Information and Computation ••• (••••) ••••••

Contents lists available at ScienceDirect

Information and Computation

www.elsevier.com/locate/yinco

Dynamic Interpolation Search revisited ☆

Alexis Kaporis [a], Christos Makris [b], Spyros Sioutas [b], Athanasios Tsakalidis [b], Kostas Tsichlas [d], Christos Zaroliagis [b,c,*]

[a] Department of Information & Communication Systems Engineering, University of the Aegean, Karlovassi, Samos, 83200, Greece
[b] Department of Computer Engineering and Informatics, University of Patras, 26500 Patras, Greece
[c] Computer Technology Institute & Press "Diophantus", N. Kazantzaki Str, Patras University Campus, 26504 Patras, Greece
[d] Department of Informatics, Aristotle University of Thessaloniki, Greece

ARTICLE INFO

Article history:
Received 11 June 2018
Received in revised form 19 February 2019
Accepted 15 March 2019
Available online xxxx

Keywords:
Interpolation Search
Dynamic predecessor search
Dynamic search data structure

ABSTRACT

A new dynamic Interpolation Search (IS) data structure is presented that achieves $O(\log\log n)$ search time with high probability on unknown continuous or even discrete input distributions with measurable probability of element collisions, including power law and Binomial distributions. No such previous result holds for IS when the probability of element collisions is measurable. Moreover, our data structure exhibits $O(1)$ search time with high probability (w.h.p.) for a wide class of input distributions that contains all those for which $o(\log\log n)$ expected search time was previously known.

1. Introduction

The dynamic predecessor search problem is one of the fundamental problems in computer science. In this problem we have to maintain a set of elements subject to insertions and deletions such that given a query element y we can retrieve the largest element in the set smaller or equal to y. Well known search methods use an arbitrary rule to *select* a splitting element and *split* the stored set into two subsets; in binary search, each recursive split selects as splitting element, in a "blind" manner, the middle (or an element close to the middle) element of the current set. Using this technique, several results for the dynamic predecessor search problem have been achieved on the Random Access Machine (RAM) and the Pointer Machine (PM) models of computation. Before discussing the results, we review these models and their variants.

1.1. Models of computation

Our discussion in this section follows the exposition in [29]. A Random Access Machine (RAM) [2,11,12,46,52] 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)

☆ This work was partially supported by the FET Unit of EC under contracts no. FP6-021235-2 (FP6 IST/FET-Open/Project ARRIVAL) and no. ICT-215270 (FP7 ICT/FET-Proactive/Project FRONTS), and by the Action PYTHAGORAS with matching funds from the EU Social Fund and the Greek Ministry of Education.
* Corresponding author at: Department of Computer Engineering and Informatics, University of Patras, 26500 Patras, Greece.
E-mail addresses: kaporisa@aegean.gr (A. Kaporis), makri@ceid.upatras.gr (C. Makris), sioutas@ceid.upatras.gr (S. Sioutas), tsak@ceid.upatras.gr (A. Tsakalidis), tsichlas@csd.auth.gr (K. Tsichlas), zaro@ceid.upatras.gr (C. Zaroliagis).

https://doi.org/10.1016/j.ic.2019.104465
0890-5401/

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 for preventing this is to set a limit on the size of representable integers and to restrict the operations allowed on reals [11,12,52]. 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 (with logarithmic word size) RAM variant is the so-called *unit-cost real RAM* [42,47] 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 [44]). In [29] we have proved that each real number can be represented by its truncated version, up to a sufficiently large precision, and $O(\log n)$ bits suffice to represent the truncated version.

Yet another variant of the unit-cost RAM is the so-called *word RAM* [20,24]. 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 [5,6,24,53,58,59], as for example numbers following the IEEE 754 floating-point standard. In particular, floating point numerical values are typically either a word or a multiple of a word. 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) [50,52] 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 [50].

Differences between the models and all related solutions on predecessor queries are presented in the following section.

1.2. Previous work and motivation

The arbitrary rule technique for choosing the splitting element has guided a host of approaches for solving the dynamic predecessor search problem in the PM and the RAM models of computation. In the unit-cost real RAM model, known balanced search trees like AVL-trees [1], red-black trees [51] and (a, b)-trees [25], support search and update operations in $O(\log n)$ time when storing n elements. For comparison-based algorithms, the search time cannot be further reduced, since the lower bound of $\Omega(n \log n)$ for sorting n elements would be violated. In the word RAM model, there are a few data structures for predecessor queries. Using the van Emde Boas data structure [55], an $O(\log w)$ time per query and $\Theta(|\mathcal{U}|)$ space, where $\mathcal{U} = 1, \ldots, 2^w$ is the universe, can be achieved. By combining hashing with the van Emde Boas data structure the space is reduced to $\Theta(n)$, while the time per query is $O(\log w)$ with high probability. Y-fast tries [56] also achieve the same bounds. Fusion trees [20] can achieve $O(\log_w(n))$ time with high probability, and $\Theta(n)$ space. Note that the query time of a fusion tree may or may not be better than that of a Y-fast trie, depending on the values of w and n. If we chose the optimal data structure depending on these values, then it turns out by a small calculation that the query time becomes $O(\sqrt{\log n})$ time with high probability, with $\Theta(n)$ space. A lower bound of $\Omega(\sqrt{\frac{\log n}{\log \log n}})$ was proved by Beame and Fich [7]; a data structure achieving this time performance has been presented by Andersson and Thorup [5,6]. Finally, Pătraşcu and Thorup [37] prove a separation between near-linear and polynomial space and at the same time they provide matching upper bounds for a static set of integers for the word RAM.

