $(\lceil n^{\alpha}\rceil, \sqrt{n}])$-smooth $\subset$ $(n/(\log\log n)^{1+\epsilon}, n^{1-\delta})$-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 RAM model. We remind that, although not explicitly stated, all the aforementioned papers [2, 14, 15, 26, 28–30, 39, 42] use the unit-cost real RAM model, without supporting analytic functions, but enhanced with the floor function where it is implicitly assumed that the resulting integer has $O(\log n)$ bits, since it indexes the ID array.

1.3 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 $O(\log\log d)$ expected search time, with d being the distance of the target element from the finger element. It works on the unit-cost real RAM model of computation for the same class of unknown smooth density functions μ considered in [2], 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 search time is $O(\log\log d + \phi(n))$ *with high probability*,[2] where $\phi(n)$ is any slowly growing function of n (e.g., the inverse Ackermann function [35]). The update time of our data structure is $O(1)$. In particular, deletions are performed in $O(1)$ worst-case time, while insertions are performed in $O(1)$ time with high probability. Using standard techniques, we are also able to

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

show that in the worst-case we can achieve $O(\sqrt{\log d/\log\log d})$ search time and $O(1)$ update (insertion or deletion) time. Moreover, for the same classes of restricted smooth densities considered in [2], 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 [2, 26] 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* (SIST—see Sect. 2). The elements (unlike in [26]) are not stored in the leaves, but (similarly to [2]) 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. Actually, we show that $O(\log n)$ bits suffice to represent the truncated elements. Buckets are treated as a kind of "indexing structure" to the real elements and are implemented using the q^*-heap machinery [13, 40, 41] (Sect. 2.2). 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 (SIST) and bottom level data structures (buckets) can give better bounds, since deletions of elements may create long sequences of consecutive empty buckets (which all must be parsed, till the first non empty one that contains the predecessor of the target element). 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 behavior 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 consecutive sequences 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., [8]), 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 first 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 unit-cost real RAM (without analytic functions). 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 [2, 14, 15, 26, 28–30, 39, 42]. 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 the standard operations of the unit-cost real RAM, 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 can indeed make this assumption, since we prove that a precision of $O(\log n)$ bits suffices for truncated elements. 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 (truncated real numbers), can be performed in constant worst-case time on $O(\log n)$-bit operands.

The remainder of the paper is organized as follows. In Sect. 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 Sect. 3, while the analysis of the time complexities of its operations is discussed in Sect. 4. The analysis of the combinatorial game, upon which our expected search time is based, is given in Sect. 5. We conclude in Sect. 6. A preliminary version of this work appeared in [22].

2 Preliminaries

The *predecessor search* 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 $X_i \in [a, b] \subset \mathbb{R}$, $1 \le i \le n$, obeys an unknown (in our case (f_1, f_2)-smooth; see Definition 1) 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 y, 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). In this paper, we will mainly deal with the *finger search* variant of the predecessor search problem, where the searching procedure does not necessarily start from the entry point of the data structure, but from an arbitrary element (leaf of the search tree) pointed to by a finger f.

2.1 Static Interpolation Search Tree

One crucial component of our design is a search tree data structure, which we call *Static Interpolation Search Tree* (SIST). It is a static and explicit version of the search trees used in [2, 26] that both address the predecessor search problem. Following [2, 26], a static interpolation search tree corresponding to the ordered file S of n elements, stored in the leaves of SIST, is fully characterized by three functions $H(n) : \mathbb{N} \to \mathbb{R}_0^+$, $R(n) : \mathbb{N} \to \mathbb{R}_0^+$ and $I(n) : \mathbb{N} \to \mathbb{R}_0^+$, which are non-decreasing and invertible with a second derivative less than or equal to zero. $H(n)$ denotes the height of the tree.[3] $R(n)$ denotes the out-degree[3] of the root of the tree, splitting the ordered file of n elements into $R(n)$ equal subfiles. That is, these $R(n)$ children of the root node partition the ordered file S into $R(n)$ equal subfiles

$$S_1 = \{X_{(1)}, \dots, X_{(\frac{n}{R(n)})}\}, \quad \dots, \quad S_{R(n)} = \{X_{((R(n)-1)\frac{n}{R(n)}+1)}, \dots, X_{(n)}\} \qquad (1)$$

[3]Whenever $H(n)$ refers to height, $R(n)$ refers to out-degree, and $I(n)$ refers to number of equal parts, we mean $\lceil H(n) \rceil$, $\lceil R(n) \rceil$, and $\lceil I(n) \rceil$, respectively.

$I(n)$ denotes the number of equal parts[3] that the interval $[a,b] \subseteq \mathbb{R}$ (in which the elements of F lie) is partitioned. That is, there are $I(n)$ parts denoted as

$$\begin{aligned} &I_1 = \left(a, a + \frac{b-a}{I(n)}\right], \quad I_2 = \left(a + \frac{b-a}{I(n)}, a + 2\frac{b-a}{I(n)}\right], \quad \ldots, \\ &I_{I(n)} = \left(a + \big(I(n)-1\big)\frac{b-a}{I(n)}, a + I(n)\frac{b-a}{I(n)}\right] \end{aligned} \tag{2}$$

of size $\frac{b-a}{I(n)}$ each.

In general, each node v at depth $i = 0, \ldots, H(n)$ of SIST is the corresponding root of a subtree of n_i elements (with $n_0 = n$) and is associated to a pair of arrays, namely ID and REP, of size $I(n_i)$ and $R(n_i)$, respectively. The ID and REP arrays help to locate the appropriate child (subtree of $\frac{n_i}{R(n_i)}$ elements) of node v eligible to contain the predecessor of the target element y. This is achieved as follows.

For the root of SIST, index REP[i], $i = 1, \ldots, R(n)$, points to the i-th subfile $S_i = \{X \in S \mid X_{((i-1)\frac{n}{R(n)})} < X \leq X_{(i\frac{n}{R(n)})}\}$, as defined in (1) above. More compactly, REP[i] can be seen as the *representative* of the element $X_{(i\frac{n}{R(n)})}$ of subfile S_i; i.e., REP[i] maps only to $X_{(i\frac{n}{R(n)})}$. On the other hand, the i-th index ID[i], $i = 1, \ldots, I(n)$, of the root node of SIST points to each representative element $X_{(i_1\frac{n}{R(n)})}, \ldots, X_{(i_k\frac{n}{R(n)})} \in I_i$, with I_i as defined in (2) above. In other words, the i-th ID index points to *all* (possibly many) representatives spread within the i-th part of interval $[a,b] \subseteq \mathbb{R}$. When searching for the predecessor of a target element y, the first *interpolation* step determines in $O(1)$ time the natural number j_y

$$j_y = \left\lfloor \frac{y-a}{b-a} I(n) \right\rfloor + 1 \tag{3}$$

which denotes the j_y-th interval I_{j_y} of length $\frac{b-a}{I(n)}$ where the target element y lies:

$$I_{j_y} = \left(a + (j_y - 1)\frac{b-a}{I(n)}, a + j_y\frac{b-a}{I(n)}\right] \tag{4}$$

Suppose that in this subinterval I_{j_y} lie $k = O(1)$ REP indices (representative elements). Then we can determine within $O(1)$ time the unique REP index that corresponds to the subfile that the predecessor of the target element y must be subsequently searched for. This is because y is compared with at most $O(1)$ other representatives that lie in I_{j_y}.

For an internal node v of SIST, the associated ID and REP arrays can be defined in a similar way. Consider an internal node v of SIST at depth $i \geq 0$ and assume that n_i (recall $n_0 = n$) elements of S are stored in the subtree rooted at v taking values in the subinterval $[\ell, u] \subseteq [a,b] \subseteq \mathbb{R}$. The internal node v at depth i is associated with an array REP[$1..R(n_i)$] of sample elements, containing one representative element for each of its subtrees and an array ID[$1..I(n_i)$]. Similarly to the case of the root node above, the ID array partitions the interval $[\ell, u] \subseteq \mathbb{R}$ into $I(n_i)$ equal parts, each of length $\frac{u-\ell}{I(n_i)}$, and the REP array partitions its associated ordered subfile of S into $R(n_i)$ equal subfiles, each of size $\frac{n_i}{R(n_i)}$. 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 SIST. The ID array is used to interpolate the REP array in order to determine the subtree of v from which the search procedure will continue, in a way similar to the first interpolation search at the root node. In particular, the ID$[1..I(n_i)]$ array associated with v has the property that $\forall s = 1, \dots, I(n_i)$ the s-th ID entry ID$[s]$ points to each j-th entry of REP$[1..R(n_i)]$ such that (recall (4))

$$\mathrm{REP}[j] \in \left(\ell + (s-1)\frac{u-\ell}{I(n_i)}, \ell + s\frac{u-\ell}{I(n_i)}\right] \tag{5}$$

Now, when searching for the predecessor of target element y, we interpolate the REP array and locate the corresponding representative of the subtree containing y's predecessor by working similarly to (3): compute in $O(1)$ time the index $j'_y = \lfloor (y-\ell)/(u-\ell)) \rfloor I(n_i) + 1$ of the ID array of node v. In turn, this computed ID$[j'_y]$ entry points to all representatives REP$[j]$ satisfying (5) with respect to $s = j'_y$, and hence it remains to sequentially search within these representatives until the appropriate subtree containing y's predecessor is located.

Having discussed SIST in detail, it remains to describe some important properties of the functions $H(n)$, $R(n)$, and $I(n)$ towards formally establishing in Lemma 1 the time and space bounds for building a SIST. As mentioned in the Introduction, we are particularly interested in the class of $(n \cdot g(H(n)), H^{-1}(H(n)-1))$-smooth distributions.