In the remainder of the paper and in accordance with previous work, all results that will be discussed concern the unit-cost real RAM model, unless explicitly stated otherwise.

The aforementioned lower bounds can be surpassed if we take into account the input distribution of the elements and consider expected complexities; in this case, the extra knowledge about the probabilistic nature of the elements stored in the set may lead to better selections of splitting elements.

The main representative of these techniques is the method of *Interpolation Search* (IS) introduced by Peterson [41], where the splitting element was selected close to the expected location of the target element. In this method the splitting element is selected by taking advantage of the statistical properties of the elements stored in the current set. In this way, the

consecutive splitting elements are spread closer and closer to the target element y, thus gradually eliminating the size of the subset to be searched for y and improving the search time. Yao and Yao [60] proved a $\Theta(\log\log n)$ average search time for stored elements that are uniformly distributed. This was an exponential improvement upon binary search and the basic idea was to select as splitting element the closest one to the *expected* location of y in the current set. Since all elements are uniformly spread, the elements of the set that are near to the expected location of y are *sparse* enough. Therefore, selecting as splitting element the closest one to the expected location of y prunes greatly the size of the remaining set. In [21,22,38–40] several aspects of IS are described and analyzed. Willard [57] proved the same search time for the extended class of *regular* input distributions. In this (non-uniform) class, the probability density is allowed to fluctuate over a given range $[a, b]$, it is zero for elements outside this range, and its first derivative remains bounded; a regular distribution is not uniformly spread over $[a, b]$, and it has bounded rate of probability mass accumulation, guaranteeing an *almost* uniform spread of probability measure over $[a, b]$. Therefore, any subinterval of $[a, b]$ remains sparse enough, and there are not too many elements spread around the target element y. Once more, the search method enjoys the property that recursively selecting as splitting element the *expected* location of the target y in the current set leads rapidly to the *real* location of y. The IS method was generalized [13] to non-random input data that possess enough "pseudo-randomness" for effective IS to be applied.

The study of dynamic insertions of elements with respect to the uniform distribution and random deletions was initiated in [18,26]. In [18] an implicit data structure was presented supporting insertions and deletions in $O(n^{\varepsilon})$ time, $\varepsilon > 0$, and IS with expected time $O(\log\log n)$. The structure of [26] has expected insertion time $O(\log n)$, amortized insertion time $O(\log^2 n)$ and it is claimed, without rigorous proof, that it supports IS. Mehlhorn and Tsakalidis [32] demonstrated a novel dynamic version of the IS method, the *Interpolation Search Tree (IST)*, with $O(\log\log n)$ expected search and update time for a larger class than that of regular distributions. 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 (representatives), one element from each subtree, and an ID array that stores a set of sample elements approximating the inverse distribution function. 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.

Mehlhorn and Tsakalidis [32] considered μ-random insertions and random deletions. An insertion is μ-random[1] 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. They introduced the notion of a (f_1, f_2)-*smooth* probability density μ, in order to control the distribution of the elements in each subinterval dictated by an ID index. 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). The class of smooth distributions is a superset of uniform, bounded, and several non-uniform distributions (including the class of regular distributions). The results in [32] hold for $(n^{\alpha}, \sqrt{n})$-smooth densities, where $1/2 \leq \alpha < 1$ (cf. Section 2.1 for the formal definition of an (f_1, f_2)-smooth density).

Andersson and Mattsson [3], by further generalizing and refining the notion of smooth distributions, presented a variant of the IST called *Augmented Sampled Forest* extending the class of input distributions for which $\Theta(\log\log n)$ search time is expected. In particular, the time complexities of their structure holds for the larger class of $\left(\frac{n}{(\log\log n)^{1+\varepsilon}}, n^{\delta}\right)$-smooth densities, where $\delta \in (0, 1)$, $\varepsilon > 0$. Moreover, this structure exhibited $o(\log\log n)$ expected search time for some classes of input distributions. Finally in [27,29], a finger search version of these structures was presented with $O(1)$ update time (update position given) and $O(\log\log d)$ expected search time, where d denotes the distance between the search element and an element pointed to by a finger.

We note that all the aforementioned results on IS use the unit-cost real RAM model without supporting analytic functions, but enhanced with the floor function (to carry out the interpolation step; see Section 2.2), where it is implicitly assumed that the resulting integer has $O(\log n)$ bits since it indexes the ID array.

In this work, we further continue the investigation of IS. Our investigation is motivated by both theoretical and practical considerations. On the one hand, the average case analysis of IS is among the top open problems in the analysis of searching (see e.g., [45]), and as we show below, the problem was open for input distributions producing duplicate elements. On the other hand, recent developments in databases and computer hardware indicate that IS is of high (practical) importance today. In [23], the problem of fast searching within the nodes (pages) of a B-tree is investigated and the IS importance is argued by identifying IS as "one of the techniques required to win transaction processing benchmarks". As it is demonstrated in [23], modern developments in hardware (disks and CPUs) favor IS over binary search as a considerably faster method for reducing cache faults in B-tree indexes (a critical performance factor). The major drawback of IS identified in [23] is its poor performance when applied to non-uniform (skewed) data that produce duplicate elements, and a dozen of heuristics are discussed that (empirically) make IS to perform better when applied to non-uniform data. The current absence of provably good IS techniques (and analyses) in both a static and dynamic setting is emphasized in [23], encouraging researchers to work further towards new IS techniques that avoid skew and can handle duplicate elements.

[1] In the rest of the paper, we say that a random variable is μ-random or μ-randomly distributed if it follows the probability distribution μ.

1.3. New results

The analysis of all the aforementioned IS structures [3,21,22,27,29,32,39–41,57,60] was heavily based on the assumption that the conditional distribution on a subinterval during an arbitrary interpolation step remains unaffected. In particular, in [3,27,29,32] IS is performed on each node of a tree structure under the assumption that all elements in the subtree resulted during the previous interpolation step remain μ-random, conditioning only on the subinterval these elements belong to. In this way, tail bounds for the number of elements appearing in an arbitrary interval were obtained and this led to $O(1)$ expected search time, during each interpolation step.

Our ***first contribution*** in this work (Section 2) is to show that the above assumption is valid only when the produced elements are *distinct* (as indeed assumed in [3,21,22,27,29,32,39–41,57,60]), i.e., they are produced under some continuous distribution where the probability of collision is zero; otherwise, it *fails*. This means that the probabilistic analyses of previous dynamic interpolation search data structures are *inapplicable* to sequences of non-distinct elements, produced by discrete probability distributions with measurable (non-zero) probability of element collisions.

This lack of generalization does not have only theoretical, but also serious practical implications. There exist applications where we need to store duplicates, and thus the theoretically used density distribution modeling the input process should *not* produce distinct elements. A classical example is the creation of secondary indices in databases [31]. In a secondary index, duplicate values correspond to different records and they should be stored as distinct entities. There are also specific applications where interpolation search comes into play, such as fast searching in B-trees [23] (whose importance was discussed earlier; cf. end of Section 1.2), or the case of searching tables with alphabetic elements (e.g., names, dictionary entries) [38]. The elements in such tables follow a non-uniform, unknown, discrete probability distribution and collisions *do* occur. Other useful applications of interpolation search in non-uniform data are discussed in [10,17,23,38,40,43]. In all these papers it has been empirically observed that interpolation search has a very poor performance in such data. To alleviate this problem a series of heuristics have been introduced in [10,17,23,38,40,43], but no rigorous performance analyses have been provided. In [38,39], it was suggested that such an analysis would be possible if one considers the idea in [22] that translates any continuous input distribution to a uniform one.

In Section 2, we also show that this idea of taking advantage of the cumulative distribution [22,38,39] does *not* apply to discrete distributions with measurable probability of element collisions (a fact that was indeed experimentally verified in [38]). The above pluralism of efforts demonstrates the necessity to handle non-uniform data generated by discrete distributions with measurable probability of element collisions. One could be tempted to argue that the inapplicability of the previous analyses could be faced by simply storing duplicate elements once; moreover, in these structures the main rebalancing tool is local/global rebuilding, which can be easily modified to produce input sequences with distinct elements. Both arguments are wrong, however, since the new sequences of distinct elements are artificial sequences, different from the initial. Consequently, important statistical properties of the elements are destroyed and the probabilistic analyses fail.