First, as (1) easily implies, each child at depth 1 of the root node corresponds to a subfile of S of size $n_1 = n/R(n)$ (recall that $n_0 = n$). Hence, for the height of SIST we get $H(n_1) = H(n/R(n)) = H(n) - 1$. Recall that H is invertible, so by applying H^{-1} to the previous equation we get $n/R(n) = H^{-1}(H(n)-1)$. Solving this with respect to $R(n)$ we get that, for having a SIST of height $H(n)$, the degree $R(n)$ must be $R(n) = n/H^{-1}(H(n)-1)$. This dictates that $H^{-1}(i) \neq 0$, for $1 \leq i \leq H(n)-1$. In order to handle the largest 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

$$I(n) = n \cdot g\big(H(n)\big) \tag{6}$$

for a function $g : \mathbb{N} \to \mathbb{R}$ such that $\sum_{i=1}^{\infty} g(i) = \Theta(1)$, so that the space of SIST remains linear.

Now, consider a node v at depth $i \geq 0$ and assume that n_i elements of S are stored in the subtree rooted at v. Then, node v has $R(n_i)$ children, each one corresponding to a subfile of S of size $n_{i+1} = n_i/R(n_i)$. It can be easily verified that $H(n_i) = H(n) - i$, which implies that

$$n_i = H^{-1}\big(H(n) - i\big) \tag{7}$$

Since $n_{i+1} = n_i/R(n_i)$, we have that $H(n_i/R(n_i)) = H(n_{i+1}) = H(n_i) - 1 = H(n) - i - 1$ implying that

$$n_i/R(n_i) = H^{-1}\big(H(n) - i - 1\big) \tag{8}$$

or that v has degree $R(n_i) = H^{-1}(H(n)-i)/H^{-1}(H(n)-i-1))$. Moreover, $I(n_i) = n_i \cdot g(H(n_i)) = n_i \cdot g(H(n) - i)$.

Since we are interested in the class of $(n \cdot g(H(n)), H^{-1}(H(n) - 1))$-smooth distributions, we observe that by choosing $H(n) = \Theta(\log n)$ we get the class of $(n \cdot g(\Theta(\log n)), \Theta(n))$-smooth distributions that contains all densities. Hence, as in [2], we can always assume that $H(n) = O(\log n)$. The following lemma characterizes the time and space complexity of SIST.

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

Proof The proof is similar to the proof of [2, Theorem 6] and we provide it here for completeness.

Let $C(n)$ be the time to build a SIST on 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 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

$$P(n) = \frac{I(n)}{n} + \frac{R(n)}{n} + P\left(\frac{n}{R(n)}\right) \tag{9}$$

Taking into account (8), we have $\frac{R(n_i)}{n_i} = \frac{1}{H^{-1}(H(n)-i-1)}$, and (6) gives $\frac{I(n_i)}{n_i} = g(H(n) - i)$, for $i \geq 0$ with $n_0 = n$. Substituting these, with $n = n_0$, into (9) we get

$$P(n) = g\bigl(H(n)\bigr) + \frac{1}{H^{-1}(H(n) - 1)} + P\bigl(H^{-1}\bigl(H(n) - 1\bigr)\bigr) \tag{10}$$

Equation (7) gives $H^{-1}(H(n) - 1) = n_1$, and hence (10) becomes

$$\begin{aligned} P(n) &= g\bigl(H(n)\bigr) + \frac{1}{H^{-1}(H(n) - 1)} + P(n_1) \\ &= g\bigl(H(n)\bigr) + \frac{1}{H^{-1}(H(n) - 1)} + g\bigl(H(n) - 1\bigr) + \frac{1}{H^{-1}(H(n) - 2)} + \cdots \\ &\quad + P\bigl(\Theta(1)\bigr) \end{aligned} \tag{11}$$

Since the height is $H(n)$ and $P(\Theta(1)) = \frac{C(\Theta(1))}{\Theta(1)} = \Theta(1)$, we get from (11):

$$\begin{aligned} P(n) &= \sum_{i=1}^{H(n)-1} \left(g\bigl(H(n) - i + 1\bigr) + \frac{1}{H^{-1}(H(n) - i)}\right) + \Theta(1) \\ &= \sum_{i=2}^{H(n)} g(i) + \sum_{i=1}^{H(n)-1} \frac{1}{H^{-1}(i)} + \Theta(1) \end{aligned} \tag{12}$$

The first sum of (12) is $\Theta(1)$ by the properties of function g (see the line following (6)). 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. □

Remark 1 Equations (6) and (8) imply that the class of $(n \cdot g(H(n)), H^{-1}(H(n)-1))$-smooth distributions can be alternatively written as the $(I(n), n/R(n))$-smooth class.

Remark 2 It is easy to check that a SIST on n elements with parameters $R(n) = n^\delta$ and $I(n) = n/(\log\log n)^{1+\varepsilon}$, where $\varepsilon > 0$ and $0 < \delta < 1$, has height $H(n) = O(\log\log n)$.

2.2 q^*-Heaps

Another crucial component of our design is a search tree data structure called *q^*-heap* [40, 41], originally implemented on a word RAM of word length w. This data structure is similar to fusion trees [13, 40, 41], 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.

2.3 Global Rebuilding

To guarantee good update bounds in our new finger search data structure, we make use of the well-known incremental global rebuilding technique [27]. For reasons of completeness, we present in this subsection the main idea of incremental global rebuilding, establishing the analogue of [27, Theorem 1] with respect to the predecessor search problem, and refer the reader to [27] for the full details. We start with a few definitions.

Let T be a structure of n elements, and let $P_T(n)$ be the construction time of T, $Q_T(n)$ be the time to answer a predecessor query on T, $D_T(n)$ be the time for deleting an element from T, and $I_T(n)$ be the time for inserting an element in T. Updates are called *weak* if the procedures to carry them out on a newly constructed structure of n elements merely guarantee that, after rn total updates ($r < 1$), the query time on the resulting set is still bounded by $Q_T(n)$. Let $WD_T(n)$ and $WI_T(n)$ be the time to perform a weak deletion and insertion respectively on the structure T of n elements.

The global rebuilding technique is based on the maintenance of two structures for T, which are called OLD-MAIN and MAIN. Usually only the MAIN structure exists. Assume that a new MAIN has taken the place of the OLD-MAIN and let its size be n_0. When $\frac{1}{2}n_0$ updates are performed on MAIN, then it changes to OLD-MAIN and a construction is initiated to build a new MAIN incrementally. In the meanwhile, update and query operations are carried out in OLD-MAIN, until the new MAIN under construction takes over. This new MAIN is constructed on the set of elements currently stored by OLD-MAIN, let this number be n_1, where $n_0 - \frac{1}{2}n_0 \le n_1 \le n_0 + \frac{1}{2}n_0$. The construction of the new MAIN is performed incrementally and in an accelerated pace. The idea is that, for each one of the next $r_1 n_1$ update operations, $\frac{P_T(n_1)}{r_1 n_1}$ work is performed for the construction of the new MAIN, for some constant $r_1 < \frac{1}{3}$, since in that case $rn_0 = \frac{1}{2}n_0 + r_1 n_1 < \frac{1}{2}n_0 + \frac{1}{3} \cdot \frac{3}{2}n_0 = n_0$, and thus $r < 1$ as required. Such a process would certainly result in fully constructing the new MAIN in time. However, the update operations occurring during this incremental construction (and which are carried out on OLD-MAIN) will make the new MAIN outdated by the time its construction is completed. Hence, before the new MAIN is released, it should be updated with the update operations (insertions and deletions) occurred during its construction. For this reason, all update operations that occur during the construction of the new MAIN are inserted in a queue Q (apart from being carried out on OLD-MAIN). Now, to cater also for the updating of MAIN and still be in time, its incremental construction is sped up by actually performing more than $\frac{P_T(n_1)}{r_1 n_1}$ construction work[4] per update operation. After the construction of MAIN on the n_1 elements, all update operations in the queue Q have to be performed on MAIN. To ensure that the whole process will overtake itself (i.e., the new MAIN will take over from OLD-MAIN after carrying out a total of at most $r_1 n_1$ updates), the update operations already in Q have to be performed at an *accelerated pace* after each of the *new* update operations; note that these new update operations continue to come, still affect the OLD-MAIN, and are also inserted in Q. By *accelerated pace* we mean that for each new update operation, a number of $c > 1$ updates in Q are performed. Since the processing of these updates is accelerated, we are bound to empty Q, and as soon as this happens OLD-MAIN is discarded and the new MAIN is released. By choosing appropriately the parameter c, it can be guaranteed that the new MAIN takes over after at most $r_1 n_1$ updates (for details see [27]).

Let T' be the structure consisting of MAIN and OLD-MAIN. The time bounds for searching and updating T' can be easily derived from those of T as follows. First observe that the cost of an update operation in T' is the cost of a weak update in the OLD-MAIN (i.e., in T) as well as the worst-case cost of an incremental work related to the construction of the new MAIN. The cost of this incremental work is $O(\frac{P_T(n_1)}{n_1})$, since MAIN is going to be completed after $r_1 n_1$ update operations. Now, the time for searching T' comes straightforwardly from the fact that searching is applied to OLD-MAIN (i.e., to T), for which weak updates guarantee that the search time will remain $O(Q_T(n_1))$.

The above discussion provides a proof sketch of the following theorem, whose full proof details can be found in [27].

Theorem 1 *Given a structure T with n elements for the predecessor search problem, there is a structure T' for the same problem such that*: $Q_{T'}(n) = O(Q_T(n))$,

[4]Actually, $\frac{P_T(n_1)}{r_1 n_1} + WI_T(n_1(1+r_1))$ work per insertion, and $\frac{P_T(n_1)}{r_1 n_1} + WD_T(n_1(1+r_1))$ work per deletion [27]. As it is also shown in [27] and adopted in our case, any $r_1 < \frac{1}{3}$ suffices for the global rebuilding technique.

$D_{T'}(n) = O(WD_T(n) + \frac{P_T(n)}{n})$, and $I_{T'}(n) = O(WI_T(n) + \frac{P_T(n)}{n})$. All bounds are worst-case.

3 The Data Structure

For clarity we divide the description of the data structure into two parts. In the first part (Sect. 3.1) we provide a high level description, while in the second part (Sect. 3.2) 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

As stated in the Introduction, our data structure T is designed for the unit-cost real RAM. This is a direct consequence of modeling the elements of the structure as being generated by a continuous distribution. It serves as a common simplifying assumption for the probabilistic analysis of SIST (Sect. 2.1), constituting the upper static part of our data structure. 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 (Sect. 3.2). However, in order to speed-up update and search operations, we implement each bucket (SIST subtree of $O(\log n)$ elements), constituting the lower dynamic level of our data structure, as a q^*-heap (Sect. 2.2).

This implies that the real numbers of the upper SIST structure must be truncated to numbers of adequate precision to be processed by the lower q^*-heap structures. Thus, we have to map the arbitrary precision representation of a real number (needed for traversing probabilistically fast the upper SIST structure towards landing to the appropriate bucket) to its fixed precision representation (needed for updating deterministically fast the landed bucket) and vice-versa. Of course, there are some implications of this mapping which however can be easily tackled as shown in Sects. 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 $\widetilde{x}$, and we denote by $|\widetilde{x}|$ the number of bits in the binary representation of $\widetilde{x}$.

From a high-level point of view, our data structure T consists of two levels. The top level is a SIST (Sect. 2.1), where the recursive interpolation step stops when a subfile of size $\Theta(\log n)$ is encountered, namely a bucket. Thus, the elements (unlike in [26]) are not stored in the leaves of SIST, but (similarly to [2]) in a family of buckets being implemented as q^*-heaps (Sect. 2.2), which comprise the bottom dynamic level of our data structure. These buckets store a truncated version of the real elements along with pointers to a sorted doubly-linked list L containing the real elements. In particular, for each real element x, its truncated version $\widetilde{x}$, up to a sufficiently large precision, is stored in some bucket along with a pointer to x in L. Due to the limited number of bits in the truncation, it is possible that k such distinct real elements $x_1 \neq \cdots \neq x_k$ in L coincide when truncated to $\widetilde{x}$. Hence, the following definition is in place concerning the *chain* of a truncated element along with its associated pointers.

Definition 2 Given an arbitrary truncated element $\widetilde{x} \in [a, b]$, its *chain* of *length k* consists of all real elements $x_1 \neq \cdots \neq x_k \in L$ whose truncated value equals $\widetilde{x}$. Furthermore, there is a pointer from each such real element x_i, $1 \leq i \leq k$, to $\widetilde{x}$, while there is a pointer from $\widetilde{x}$ to just one of the reals x_i.

Given that an arbitrary $\widetilde{x}$ is being returned by a q^*-heap, the above representation introduces an additional $\Theta(k)$ time overhead for locating the targeted real element x, since we have to determine (via sequential search) in its chain the corresponding real x_j, $j \in [k]$, stored in L such that $x_j = x$, or its predecessor. We show (Lemma 4) that with high probability $O(\log n)$ bits suffice to represent the truncated elements stored in the q^*-heaps, in a way that we can retrieve and update in $O(1)$ expected time from the associated list L the real elements. This is achieved by showing that the expected chain length is $O(1)$. Furthermore, we show that (Corollary 1) the chain length is very unlikely to increase with respect to n. In other words, in expectation only $O(1)$ real elements stored in the list L are mapped to an arbitrary truncated number $\widetilde{x}$, such that $|\widetilde{x}| = O(\log n)$, for the wide class of unknown smooth input distributions.

Our approach 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) The mapping from fixed precision elements to the real ones introduces only a $O(1)$ expected overhead in searching time, and hence the searching time for an element x is dominated by the time required to search SIST and the appropriate bucket in order to locate $\widetilde{x}$.

3.2 The Details of Our Data Structure

3.2.1 Construction and Maintenance

Our data structure T is a two-level structure. The top level is a static interpolation search tree (the SIST), while the bottom level consists of buckets of elements. T is maintained by incrementally performing global reconstructions using the global rebuilding technique of Theorem 1. Assume that S_0 is the set of stored elements at the latest reconstruction, where $S_0 = \{x_1, \dots, x_{n_0}\}$ in sorted order and $x_i \in [a, b]$, $1 \leq i \leq n_0$. 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 [40, 41]. 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 $rep(i) = \max\{x : \widetilde{x} \in \mathcal{B}_i\}$. The set of elements stored in the buckets constitute an ordered collection $\mathcal{B}_1, \dots, \mathcal{B}_\rho$ such that $\max\{x : \widetilde{x} \in \mathcal{B}_i\} < \min\{y : \widetilde{y} \in \mathcal{B}_{i+1}\}$ for all $1 \leq i \leq \rho - 1$. In other words, $\mathcal{B}_i = \{\widetilde{x} : x \in (rep(i-1), rep(i)]\}$, for $2 \leq i \leq \rho$, and $\mathcal{B}_1 = \{\widetilde{x} : x \in [rep(0), rep(1)]\}$, where $rep(0) = a$ and $rep(\rho) = b$.

To be more precise, in the reconstruction stage, the set $S_1 = \{x_{i \cdot \ln n_0} : i = 1, \dots, \frac{n_0}{\ln n_0} - 1\} \cup \{b\}$ is defined. 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 $\rho = |S_1| = \frac{n_0}{\ln n_0}$. An element $x \in S_0$ 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 $\widetilde{x}$, iff $rep(i-1) < x \leq rep(i)$, for $2 \leq i \leq \rho$; otherwise ($x \leq rep(1)$), $\widetilde{x}$ is stored in $\mathcal{B}_1$.

Each element x in a leaf of the SIST maintains a pointer to the bucket which contains $\widetilde{x}$. Additionally, each bucket maintains a pointer to the leaf containing its representative. All representatives are connected in an ordered doubly-linked list $\mathcal{R}$. This list is used during insertions to determine the correct bucket to update.

The purpose of storing the elements in the leaves of the SIST, during reconstruction, is to define the range of values stored in the buckets. Note that during insertions and deletions the SIST (top level of T) remains unaffected – the insertions and deletions are carried out at the bottom level of T. This means that a leaf of the SIST may correspond to an element which has been deleted, while an element in a bucket may not be stored in any leaf of the SIST. Knowing, however, the range of values of elements stored in the buckets allows us, with the help of the list $\mathcal{R}$, to insert an element in the correct bucket.

This redundancy which comes from storing elements in buckets as well as in the leaves of the SIST during the reconstruction 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 Sect. 5). Second, the elements of S_0 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 [2, 26]. 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 [26]. □

In conjunction with T, a sorted doubly 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. To achieve this, an arbitrary element x in L maintains a pointer to its truncated version $\widetilde{x}$, while $\widetilde{x}$ maintains a pointer to one among all the elements that are truncated to it (recall Definition 2). 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.

We now turn to the description of the maintenance of our data structure T by incrementally performing global reconstructions. The crucial property comes from Theorem 9 (Sect. 5), which dictates that each time the number of updates exceeds rn_0, where $0 < r < 1$, T must be reconstructed in order to always guarantee with high probability the size of the buckets. Theorem 9 guarantees that during at most rn_0 updates the size of all buckets is bounded in size with high probability. Thus, during this period of time, weak updates are performed on T. By applying Theorem 1, in a period of rn_0 updates we turn weak updates into normal updates.

Let us now discuss the implementation of Theorem 1 in our case. Assume that a new MAIN (the structure T consisting of a SIST with buckets) becomes available at time t, where time is defined with respect to update operations. Let n_0 be the number of elements in MAIN at time t. After $\frac{1}{2}n_0$ updates this MAIN will be turned to an OLD-MAIN and the incremental construction of a new MAIN is initiated. Assume that at this point (time $t+\frac{1}{2}n_0$) the MAIN that turns to OLD-MAIN has n_1 elements. After r_1n_1 update operations in total (at time $t+\frac{1}{2}n_0+r_1n_1$), MAIN will take over from OLD-MAIN (recall from Sect. 2.3 that any constant $r_1<\frac{1}{3}$ suffices, since in that case $rn_0=\frac{1}{2}n_0+r_1n_1<\frac{1}{2}n_0+\frac{1}{3}\cdot\frac{3}{2}n_0=n_0$, and thus $r<1$ as required). By accelerating updates, that is, by making a constant number of steps for constructing MAIN per each update operation (recall Sect. 2.3), we first construct MAIN on the n_1 elements. The update operations during the construction of MAIN are not taken into account and are put in a queue Q, while they are performed on OLD-MAIN as well as on L. Finally, after MAIN has been constructed on the n_1 elements, we start applying in an accelerating pace the updates in Q on MAIN until Q becomes empty at which point OLD-MAIN is discarded. In the following we elaborate on this procedure.