Our ***second contribution*** in this paper is a new dynamic interpolation search data structure (Section 3) that overcomes the above problems, and in which the elements stored in each subtree preserve the input distribution, conditioning only on the interval that corresponds to the current subtree. The new structure is quite simple and it features a number of advantages over previous approaches:

- It exhibits similar expected $O(\log\log n)$ search time as the previous dynamic interpolation structures [3,21,22,32,39–41, 57,60].
- Its probabilistic analysis is *always* valid irrespectively of the distinctness or not of the elements in the input sequence, that is, *regardless* of whether they are produced by a continuous or a discrete distribution.
- It applies to the same classes of distributions as those in [3,32] and it holds w.h.p.,[2] while those in [3,21,22,32,39–41, 57,60] did not grant such guarantee.
- We get, as a by-product of our construction, a dynamic search data structure with $O(1)$ expected search time for a wide class of input distributions (Section 4). This result significantly extends the class of input distributions in [3] under which $O(1)$ expected search time was possible. In particular, we achieve constant query time for the class of $\left(\frac{n}{g}, \ln^{O(1)} n\right)$-smooth distributions, where g is a constant, which includes that of bounded ($(n,1)$-smooth) densities, for which $O(1)$ expected search time was known [3]. In addition, this search time (at least for the case of discrete distributions) also holds w.h.p., while those in [3] did not grant such property.

The main idea of our new data structure is to select the splitting elements or representatives (elements guiding the search) deterministically and independently of the specific stored set of elements. Based on such a selection, the initial interval of elements is partitioned into equally-sized bins. We then apply this process recursively on each bin, thus forming layers of bins that define naturally a tree structure. When every bin contains less than a poly-logarithmic number of elements, the recursive process is terminated.

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

Our approach eliminates the need for using a REP array (recall the discussion in Section 1.2) and this is in sharp contrast with all previous approaches for dynamic IS; that is, contrary to all previous approaches, our data structure does not use a REP array.

Although the class of smooth distributions includes, for appropriate choices of f_1 and f_2, any other probability distribution, the effective range of f_1, f_2 for which $O(\log\log n)$ IS time is achieved excludes distributions of major practical importance; for instance, power law [33], Binomial, etc.

Our ***third contribution*** in this work is that we are able to show (Section 5) that a slight modification of our data structure achieves $O(\log\log n)$ time w.h.p. for power law and Binomial distributions. No previous IS structure achieves such a time bound for these distributions (we mention the deterioration of IS that was experimentally observed in [10,17,23,38,40,43]).

Our data structure is *robust* (as those in [3,27,29,32,57]), i.e., it remains efficient *without* a priori knowledge of the particular continuous or discrete distribution, although certain parameters of the family of distributions is assumed to be known. Moreover, our new structure uses the same model with all the aforementioned IS data structures, that is, the unit-cost real RAM model without analytic functions, but enhanced with the floor function.

In summary, our new data structure enjoys the following key features that make it compare favorably to previous ones: (1) It works regardless of whether the input distribution is continuous or discrete. (2) It is the first one that handles duplicate elements (with respect to dynamic interpolation search). (3) It is the first one that provides high probability searching bounds. (4) Simple extensions of it achieve $O(1)$ searching bounds for certain distributions, while preserve the high probability $O(\log\log n)$ search bound for the important (but non-smooth) power law and Binomial distributions.

The rest of this paper is structured as follows. In Section 2, we define the smooth probability distributions, review interpolation search and show why the previous approaches fail when the elements are not distinct. In Section 3, we present our new interpolation search data structure. In Section 4, we show that a variant of our new structure achieves $O(1)$ expected search time for a wide class of input distributions. In Section 5, we present our interpolation search treatment of the important (but non-smooth) power law and Binomial distributions. We conclude in Section 6. Very preliminary parts of this work appeared as [28].

2. Probabilistic analysis of Interpolation Search revisited

In this section, we define formally the smooth probability distributions, review the interpolation search idea in [3,32], and show its inapplicability when the elements are not distinct (probability of collision is not zero).

2.1. Smooth probability distribution

We start with the formal definition of the smooth probability density, which is central in our discussion, both for *discrete* and *continuous* probability distributions.

2.1.1. Continuous case

Consider an unknown *continuous* probability distribution over the interval $[a,\ b]$ with density function $\mu(x)=\mu_{[a,b]}(x)$.

Given two functions f_1 and f_2, then $\mu(x)=\mu_{[a,b]}(x)$ is (f_1, f_2)-*smooth* [3,32] 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, for a random element X it holds:

$$\Pr\left[X\in\left[c_2-\frac{c_3-c_1}{f_1(n)},c_2\right]\ |c_1\le X\le c_3\right]=\int_{c_2-\frac{c_3-c_1}{f_1(n)}}^{c_2}\mu_{[c_1,c_3]}(x)dx\le\frac{\beta f_2(n)}{n} \tag{1}$$

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 $p=\int_{c_1}^{c_3}\mu(x)dx$ and $p>0$.

Intuitively, function f_1 partitions an arbitrary subinterval $[c_1,c_3]\subseteq[a,b]$ into f_1 equal-length parts, each of length $\frac{c_3-c_1}{f_1}=O\left(\frac{1}{f_1}\right)$; that is, f_1 measures how *fine* is the partitioning of an arbitrary subinterval $[c_1,c_3]\subseteq[a,b]$. Function f_2 guarantees that no part, of the f_1 possible, gets more probability mass than $\frac{\beta\cdot f_2}{n}$; that is, f_2 measures the *sparseness* of any subinterval $\left[c_2-\frac{c_3-c_1}{f_1},c_2\right]\subseteq[c_1,c_3]$. 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 [3,32]. Actually, *any* probability distribution is $(f_1,\Theta(n))$-smooth, for a suitable choice of β.