To begin with, the list L is augmented in order to facilitate the construction of MAIN by introducing additional fields to each node. Each node in L maintains a pointer to a record in a list L'—to be specified below—which is NIL if this is not applicable. If during the construction of MAIN an insertion operation of an element x is performed, then we insert x in L as well as a new entry in Q recording this insertion. We also establish a pointer from the entry corresponding to x in Q to L. If element x is deleted, then we add an entry to Q recording this deletion while the corresponding node in L is removed from the list but not destroyed. This node is called a *floating node* and has its successor and predecessor pointers set to NIL, while it maintains all other pointers. Note that this floating node may have been created by an insertion which is also recorded in Q.

Our first concern is how to start building MAIN on the n_1 elements at time $t+\frac{1}{2}n_0$, since L changes due to updates. To do that, we incrementally construct a new list L' starting from time t and ending at time $t+\frac{1}{2}n_0$, so that at the end both L and L' store the same set of elements. After time $t+\frac{1}{2}n_0$, L' does not change due to updates but exists only for the purpose of constructing the new MAIN.

To construct L', we traverse incrementally L and copy its elements to L'. That is, we initiate a pointer p_L to point to the head of L. After the update (insertion or deletion) to L at time $t+1$, we advance p_L so that the first 3 elements of L are copied to L'. For each element in L' there is a pointer to the respective element in L and vice-versa. After the insertion of x in L at time $t+i$, $2\le i\le\frac{1}{2}n_0$, we copy 3 elements of L by advancing p_L on L; if the successor of x (if we are at the end of the list we check the predecessor) has been copied to L' (by checking the corresponding pointer), then we also apply the insertion of x to L'. After the deletion of x from L at time $t+i$, $2\le i\le\frac{1}{2}n_0$, we again copy 3 elements of L by advancing p_L on L and then if x has been copied to L' (we can check this by the pointers between L and L') we also delete it from L'. After the update at time $t+\frac{1}{2}n_0$, both lists contain the same set of elements.

As soon as the current MAIN becomes OLD-MAIN at time $t+\frac{1}{2}n_0$, we proceed with the construction of the new MAIN based on the n_1 elements stored in L'. During the construction of q^*-heaps in MAIN by using the pointers between L' and L, we establish the necessary pointers between the truncated elements in the q^*-heaps and list L. Additionally, we maintain pointers from L' to the buckets (q^*-heaps) in MAIN that contain the corresponding truncated elements. These pointers facilitate the processing of $\mathcal{Q}$ as we see below. For each update in OLD-MAIN, which is added to $\mathcal{Q}$, some incremental work for the construction of MAIN on the elements of L' is performed. The minimum amount of incremental work per insertion or deletion is determined in Sect. 2.3. During the construction of q^*-heaps, we also construct the lower levels of the SIST (i.e., the SIST subtrees of height $O(H(|\mathcal{B}_i|))$ corresponding to the elements stored in the bucket $\mathcal{B}_i$) establishing pointers between its leaves and the corresponding elements in the buckets. Finally, when the construction of q^*-heaps is concluded we continue with the incremental construction of the higher levels of the SIST.

When the construction of MAIN on the n_1 elements is concluded, then all elements inside MAIN maintain pointers to list L or to some floating nodes that are to be deleted. Then, the update operations stored in $\mathcal{Q}$ are processed. For each new update that is also added to $\mathcal{Q}$, we process a constant number of updates already stored in this queue. Assume that the update extracted and processed from $\mathcal{Q}$ is the insertion of x, which has been already inserted in L. Then, the insertion algorithm described in Sect. 3.2.2 is applied with the exception that L needs not to be updated. In the case where the update from $\mathcal{Q}$ is a deletion of element x, then we follow the pointer to the floating node containing x. The algorithm for deletion described in Sect. 3.2.2 is applied to x with the exception that the node to be deleted is floating. As soon as $\mathcal{Q}$ gets empty and OLD-MAIN and MAIN contain the same set of elements, OLD-MAIN is discarded. Then the whole process restarts for the new MAIN.

3.2.2 Update and Search Operations

Having concluded the description of the data structure, we move to the discussion of the update and search operations supported by T. First, we discuss the weak 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 after a new MAIN takes over.

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 check whether any of the two adjacent elements of ψ in L point to the same truncated element $\widetilde{\psi}$. If this is the case, then we further check if the unique pointer of $\widetilde{\psi}$ to its chain points to ψ. If indeed, we update this pointer so that it now points to the adjacent element of ψ in the same chain. Finally, we remove ψ from L and terminate. This case takes $O(1)$ worst-case time, since we have to manipulate only a constant number of pointers. Otherwise (if none of the two adjacent elements of ψ in L point to the same truncated element), ψ is the only element of L pointing to $\widetilde{\psi}$ and thus we remove ψ (from L) as well as $\widetilde{\psi}$ from the respective bucket in $O(1)$ worst-case time, since the bucket is organized as a q^*-heap. This concludes the deletion operation that takes $O(1)$ worst-case time in total.

The *insertion* of a new element ψ follows a similar approach. 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. Let y' be the other adjacent element of ψ. We first check whether $\widetilde{y}$ or $\widetilde{y}'$, or both are equal to $\widetilde{\psi}$. If this is the case, then we create a pointer from ψ to the respective truncated element and terminate. Otherwise, we find the representative *rep*(i) of the bucket $\mathcal{B}_i$ in which $\widetilde{y}$ is stored. By using the list of representatives $\mathcal{R}$, we check whether $\widetilde{\psi}$ belongs to the bucket $\mathcal{B}_i$ or to some other bucket. This check is necessary, since if $\mathcal{B}_{i+1}$ is empty and $\widetilde{\psi}$ should be inserted in it due to the fact that $\widetilde{y}$ belongs to $\mathcal{B}_i$, we would erroneously insert $\widetilde{\psi}$ to bucket $\mathcal{B}_i$. Thus, if there is a sequence of such empty buckets (a very unlikely event due to Theorem 9), we may have to search $\mathcal{R}$ sequentially to locate the correct bucket. As soon as we locate the correct bucket, we insert $\widetilde{\psi}$ to the corresponding empty q^*-heap. Finally, a pointer is established from $\widetilde{\psi}$ to ψ. By Theorem 9, there is no sequence of empty buckets with high probability. Thus, since an insertion in a q^*-heap requires $O(1)$ worst-case time, we get that the insertion operation takes $O(1)$ time with high probability, due to the location of the correct bucket.

Recall that: (i) in both insertions and deletions the SIST remains unaffected, and a leaf of the SIST may correspond to an element which has been deleted, while an element in a bucket may not be stored in any leaf of the SIST; (ii) the leaves of the SIST define the range of values stored in the corresponding bucket and this is exactly why we need to use list $\mathcal{R}$, to ensure that an element is inserted in the correct bucket.

Now, we turn to the description of the predecessor *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 compares ψ with y in order to decide 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 $\widetilde{y}$ is followed in order to determine the bucket $\mathcal{B}_i$ in which $\widetilde{y}$ belongs.

Let $\mathcal{B}_{i+1}$ be the bucket to the right of $\mathcal{B}_i$. Three cases are considered, which can be distinguished by comparing ψ with the representatives of $\mathcal{B}_i$ and $\mathcal{B}_{i+1}$:

1. $\psi \le$ *rep*(i): In this case, we just retrieve from the q^*-heap that implements $\mathcal{B}_i$ the element $\widetilde{z}_0$ which is equal to the target element $\widetilde{\psi}$ or its predecessor in $O(1)$ worst-case time (Sect. 2.2). Note that (Sect. 3.1) there may be many real elements $z_0 \ne z_1 \ne \cdots \ne z_l$ stored in list L (possibly all not equal to the target ψ) with truncated value equal to $\widetilde{z}_0$ stored in the q^*-heap. According to Definition 2, these real elements z_i, $0 \le i \le l$, constitute the chain of $\widetilde{z}_0$ of length $l+1$. The element z_i can be located in L by following the pointer from $\widetilde{z}_0$ to list L and then making a sequential search during which each z_i, $0 \le i \le l$, is compared to target ψ and either a match is found or the largest element less than ψ is returned (predecessor) in $O(l)$ time (at most linear to chain length).
2. *rep*$(i) < \psi \le$ *rep*$(i+1)$: This means that element ψ belongs to bucket $\mathcal{B}_{i+1}$. Case 1 is applied for this bucket.
3. $\psi >$ *rep*$(i+1)$: The elements are stored in different buckets $\mathcal{B}_i$ and $\mathcal{B}_j$, $j \ne i+1$, containing $\widetilde{y}$ and $\widetilde{\psi}$ respectively. In this case, the search starts from *rep*(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 neighbor 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 (see Sect. 2.1, in particular equation (5)). When a leaf is reached, the pointer to its respective bucket $\mathcal{B}_j$ is followed and Case 1 is invoked for this bucket.

It follows from the above description that the time complexity of the search operation depends on the traversal of the internal nodes of the SIST, the searching time within a bucket (q^*-heap), and the expected length of the chain. Regarding the latter, observe that if the length of the chain of $\widetilde{\psi}$ is $\ell(\widetilde{\psi})$, then there will be an additive term of $O(\ell(\widetilde{\psi}))$ in the time complexity of the search operation. Thus, it would be best to bound the expected length of an arbitrary chain. We can provide such an expected bound as shown in the next section (Lemma 4) and also show that it is very unlikely that the chain length increases (Corollary 1) with respect to n.

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 an $(n \cdot g(H(n)), H^{-1}(H(n) - 1))$-smooth distribution.