It is helpful to see the intuition behind the idea of an (f_1, f_2)-smooth distribution, as quoted (in italics) from [3, paragraph above Def. 2]: "*among a number* (measured by $f_1(n)$) *of consecutive subintervals, no subinterval should be too dense* (measured by $f_2(n)$) *compared to the others*".

As it is proved in [3], the class of smooth densities defines a natural hierarchy in the sense that the class of $(f_1(n), f_2(n))$-smooth densities contains the class of $(h_1(n), h_2(n))$-smooth densities as long as $f_i(n)$, $h_i(n)$, and $f_i(n)/h_i(n)$ are non-decreasing functions, for $i=1,2$.

For instance, the class of (n, n)-smooth densities contains all classes of distributions, since for $\beta = 1$ any interval may have probability mass up to 1. The class of $\left(n^{1/2}, f_2(n)\right)$-smooth distributions contains the class of $\left(n^{1/3}, f_2(n)\right)$ distributions, since roughly for the same probability mass $\beta \frac{f_2(n)}{n}$ the first class requires smaller intervals (of length equal to $\frac{c_3-c_1}{n^{1/2}}$) to guarantee such an upper bound on the probability mass than the second class (which requires intervals of length equal to $\frac{c_3-c_1}{n^{1/3}}$). In this respect, the class of $(n^\alpha, \sqrt{n})$-smooth distributions $(1/2 \le \alpha < 1)$ is contained within the class of (n^α, n^δ)-smooth distributions for $\alpha, \delta \in (0, 1)$, $\alpha + \delta \ge 1$, which in turn is contained into the larger class of $\left(\frac{n}{(\log\log n)^{1+\varepsilon}}, n^\delta\right)$-smooth densities, for arbitrary $0 < \delta < 1$, $\varepsilon > 0$.

2.1.2. Discrete case

Before moving to the formal definition of the discrete smooth probability density, which is central in our discussion, it is imperative to discuss the characteristics of the discrete sample space, that is the universe $\mathcal{U}$ of the generated elements. This is crucial for the definition of the smoothness property for the discrete case.

In this spirit, we must clarify how the required ℓ bits to describe each element in the stored set S are related to the word length w required to describe each element of the universe $\mathcal{U}$, and how this is related to the complexity of *predecessor search*.

Note that the set S consists of $n = |S|$ elements, each of $\ell \le w$ bits, drawn from the universe $\mathcal{U} = \{1, \ldots, 2^w\}$, in a unit-cost RAM with word length $w = \log|\mathcal{U}|$. Since the ℓ bits must represent all the n elements in S, we require that $\log|S| = \log n \le \ell \le w = \log|\mathcal{U}|$. We stress that the set size $n = |S|$ is critically related to the universe size $2^w = |\mathcal{U}|$, when we wish to study computationally interesting cases of *predecessor search*. According to [37] (and the references therein) the interesting cases are when $|\mathcal{U}| \ge 2^{c\log n} = n^c$ with $c > 1$. In particular, when $c \le 1$, predecessor search takes $O(1)$ time and $O(n)$ space.

Having the above in mind, the interesting case for *predecessor search* is to define the (f_1, f_2)-*smooth* unknown *discrete* probability distribution μ over the elements of a sufficiently large universe $|\mathcal{U}| = n^c$, with $c > 1$, w.r.t. $n = |S|$.

Given two functions f_1 and f_2, then $\forall x \in \mathcal{U}$, the unknown discrete probability distribution $\mu(x)$ is (f_1, f_2)-*smooth* if there exists a constant β, such that for all $c_1, c_2, c_3 \in \mathcal{U} : c_1 < c_2 < c_3$, and for all naturals $\nu \le n$, for a random element $X \in \mathcal{U}$ it holds that:

$$\Pr\left[c_2 - \left\lfloor \frac{c_3 - c_1}{f_1(\nu)} \right\rfloor \le X \le c_2 | c_1 \le X \le c_3\right] = \sum_{x=c_2-\left\lfloor \frac{c_3-c_1}{f_1(\nu)} \right\rfloor}^{c_2} \mu_{[c_1,\ c_3]}(x) \le \frac{\beta f_2(\nu)}{\nu} \tag{2}$$

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 $x \in \{c_1, \ldots, c_3\}$ where $p = \sum_{x=c_1}^{c_3} \mu(x)$ and $p > 0$.

The above imply that all elements have small probability mass, i.e., a value with $o(1)$ probability, and no element has probability mass bounded from below by a positive constant.

To elaborate on this, consider (just to ease the exposition) the class of (ν^α, ν^δ)-smooth distributions for $\alpha, \delta \in (0, 1)$, $\alpha + \delta \ge 1$. If we initially consider the whole universe of elements with $|\mathcal{U}| = n^c, c > 1$, and $\forall \nu \le n$ we equally split it into $f_1(\nu) = \nu^\alpha$ many equal consecutive subsets of elements, then (2) implies that each subset (containing at least $\frac{|\mathcal{U}|}{f_1(\nu)} = \frac{n^c}{\nu^\alpha} = \omega(1)$ consecutive elements) gets probability mass at most $\frac{f_2(\nu)}{\nu} = \frac{\nu^\delta}{\nu}$, $\forall \nu \le n$, which is $o(1)$ as $\nu \to \infty$ and $f_2(\nu) = \nu^\delta$, $\forall \delta \in (0, 1)$. Hence, as $n \to \infty$, each element in $\mathcal{U}$ has $o(1)$ probability mass. This reasoning generalizes verbatim when conditioning to *any* subset Z of $\mathcal{U}$, containing $|Z| \le n^c = |\mathcal{U}|$ consecutive elements.

Note here a technical difference between (1) and (2), due to the comparison of a "real subinterval" (where random elements are drawn w.r.t. a continuous μ) to a "discrete subset" (where random elements are drawn w.r.t. a discrete μ). A real subinterval contains an *infinitum* of elements, and this is the reason that the upper bound of the real subinterval probability mass in (1) holds for any n. But, the case in (2) now deals with a discrete universe $\mathcal{U}$, where each subset Z of $\mathcal{U}$ is limited to contain $|Z| \le |\mathcal{U}|$ consecutive discrete elements. Hence, when conditioning to an arbitrary Z, we relax the requirement of any n in (1), and now it holds only for naturals $\nu \le |Z| \le |\mathcal{U}|$.

Once more, we can describe (2) by rephrasing the intuitive description of (f_1, f_2)-smooth distribution in [3, paragraph above Def. 2] as: "*among a number* (measured by $f_1(\nu)$) *of consecutive subsets, each containing consecutive elements from* $\mathcal{U}$*, no subset containing consecutive elements from* $\mathcal{U}$ *should be too dense* (measured by $f_2(\nu)$) *compared to the others*".

It can be proved (by a straightforward adaptation of the proof in [3] for the continuous case) that the class of (discrete) smooth densities, defined by (2), establishes a natural hierarchy in the sense that the class of $(f_1(n), f_2(n))$-smooth densities contains the class of $(h_1(n), h_2(n))$-smooth densities as long as $f_i(n)$, $h_i(n)$, and $f_i(n)/h_i(n)$ are non-decreasing functions, for $i = 1, 2$.

2.2. Interpolation search

The fundamental dynamic predecessor searching problem in data structures is defined, for our purposes, as follows. Consider the random set $S = \{X_1, \ldots, X_n\}$, where each element $X_i \in [a, b] \subset \mathbb{R}$, obeys an unknown (discrete or continuous)

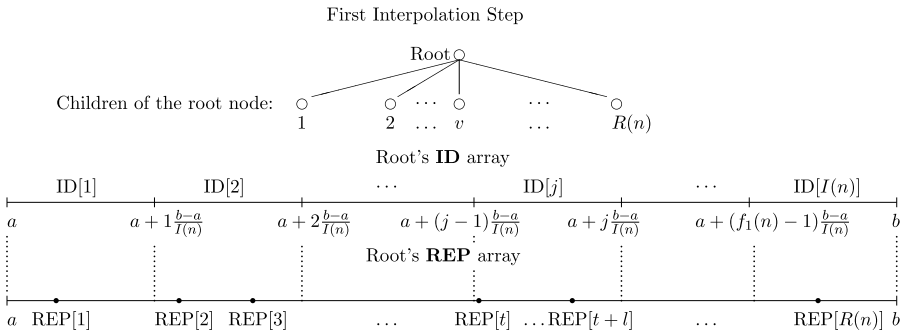

Fig. 1. First interpolation step on the root node of the IST.

distribution μ, $i = 1, \ldots, n$. Let the sequence $P = \langle X_{(1)}, \ldots, X_{(n)} \rangle$ be an increasing ordering of the set S. The goal is to find the largest element $X_{(j)} \in S$ that precedes a *target* element y in P. We describe how the *Augmented Sampled Forest (ASF)* [3], which is a generalization of the *Interpolation Search Tree (IST)* [32], can be used to search for this target element y.

Assume that the (discrete or continuous) distribution μ is $(I(n), n/R(n))$-*smooth*, where $I(n) : \mathbb{R}_0^+ \to \mathbb{R}_0^+$ and $R(n) : \mathbb{R}_0^+ \to \mathbb{R}_0^+$ are two non-decreasing and invertible functions with a second derivative less than or equal to zero.