Lemma 3 *The preprocessing time and the space usage of our data structure is $\Theta(n)$. Deletions are performed in $O(1)$ worst-case time, while insertions are performed in $O(1)$ time with high probability.*

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 bounds of the update operations follow from the discussion in Sect. 3.2.2, the results in [40, 41] (see also Sect. 2.2) as well as Theorem 1. In particular, since $P_T(n) = O(n)$ and $WD_T(n) = WI_T(n) = O(1)$, it follows by Theorem 1 that the insertion and deletion time is $O(1)$, the former with high probability while the latter in the worst-case. □

We now turn to the time complexity of the predecessor search operation. 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

As it was mentioned earlier (Sect. 3.2.2), the search time is affected (among others) by the expected length of the chain (Definition 2) of a truncated element, and before going into the details of our search time investigation, we provide a bound for this length. Note that the chain of a truncated element is initially created during the construction of SIST and it is subsequently affected only by update operations (as new elements are inserted or existing elements are deleted). Hence, the sought chain length should be provided with respect to update operations. This is precisely recorded by Lemma 4 below, which states that the length of a chain is $O(1)$ in expectation during the update operations.

This result is crucial in efficiently locating—via q^*-heaps—an arbitrary target real ψ. To see this, recall 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 $\widetilde{z}$ that coincides with or is the predecessor of our target element $\widetilde{\psi}$. Then, Lemma 4 implies that the chain of $\widetilde{z}$ has $O(1)$ expected length, which are real elements inserted during the rn_0 update operations. Therefore, by following the pointer from $\widetilde{z}$ towards one of these real elements among all of them that consist its chain, we can determine within $O(1)$ expected search time, if the real ψ appears in the linked list L and find its predecessor. Moreover, Lemma 4 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. Finally, an easy consequence of Lemma 4 is Corollary 1 showing that it is very unlikely that the chain length gets large.

Lemma 4 *Let the elements in our data structure belong to* $[a,b]\subset\mathbb{R}$ *and are drawn according to an* (f_1,f_2)*-smooth input distribution* μ, *where* $f_1(n)=\frac{n}{\log^{1+\epsilon}\log n}$ *and* $f_2(n)=n^{1-\delta}=n^{\alpha}$, $\alpha<1$. *Consider an arbitrary truncated element* $\widetilde{x}\in[a,b]$ *encoded with up to* $|\widetilde{x}|=O(\log n)$ *bits. Then, during each update operation, the expected chain length of* $\widetilde{x}\in[a,b]$ *is* $O(1)$.

Proof Let the elements in S_0 ordered increasingly as $x_1,x_2,\ldots,x_{n_0}$, 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 (f_1,f_2)-smooth input distribution μ, where (recall Remark 1) $f_1(n)=I(n)=\frac{n}{\log^{1+\epsilon}\log n}$ and $f_2(n)=n/R(n)=n^{\alpha}$, $\alpha<1$. For convenience, let $x_0=a$ and $x_{n_0+1}=b$.

According to Definition 1 and the discussion in Sect. 2.1, the interval $[a,b]$ is divided initially to $f_1(n)$ equally sized subintervals each of which gets at most $\frac{\beta\cdot f_2(n)}{n}$ mass probability and $\frac{\beta\cdot f_2(n)}{n}\times n=\beta\cdot f_2(n)=\beta n^{\alpha}$ elements in expectation. 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:

$$n^{\alpha^h}=C \quad\Longrightarrow\quad h=O(\log_{1/\alpha}\log n) \tag{13}$$

Note that in the 1-st recursion the number of subintervals 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 such subinterval 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 subintervals. In general, in the

$(i+1)$-th recursive division, each subinterval 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 subintervals.

We call the subintervals at the final h-th level (with h determined by (13) above) of recursion as *indivisible subintervals*. Notice that (13) implies that for each update operation, each such indivisible subinterval is expected to contain only $O(1)$ real elements stored in L. Thus, an arbitrary truncated element $\widetilde{x} \in [a,b]$ can be indexed in terms of these indivisible subintervals, while guaranteeing $O(1)$ expected chain length (Definition 2). It only remains to show that $O(\log n)$ bits suffice to encode this indexing via indivisible subintervals. But, this reduces in showing that the total number of indivisible subintervals is $n^{O(1)}$, which is proved below.

Taking into account (13), in the final level of recursion the total number of indivisible subintervals is

$$\begin{aligned}\prod_{i=0}^{h} f_1\bigl(n^{\alpha^i}\bigr) = \prod_{i=0}^{O(\log_{1/\alpha}\log n)} f_1\bigl(n^{\alpha^i}\bigr) &= \prod_{i=0}^{O(\log_{1/\alpha}\log n)} \frac{n^{\alpha^i}}{\log^{1+\epsilon}\log n^{\alpha^i}} \\ &< \prod_{i=0}^{O(\log_{1/\alpha}\log n)} n^{\alpha^i}\end{aligned} \tag{14}$$

It follows that the total number of bits needed to represent all these indivisible subintervals is at most

$$\log\left(\prod_{i=0}^{O(\log_{1/\alpha}\log n)} n^{\alpha^i}\right) \tag{15}$$

which is

$$\begin{aligned}\sum_{i=0}^{O(\log_{1/\alpha}\log n)} \log n^{\alpha^i} = \log n \sum_{i=0}^{O(\log_{1/\alpha}\log n)} \alpha^i &< \log n \sum_{i=0}^{\infty} \alpha^i \\ &= \log n \cdot \frac{1}{1-\alpha} = O(\log n)\end{aligned} \tag{16}$$

Thus, $O(\log n)$ bits are sufficient to represent all indivisible subintervals, and as a result of the aforementioned recursive division each such indivisible subrange contains $O(1)$ elements in expectation. □

The following corollary is an easy consequence of Lemma 4.

Corollary 1 *Let the elements in our data structure belong to* $[a,b] \subset \mathbb{R}$ *and are drawn according to an* (f_1, f_2)*-smooth input distribution* μ, *where* $f_1(n) = \frac{n}{\log^{1+\epsilon}\log n}$ *and* $f_2(n) = n^{1-\delta} = n^{\alpha}$, $\alpha < 1$. *Consider an arbitrary truncated element* $\widetilde{x} \in [a,b]$ *encoded with up to* $|\widetilde{x}| = O(\log n)$ *bits. Then, during each update operation, the chain length of* $\widetilde{x} \in [a,b]$ *is* $O(\phi(n))$ *with high probability, for any function* $\phi(n)$ *slowly growing with respect to* n.

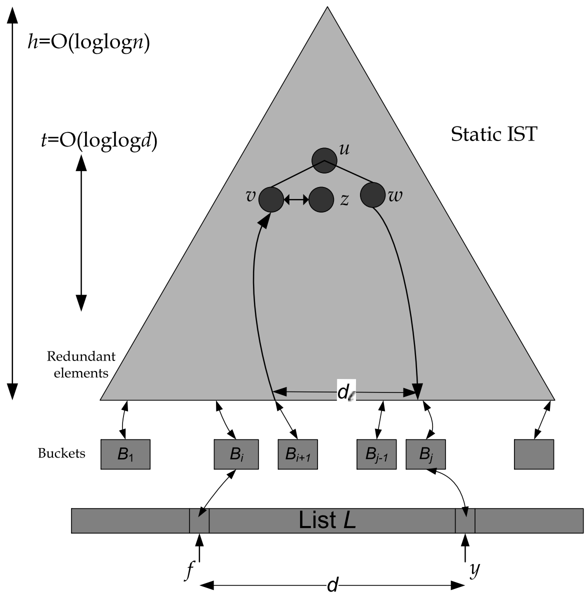

Fig. 1 An overview of the tree structure as well as the search path from a finger f to an element y

Proof It is a straightforward application of the Markov's inequality, since by Lemma 4 the expected chain length of any $\widetilde{x} \in [a,b]$ is $O(1)$. That is, if the random variable $\ell(\widetilde{x})$ is the chain length of $\widetilde{x}$ then $\Pr[\ell(\widetilde{x}) > \phi(n)] \leq \frac{\mathbb{E}[\ell(\widetilde{x})]}{\phi(n)} \to 0$, as $\phi(n) \to \infty$ with respect to n. □

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

Theorem 2 *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 $\widetilde{f}$ and $\widetilde{y}$ 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 time complexity of a search operation is equal to*: (a) $O(\frac{\log|\mathcal{B}_i|}{\log\log n} + \frac{\log|\mathcal{B}_j|}{\log\log n} + \log\log d)$ *in expectation*; (b) $O(\frac{\log|\mathcal{B}_i|}{\log\log n} + \frac{\log|\mathcal{B}_j|}{\log\log n} + \log\log d + \phi(n))$ *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 Fig. 1): (i) the search for $\widetilde{y}$ in the q^*-heap implementing $\mathcal{B}_i$ or its adjacent bucket $\mathcal{B}_{i+1}$, (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 $\widetilde{y}$, (iii) the search for $\widetilde{y}$ in the q^*-heap implementing $\mathcal{B}_j$ and (iv) the search in L for y.

From the results in [40, 41] (see also Sect. 2.2), the time complexity for the execution of steps (i) and (iii) is equal to $O(\frac{\log|\mathcal{B}_i|}{\log\log n_0} + \frac{\log|\mathcal{B}_j|}{\log\log n_0})$, 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 $O(\frac{\log|\mathcal{B}_i|}{\log\log n}+\frac{\log|\mathcal{B}_j|}{\log\log n})$. Once locating $\widetilde{y}$ in $\mathcal{B}_j$, we get from Lemma 4 and Corollary 1 that step (iv) can be accomplished in $O(1)$ expected time, or in $O(\phi(n))$ time with high probability.