The ASF is a two level data structure; the top level is an *ideal* static IST [32] while the bottom level is a sequence of buckets. The structure is maintained by using the global rebuilding technique [36] and its expected search time is dominated by the expected search time at the top level. At the top level, the root node has $R(n)$ children,[3] and similarly each child node has $R\left(\frac{n}{R(n)}\right)$ subchildren. The root node corresponds to the ordered sequence P of size n. Each child corresponds to a part of P of size $\frac{n}{R(n)}$. That is, these $R(n)$ children partition the ordered sequence P into $R(n)$ equal-sized subsequences $P_1, \ldots, P_{R(n)}$, of the form $\left\langle X_{(1)}, \ldots, X_{(\frac{n}{R(n)})} \right\rangle, \ldots, \left\langle X_{((R(n)-1)\frac{n}{R(n)}+1)}, \ldots, X_{(n)} \right\rangle$. This procedure is recursively applied to their children and so on, until the number of elements becomes $\log^{O(1)} n$, in which case the elements are put into buckets.

Each node of the tree contains a pair of arrays, namely ID and REP, that help to locate the appropriate child eligible to contain the target element y. In the root node the set of indices of the ID array is $[1, \ldots, I(n)]$ and the set of indices of the REP array is $[1, \ldots, R(n)]$. The role of the ID array of the root node is to partition the interval $[a, b]$ into $I(n)$ equal-length parts, each of length $\frac{b-a}{I(n)}$. When searching for an element y, the first *interpolation* step determines in $O(1)$ time the number j

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

which denotes the j-th interval I_j of length $\frac{b-a}{I(n)}$ that contains the target y:

$$I_j = \left[a + (j-1)\frac{b-a}{I(n)},\ a + j\frac{b-a}{I(n)} \right) \tag{4}$$

The role of the array REP$[1, \ldots, R(n)]$ of the root node is to partition the ordered sequence P into $R(n)$ equal-sized subsequences, each of size $\frac{n}{R(n)}$. The index REP$[i]$, $i = 1, \ldots, R(n)$, points to the i-th subsequence P_i, where $P_1 = \left\langle X \in P \mid X_{(1)} \le X \le X_{(\frac{n}{R(n)})} \right\rangle$ and $P_i = \left\langle X \in P \mid X_{\left((i-1)\frac{n}{R(n)}\right)} < X \le X_{\left(i\frac{n}{R(n)}\right)} \right\rangle$ for $2 \le i \le R(n)$. Alternatively, REP$[i]$ can be seen as the *representative* of the element $X_{\left(i\frac{n}{R(n)}\right)}$ of P_i.

The first interpolation step, provided by Eq. (3), determines within $O(1)$ time the subinterval I_j described by Eq. (4), where the target element y belongs. If in this subinterval correspond $O(1)$ REP indices, then within $O(1)$ time we can determine the unique REP index that corresponds to the subsequence that element y may belong.

Fig. 1 illustrates the interplay between the ID and the REP arrays of the root node. The upper line represents the partition of the interval $[a, b]$ into $I(n)$ equal-length parts, and the lower line represents the partition of $[a, b]$ into $R(n)$ (not necessarily of equal size) parts by the REP array. Two consecutive vertical dashed lines that enclose an ID$[i]$ interval in the upper line, enclose all the REP indices REP$[i_1], \ldots,$ REP$[i_k]$ in the lower line with their representative elements $X_{\left(i_1\frac{n}{R(n)}\right)}, \ldots, X_{\left(i_k\frac{n}{R(n)}\right)}$ belonging to this ID$[i]$ interval, $i = 1, \ldots, I(n)$.

[3] Whenever $R(n)$ and $I(n)$ refer to an integral quantity, we mean $\lceil R(n) \rceil$ and $\lceil I(n) \rceil$, respectively. Similarly, we assume that all fractions (like $n/R(n)$) are contained in $\lceil \cdot \rceil$.

Assume now that when searching for y, using Eq. (3), the first interpolation step yields the ID index ID[1], i.e., $j=1$. Then, Eq. (4) implies that $y \in \left[a, a+\frac{b-a}{I(n)}\right]$. In this case (see Fig. 1), the ID index ID[1] *happily* points to the *unique* REP index REP[1]. Therefore, if y appears in sequence P, then it should be searched in the subsequence $P_1 = \left\langle X_{(1)}, \ldots, X_{\left(\frac{n}{R(n)}\right)}\right\rangle$ of size $\frac{n}{R(n)}$. Fortunately, the first interpolation step is highly efficient, since within $O(1)$ time the size n of the initial sequence is pruned to a subsequence of size $\frac{n}{R(n)}$ to be searched for y. However, as Fig. 1 illustrates, it could also be possible for the first interpolation step to *unhappily* yield the ID index ID[j] whose vertical lines enclose $l+1$ REP indices REP[t], ..., REP[$t+l$]. Therefore, to locate the unique subsequence that y may belong, amongst subsequences $P_t, \ldots, P_{t+l}$, we have to compare y with each REP[s] $\equiv X_{\left(s\frac{n}{R(n)}\right)} \in P, s=t,\ldots,t+l$. This requires $\Omega(l)$ time and if this interval is *dense*, then l may become inefficiently large.