Step (ii) introduces the distance d (number of elements between f and y in L) in the time complexity. To begin with, if f and y lie in the same bucket $\mathcal{B}_i$ or in adjacent buckets then the first two cases in the description of the search procedure (Sect. 3.2) guarantee that in an $O(1)$ number of steps we will have identified this case. Assume, that this is not the case. This means that the minimum distance between f and y is $\Theta(\log n)$ with high probability (see Theorem 9) since there is at least one bucket whose elements lie between f and y in list L.

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 (see 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 leaves of the subtrees rooted at nodes u and z and let t be the height of u. From Remark 2 and Lemma 2, $t=O(\log\log u_\ell)$. Finally, let $d_{\mathcal{B}}$ be the distance between f and y in terms of number of buckets and let d_ℓ be the number of leaves between $rep(i)$ and $rep(j)$. Apparently, $\log n\le z_\ell<u_\ell$ with high probability, since at least one bucket is between f and y. Additionally, it holds that $z_\ell\le d_\ell\le u_\ell$. With respect to distance we get that $d_\ell=\Theta(d_{\mathcal{B}}\log n)$ with high probability, and $d=\Theta(d_{\mathcal{B}}\log n)$ with high probability by Theorem 9. As a result, $d=\Theta(d_\ell)$ with high probability, meaning that with high probability the distance measured in term of leaves of the SIST is asymptotically equal to the real distance between f and y with respect to list L.

Since $t=O(\log\log u_\ell)$ and $z_\ell=(u_\ell)^\delta$, we conclude that $(O(2^{2^t}))^\delta\le d_\ell\le O(2^{2^t})$. Since the exponent δ, when considered in the double logarithmic time complexity, becomes an additive term and $d=\Theta(d_\ell)$ 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 prove (by exploiting the probabilistic analysis in [26]) 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 a leaf of SIST. Let $v_1,\dots,v_t$ be the nodes in the path listed in order of visit and consider a node v_i arbitrarily 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, $n_i\ge\log n$. In [26, 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 p steps is bounded from above by $(\frac{c}{p})^{p\sqrt{n_i}}$. 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 p steps is bounded from above by $(\frac{c}{p})^{p\cdot n_i^{1-\delta}}$. For $p=2c$ 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 the interpolation takes more than $2c$ steps. Then, it follows that $q\le\sum_{i=1}^{t}(\frac{1}{2})^{2cn_i^{1-\delta}}\le t(\frac{1}{2})^{2c(\log n)^{1-\delta}}$. Hence, for the probability q' that the descent phase takes less than $2ct$ steps we have $q'\ge 1-t(\frac{1}{2})^{2c(\log n)^{1-\delta}}$. Since $t=O(\log\log d)=O(\log\log n)$ we get that $t(\frac{1}{2})^{2c(\log n)^{1-\delta}}\to 0$, as n grows, and thus we conclude that the descent phase in step (ii) takes $2ct = O(\log\log d)$ 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 (Sect. 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 9). In particular, we show that $|\mathcal{B}_i| = \Theta(\log n)$ with high probability for a bucket $\mathcal{B}_i$. Plugging this into Theorem 2 and taking into account Lemma 3, we get the main result of the paper.

Theorem 3 *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 a search time of*: (i) $O(\log\log d)$ *in expectation*; (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 $\Theta(n)$. Deletions are performed in $O(1)$ worst-case time, while insertions are performed in $O(1)$ time with high probability.*

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 $0 < \delta < 1$. In this section, we show that 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 [2], where $\sum_{i=1}^{\infty} g(i) = \Theta(1)$, and $H(n)$ is as defined in Sect. 2, thus being able to achieve $o(\log\log d)$ expected time complexity for several distributions.

As it is proved in [2], 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 [2].

We first observe that the results of Lemma 4 and Corollary 1 carry over to the general class of $(n \cdot g(H(n)), H^{-1}(H(n) - 1))$-smooth densities. In particular, by working as previously in the proof of Lemma 4 but for this general class of input distributions, we get that the total number of indivisible subintervals is

$$\prod_{i=0}^{h} f_1(n_i) = \prod_{i=0}^{h} n_i \cdot g\bigl(H(n_i)\bigr)$$

The total number of bits needed to represent all these indivisible subintervals is at most

$$\log\left(\prod_{i=0}^{h} n_i \cdot g\bigl(H(n_i)\bigr)\right) = \sum_{i=0}^{h} \log\bigl(n_i \cdot g\bigl(H(n_i)\bigr)\bigr) < \log n \sum_{i=0}^{h} g\bigl(H(n_i)\bigr) = O(\log n)$$

where the last equality follows from the fact that $\sum_{i=0}^{h} g(H(n_i)) < \sum_{i=1}^{\infty} g(i) = \Theta(1)$, because $H(n_{i+1}) = H(H^{-1}(H(n_i) - 1)) = H(n_i) - 1$. Hence, we have proved that Lemma 4 carries over to the general class of smooth densities, and consequently the same applies for Corollary 1. We record this fact in the next lemma.

Lemma 5 *Let the elements in our data structure belong to $[a,b] \subset \mathbb{R}$ and are drawn according to an (f_1, f_2)-smooth input distribution μ, where $f_1(n) = I(n) = n \cdot g(H(n))$ and $f_2(n) = n/R(n) = H^{-1}(H(n) - 1))$, where $\sum_{i=1}^{\infty} g(i) = \Theta(1)$, $H(n) : \mathbb{N} \to \mathbb{R}_0^+$ 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 $1 \leq i \leq H(n) - 1$. Consider an arbitrary truncated element $\widetilde{x} \in [a,b]$ encoded with up to $|\widetilde{x}| = O(\log n)$ bits. Then, during each update operation*: (a) *the expected chain length of $\widetilde{x} \in [a,b]$ is $O(1)$*; (b) *the chain length of $\widetilde{x} \in [a,b]$ is $O(\phi(n))$ with high probability, for any function $\phi(n)$ slowly growing with respect to n.*

By examining the proof of Theorem 2, we can see that the specific choice of the class of smooth densities comes into play when analyzing steps (ii) and (iv) of the search procedure in T. Step (iv) concerns the location of the search element y in L, whose time is dominated by the length of the chain of $\widetilde{y}$ and is given by Lemma 5. Hence, to provide a search time bound for the aforementioned general class of smooth densities, it remains to provide a bound for step (ii).

Let t denote the time complexity of step (ii). We can generalize the proof of Theorem 2 by applying the following argument: let $h(z)$ be the height of z and $h(u)$ be the height of u. Let z_ℓ be the number of leaves in the subtree rooted at z and let u_ℓ be the number of leaves in the subtree rooted at u. Then, the following hold: (a) $t = \Theta(h(u))$; (b) $d_\ell = \Theta(d)$ with high probability; (c) $z_\ell \leq d_\ell \leq u_\ell \Rightarrow h(z) \leq H(d_\ell) \leq h(u)$; and (d) $h(u) = h(z) + 1$. From (a) and (d) we get that $t = O(h(z))$, and from (b) and (c) we get $t = O(H(d))$. The above discussion establishes the following theorem.

Theorem 4 *Let μ be a $(n \cdot g(H(n)), H^{-1}(H(n) - 1))$-smooth density, where $\sum_{i=1}^{\infty} g(i) = \Theta(1)$, $H(n) : \mathbb{N} \to \mathbb{R}_0^+$ 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 a search time of*: (i) $\Theta(H(d))$ *in expectation*; (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 $\Theta(n)$. Deletions are performed in $O(1)$ worst-case time, while insertions are performed in $O(1)$ time with high probability.*

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 [2] gives an expected search time complexity of $\Theta(\log^* n)$. However, by applying Theorem 4, 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 [2]. These concern the two extreme cases of $H(n)$, $H(1) = \Theta(1)$ and

$H(n) = \Theta(\log n)$, for which the corresponding distributions may be zero in an interval. The former case is the class of $(n, 1)$-smooth densities, which is equivalent to the class of bounded densities, and for which our approach clearly achieves an expected $\Theta(1)$ search time. The latter case is the class of $(n \cdot g(\Theta(\log n)), \Theta(n))$-smooth densities that contains all densities, and for which our approach achieves an expected $\Theta(\log n)$ search time. That is, for these two extreme cases our approach achieves the same results as the method in [2].

4.3 Worst-Case Guarantees

The data structure for the predecessor problem presented here provides bounds with high probability when certain assumptions hold, the strictest being the assumption that the elements are generated by the same smooth distribution. If this assumption does not hold, then our structure fails to provide any guarantees. In order to alleviate this problem, we follow the standard approach of maintaining two data structures back-to-back. We employ a data structure W for the predecessor problem that guarantees worst-case time bounds, while the T structure guarantees expected time complexities. Structure W can be any worst-case constant update finger search data structure, e.g., like the ones in [7] or [4].

The structure T is attached a flag *active* denoting whether this structure is valid subject to searches and updates, or invalid. Thus, when *active* is TRUE both structures T and W maintain the same set of elements. Update operations are performed on both structures in worst-case constant time, since T is valid as long as there are no sequences of empty buckets as shown below. Predecessor queries are performed in alternating steps between W and T and the structure that first concludes the query returns the answer. When *active* is FALSE, then only W is valid and it is the only structure that serves queries and updates while the new T structure is constructed incrementally according to Theorem 1.

Initially or at the end of every reconstruction, T is valid (*active* is set to TRUE). T becomes invalid in two different cases:

1. When the size of any bucket during update operations is not in the range $[\frac{1}{c}\ln n, c\ln n]$, for some appropriately chosen constant $c > 2$ defined by the quantity $\kappa(t)$ in the proof of Theorem 8. This may be the result of a concept drift (i.e., a change in the distribution generating the data) or simply an unfortunate (with very small probability) sequence of update operations. This criterion is applied to OLD-MAIN and to MAIN as well, provided that MAIN is in the phase of processing the updates in $\mathcal{Q}$. We signal this event by setting *active* to FALSE.
2. When the search in the SIST takes a lot of time. This can happen when the distribution is not $(n/(\log\log n)^{1+\varepsilon}, n^{1-\delta})$-smooth. We signal this event, setting *active* to FALSE, when in a constant-size sequence of search operations it is the case that W always concludes first.

By making use of the results in [4] and using Theorem 3 to implement T, the above discussion provides the final result of this paper which is summarized in the following theorem.

Theorem 5 *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 a search time of*: (i) $O(\log\log d)$ *in expectation*; (ii) $O(\log\log d + \phi(n))$ *with high probability*; (iii) $O(\sqrt{\frac{\log d}{\log\log d}})$ *in the worst-case. 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 $O(1)$.*

The same reasoning can also be applied to the general class of $(n \cdot g(H(n)), H^{-1}(H(n) - 1))$-smooth distributions (Sect. 4.2), in which case T is implemented using Theorem 4. In this case, the worst-case bounds stated in Theorem 5 remain unaffected, and only the search time becomes $\Theta(H(d))$ in expectation and $\Theta(H(d) + \phi(n))$ with high probability.

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 Sect. 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 $[rep(0), rep(1)] \cup (rep(1), rep(2)] \cup \cdots \cup (rep(\rho - 1), rep(\rho)]$, where $rep(0) = a$, $rep(\rho) = b$, and $\forall i = 1, \ldots, \rho - 1$, the elements $rep(i) \in [a, b]$ are those defined in Sect. 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 $rep(i - 1) < x \le rep(i)$, $i = 2, \ldots, \rho$, otherwise x is inserted in (deleted from) $\mathcal{B}_1$.

5.1 Almost Uniform Bins

Our aim is to prove that with high probability the maximum load of any bin is $O(\ln n)$, and that no bin remains empty as $n \to \infty$. If we knew the distribution $\mu(\cdot)$, then we could partition the interval $[a, b]$ into $\rho = \frac{n}{\ln n}$ distinct bins (parts), $[rep_\mu(0), rep_\mu(1)] \cup (rep_\mu(1), rep_\mu(2)] \cup \cdots \cup (rep_\mu(\rho - 1), rep_\mu(\rho)]$, 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:

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

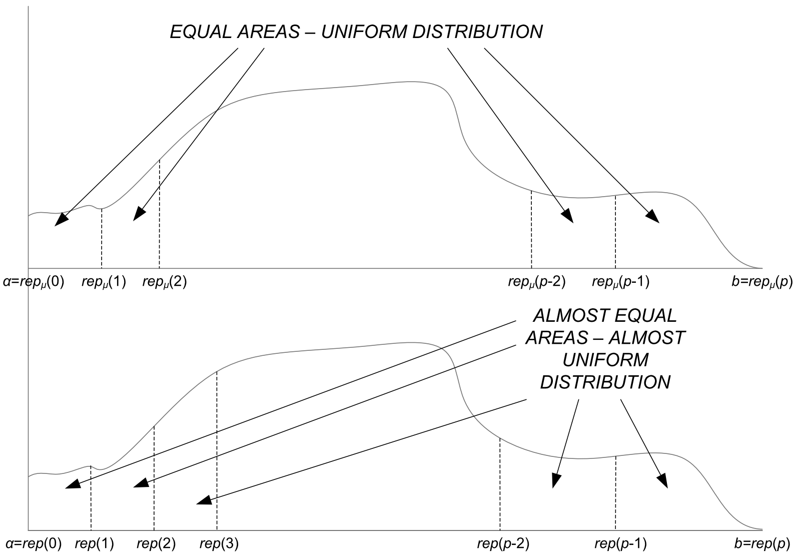

Fig. 2 Plot of an unknown probability density $\mu(x)$, $x \in [a,b]$. The upper graphic represents the uniform bins defined by: $rep_\mu(0), rep_\mu(1), \dots, rep_\mu(\rho)$. The lower graphic represents the *almost* uniform bins defined by: $rep(0), rep(1), \dots, rep(\rho)$

Remark 3 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:

$$\Pr\bigl[x \in \bigl(rep(i-1), rep(i)\bigr]\bigr]' = \int_{rep(i-1)}^{rep(i)} \mu(t)\,dt = \Theta\left(\frac{1}{\rho}\right) = \Theta\left(\frac{\ln n}{n}\right), \quad i = 1, \dots, \rho$$

Remark 4 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 3 and 4 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 5 The above intuitive argument does not imply that $\lim_{n\to\infty} rep(i) = rep_\mu(i)$, $\forall i=0,\dots,\rho$. 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 $\Theta(\frac{1}{\rho}) = \Theta(\frac{\ln n}{n})$, regardless of the particular distribution density $\mu(\cdot)$.

Theorem 6 *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*:

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

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

$$\binom{n}{\alpha(n)n} q(n)^{\alpha(n)n}\bigl(1-q(n)\bigr)^{(1-\alpha(n))n} \sim \left[\left(\frac{q(n)}{\alpha(n)}\right)^{\alpha(n)}\left(\frac{1-q(n)}{1-\alpha(n)}\right)^{1-\alpha(n)}\right]^n \tag{17}$$

where the expression on the right is asymptotically equal to the expression on the left if we use Stirling's approximation $n!\sim(\frac{n}{e})^n\sqrt{2\pi n}$ and ignore inverse polynomial multiplicative terms. Expression (17) 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 subinterval of $[a, b]$ axis of measure $q(n)$ is at most

$$\frac{n}{\ln n} \cdot \left[\left(\frac{q(n)}{\alpha(n)} \right)^{\alpha(n)} \left(\frac{1 - q(n)}{1 - \alpha(n)} \right)^{1-\alpha(n)} \right]^n \to 0 \tag{18}$$

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 6 imply the following.

Corollary 2 *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 3 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 3) 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 2 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 4 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 2) 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 2, 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 2, 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, $t=\frac{j}{n}$, $m(t)=\frac{M(tn)}{n}$, with range $(t,m(t)) \in [0,r] \times [1,m(r)]$. The following theorem provides an estimation on $m(t)$.

Theorem 7 *For each operation* $0 \le t \le r$*, 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 $\gamma^+ tn$, $\gamma^+ \in [0,1]$, denote the current fraction of insertion operations among the currently performed tn operations, and let $\gamma^- tn$ 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 $\gamma^+ tn$ of insertions minus the number $\gamma^- tn$ 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 $\gamma^+ tn$. Since in each of the tn operations a ball is inserted with probability p, the random variable $\gamma^+ tn$ of currently inserted balls obeys the binomial $B(tn, p)$ distribution. As a result, the random variable $\gamma^+ tn$ is sharply concentrated to its expected value ptn. That is, $\gamma^+ tn \to ptn$ with high probability, as $n \to \infty$. Equivalently, $\gamma^+ \to p$ with high probability, as $n \to \infty$. □

Remark 6 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 6 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 8.

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 [10]). 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 8

(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, $E[X(t)] \to 0$.*

(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 $\kappa \geq (-Cm(t) + 2)/(C \cdot LW(-\frac{Cm(t)-2}{Cm(t)e})) = O(1)$, and C is the positive constant defined in Definition 3. If $p > 1/2$, then for each operation t, $E[Z_\kappa(t)] \to 0$.*

Proof (i) Recall the definitions of the positive constants c and C (Definition 3, at the end of Sect. 5.1). From Lemmata 6 and 7, $\forall i = 1, \dots, \rho = \frac{n}{\ln(n)}$, it holds:

$$\Pr\bigl[Y_i(t) = 0\bigr] \leq \left(1 - c\frac{\ln(n)}{n}\right)^{m(t)n} \sim e^{-cm(t)\ln(n)} = \frac{1}{n^{cm(t)}} \tag{19}$$

From (19), by linearity of expectation, we obtain:

$$E\big[X(t)\mid m(t)\big]\le\sum_{i=1}^{\rho}\Pr\big[Y_i(t)=0\big]\le\frac{n}{\ln(n)}\cdot\frac{1}{n^{cm(t)}}\tag{20}$$

From Theorem 7 and Remark 6 it holds:

$$\forall t\ge 0,\quad \frac{1}{n^{cm(t)}}\le\frac{1}{n^{cm(0)}}=\frac{1}{n}$$

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 7, (20) becomes,

$$E\big[X(0)\mid m(0)\big]\le\frac{n}{\ln(n)}\cdot\frac{1}{n^{cm(0)}}=\frac{n}{\ln(n)}\cdot\frac{1}{n}=\frac{1}{\ln(n)}\to 0$$

Finally, from Markov's inequality, we obtain

$$\Pr\big[X(t)>0\mid m(t)\big]\le E\big[X(t)\mid m(t)\big]\le E\big[X(0)\mid m(0)\big]\to 0$$

(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 $\mathcal{B}_i$ contains $Y_i(t)=\Theta(m(t)\ln(n))$ balls in expectation, $i=1,\dots,\rho$. 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 $Y_i(t)\ge\kappa\ln(n)>m(t)\ln(n)$ is exponentially small, for $\kappa\ge-\frac{Cm(t)-2}{C\cdot LW(-\frac{Cm(t)-2}{Cm(t)e})}$. It suffices to prove this for $\mathcal{B}_{i'}$ which is the most likely to receive balls. To this end,

$$\Pr\big[Y_{i'}(t)\ge\kappa\ln(n)\mid m(t)\big]=\sum_{j=\kappa\ln(n)}^{m(t)n}\Pr\big[Y_{i'}(t)=j\mid m(t)\big]\tag{21}$$

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:

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

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

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

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

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

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 $m(t)C\ln(n)$, (21) becomes:

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

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:

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

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

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

The part (ii) in the proof of Theorem 8 can be straightforwardly adapted to show that with high probability no bin receives $o(\log n)$ balls. This remark along with Theorem 8 establish the following result.

Theorem 9 *Consider the aforementioned random process of n balls and $n/\ln n$ bins modeling the update operations in our data structure T, and where during each operation $j = 0, \dots, rn$ (r constant) with probability $p > 1/2$ a μ-random ball $x \in [a, b]$ is inserted into an appropriate bin of T and with probability $1 - p$ an existing ball is deleted (uniformly at random) from the current number of $M(j)$ balls that are stored in T. Then, with high probability, there is no sequence of empty bins and each bin receives $\Theta(\log n)$ balls.*

6 Conclusions

In this paper we presented a new finger search tree with $O(1)$ update time and linear space that supports finger searching queries in $O(\log\log d)$ expected time, or in $O(\log\log d + \phi(n))$ 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 $O(1)$ update time and expected search time of $O(H(d))$, or search time of $O(H(d)+\phi(n))$ 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 $o(\log\log d) + O(\phi(n))$ search time with high probability. Our result is an improvement over the general searching problem considered in [2], since we can achieve better search bounds with high probability.

Since the techniques of Sect. 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.

Acknowledgements We are indebted to the anonymous referees for their valuable comments that improved considerably the presentation of the paper, as well as for their constant insistence and patience which helped us to resolve several subtle points and thus improving the quality of the paper. We would also like to thank the editor for his persistence in clarifying various involved issues and for his guidance during the whole revision process.

References

1. Aho, A.V., Hopcroft, J.E., Ullman, J.D.: The Design and Analysis of Computer Algorithms. Addison-Wesley, Reading (1974)
2. Andersson, A., Mattsson, C.: Dynamic interpolation search in $o(\log\log n)$ time. In: Proc. 20th Coll. on Automata, Languages and Programming—ICALP'93, Lecture Notes Comput. Sci., vol. 700, pp. 15–27. Springer, Berlin (1993)
3. Anderson, A., Thorup, M.: 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, New York (2000). See also http://arxiv.org/abs/cs.DS/0210006
4. Anderson, A., Thorup, M.: Dynamic ordered sets with exponential search trees. J. ACM **54**(3), 1–40 (2007). Article 13
5. Atallah, M.J., Goodrich, M., Ramaiyer, K.: Biased finger trees and three-dimensional layers of maxima. In: Proc. 10th ACM Symposium on Computational Geometry, pp. 150–159 (1994)
6. Beame, P., Fich, F.: Optimal bounds for the predecessor problem and related problems. J. Comput. Syst. Sci. **65**(1), 38–72 (2002)

7. Brodal, G.S., Lagogiannis, G., Makris, C., Tsakalidis, A., Tsichlas, K.: Optimal finger search trees in the pointer machine. J. Comput. Syst. Sci. **67**, 381–418 (2003). Preliminary version in Proc. 34th ACM Symposium on Theory of Computing—STOC 2002, pp. 583–592 (2002)
8. Cole, R., Frieze, A., Maggs, B., Mitzenmacher, M., Richa, A., Sitaraman, R., Upfal, E.: On balls and bins with deletions. In: Randomization and Approximation Techniques in Computer Science—RANDOM'98. Lecture Notes Comput. Sci., vol. 1518, pp. 145–158. Springer, Berlin (1998)
9. Cook, S.A., Reckhow, R.A.: Time bounded random access machines. J. Comput. Syst. Sci. **7**, 354–375 (1973)
10. Corless, R.M., Gonnet, G.H., Hare, D.E.G., Jeffrey, D.J., Knuth, D.E.: On the Lambert W function. Adv. Comput. Math. **5**, 329–359 (1996)
11. Dietz, P., Raman, R.: A constant update time finger search tree. Inf. Process. Lett. **52**, 147–154 (1994)
12. Frederickson, G.: Implicit data structures for the dictionary problem. J. ACM **30**(1), 80–94 (1983)
13. Fredman, M.L., Willard, D.E.: Surpassing the information theoretic bound with fusion trees. J. Comput. Syst. Sci. **47**, 424–436 (1993)
14. Gonnet, G.: Interpolation and interpolation-hash searching. PhD Thesis, University of Waterloo, Waterloo (1977)
15. Gonnet, G., Rogers, L., George, J.: An algorithmic and complexity analysis of interpolation search. Acta Inform. **13**(1), 39–52 (1980)
16. Guibas, L., Hershberger, J., Leven, D., Sharir, M., Tarjan, R.E.: Linear time algorithms for visibility and shortest path problems inside simple polygons. Algorithmica **2**, 209–233 (1987)
17. Guibas, L., McCreight, E., Plass, M., Roberts, J.: A new representation for linear lists. In: Proc. 9th Annual ACM Symposium on Theory of Computing—STOC'77, pp. 49–60 (1977)
18. Hagerup, T.: Sorting and searching on the word RAM. In: Theoretical Aspects of Computer Science—STACS'98. Lecture Notes Comput. Sci., vol. 1373, pp. 366–398. Springer, Berlin (1998)
19. Hershberger, J.: 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)
20. Hoffman, K., Mehlhorn, K., Rosenstiehl, P., Tarjan, R.E.: Sorting Jordan sequences in linear time using level-linked search trees. Inf. Control **68**(1–3), 170–184 (1986)
21. Itai, A., Konheim, A., Rodeh, M.: A sparse table implementation of priority queues. In: Proc. ICALP'81, Lecture Notes Comput. Sci., vol. 115, pp. 417–431. Springer, Berlin (1981)
22. Kaporis, A., Makris, C., Sioutas, S., Tsakalidis, A., Tsichlas, K., Zaroliagis, C.: Improved bounds for finger search on a RAM. In: Algorithms—ESA 2003. Lecture Notes Comput. Sci., vol. 2832, pp. 325–336. Springer, Berlin (2003)
23. Knuth, D.E.: Deletions that preserve randomness. IEEE Trans. Softw. Eng. **3**, 351–359 (1977)
24. Knuth, D.E.: The Art of Computer Programming: Sorting and Searching. Addison-Wesley, Reading (1997)
25. Mehlhorn, K., Tsakalidis, A.: Data structures. In Handbook of Theoretical Computer Science—Vol. I: Algorithms and Complexity, pp. 303–341. MIT Press, Cambridge (1990). Chap. 6
26. Mehlhorn, K., Tsakalidis, A.: Dynamic interpolation search. J. ACM **40**(3), 621–634 (1993)
27. Overmars, M., Leeuwen, J.: Worst case optimal insertion and deletion methods for decomposable searching problems. Inf. Process. Lett. **12**(4), 168–173 (1981)
28. Pearl, Y., Itai, A., Avni, H.: Interpolation search—a $\log\log N$ search. Commun. ACM **21**(7), 550–554 (1978)
29. Perl, Y., Reingold, E.M.: Understanding the complexity of the interpolation search. Inf. Process. Lett. **6**(6), 219–222 (1977)
30. Peterson, W.W.: Addressing for random storage. IBM J. Res. Dev. **1**(4), 130–146 (1957)
31. Preparata, F., Shamos, M.: Computational Geometry. Springer, Berlin (1985)
32. Schönhage, Arnold: On the power of random access machines. In Proc. 6th Int. Colloq. on Automata, Languages and Programmng—ICALP'79. Lecture Notes Comput. Sci., vol. 71, pp. 520–529. Springer, Berlin (1979)
33. Seidel, R., Aragon, C.R.: Randomized search trees. Algorithmica **16**, 464–497 (1996)
34. Supowit, K.J., Reingold, E.M.: Divide and conquer heuristics for minimum weighted Euclidean matching. SIAM J. Comput. **12**(1), 118–143 (1983)
35. Tarjan, R.E.: Efficiency of a good but not linear set-union algorithm. J. ACM **22**, 215–225 (1975)
36. Tarjan, R.E.: A class of algorithms which require nonlinear time to maintain disjoint sets. J. Comput. Syst. Sci. **18**(2), 110–127 (1979)
37. Tarjan, R.E.: Data Structures and Network Algorithms. SIAM, Philadelphia (1983)
38. Thorup, M.: On RAM priority queues. SIAM J. Comput. **30**(1), 86–109 (2000)

39. Willard, D.E.: Searching unindexed and nonuniformly generated files in $\log\log N$ time. SIAM J. Comput. **14**(4), 1013–1029 (1985)
40. Willard, D.E.: 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)
41. Willard, D.E.: Examining computational geometry, van Emde Boas trees, and hashing from the perspective of the fusion tree. SIAM J. Comput. **29**(3), 1030–1049 (2000)
42. Yao, A.C., Yao, F.F.: The complexity of searching an ordered random table. In: Proc. 17th IEEE Symp. on Foundations of Computer Science—FOCS'76, pp. 173–177 (1976)