We conclude that the search efficiency highly depends on the distribution of the REP indices over each ID subinterval of $[a,b]$. In other words, each ID index that corresponds to a *dense* subinterval of $[a,b]$ causes a great slow down of the search speed.

Most importantly, by applying the same process recursively, suppose that the second interpolation step now yields REP[$s-1$] $< y \le$ REP[s]. Then, y must be searched for into the subsequence $P_s = \left\langle X_{\left((s-1)\frac{n}{R(n)}+1\right)}, \ldots, X_{\left(s\frac{n}{R(n)}\right)}\right\rangle$. A crucial observation is that its endpoints $X_{\left((s-1)\frac{n}{R(n)}+1\right)}, X_{\left(s\frac{n}{R(n)}\right)}$ may in general be neither μ-random nor smooth; we will elaborate on this matter in the proof of Lemma 1 (Section 3).

Now, an (f_1, f_2)-*smooth* probability density μ is used to control the distribution of the elements in each subinterval dictated by an ID index. Intuitively, suppose that after some interpolation steps we reach a node u of the search tree. This is the root of a subtree with n_v nodes spread into the interval $I_u=[a_u,b_u] \subseteq [a,b]$. Then, $f_1(n_u) = I(n_u)$ equals the number of the equal-length subintervals, each of length $\frac{b_u-a_u}{f_1(n_u)}$, that the ID array of node u partitions I_u, measuring the fineness of the partitioning of an arbitrary subinterval. On the other hand, the conditional on I_u probability mass of μ in any ID subinterval of I_u is at most $\beta f_2(n_u)/n_u = O(1/R(n_u))$, i.e., f_2 determines the sparseness of any subinterval.

In [32] the authors prove that, for an unknown $(n^\alpha, \sqrt{n})$-smooth density μ, $1/2 \le \alpha < 1$, in an arbitrary ID subinterval, $O(1)$ REP indices are expected to appear. In [3] this was extended to the larger class of $\left(\frac{n}{(\log\log n)^{1+\varepsilon}}, n^\delta\right)$-smooth densities, for arbitrary $0<\delta<1$, $\varepsilon>0$.

2.3. A randomness invariant

For the case of continuous distributions, the analyses in [3,27,29,32] assume that the elements in an arbitrary subsequence $P_v = \left\langle X_{\left((v-1)\frac{n}{R(n)}\right)}, \ldots, X_{\left(v\frac{n}{R(n)}\right)}\right\rangle$, dictated by an interpolation step, *remain μ-randomly distributed* conditioned only on the endpoints of the subinterval $\left(X_{\left((v-1)\frac{n}{R(n)}\right)}, X_{\left(v\frac{n}{R(n)}\right)}\right] \subseteq [a,b]$. That is, for a random element x in subsequence P_v, its *conditional* probability density equals

$$\mu\left[x | x \in \left(X_{\left((v-1)\frac{n}{R(n)}\right)}, X_{\left(v\frac{n}{R(n)}\right)}\right]\right] = \frac{\mu(x)}{\Pr\left[x \le X_{\left(v\frac{n}{R(n)}\right)}\right] - \Pr\left[x \le X_{\left((v-1)\frac{n}{R(n)}\right)}\right]} = \frac{\mu(x)}{\Pr\left[X_{\left((v-1)\frac{n}{R(n)}\right)} < x \le X_{\left(v\frac{n}{R(n)}\right)}\right]} \tag{5}$$

This nice property in (5), which is *true* for continuous distributions, allows to use recursively Ineq. (1) per subinterval, with crucial role in tuning (via parameters f_1, f_2) the probability mass per such subinterval dictated by each interpolation step.[4]

However, the rule is that in general a discrete distribution has *non zero* probability of element collisions. A side effect of this, as we exhibit in Section 2.4, is that the elements into an arbitrary subsequence $P_v = \left\langle X_{\left((v-1)\frac{n}{R(n)}\right)}, \ldots, X_{\left(v\frac{n}{R(n)}\right)}\right\rangle$, dictated by an interpolation step, *do not* remain as μ-randomly distributed conditioning only on the endpoints of the

[4] The analyses in [22,38,39] ingeniously apply the cumulative distribution function F on the ordered elements in $P = \langle X_{(1)}, \ldots, X_{(n)}\rangle$, yielding $P_F = \langle F(X_{(1)}), \ldots, F(X_{(n)})\rangle$. Now, each $F(X_{(i)}) \in P_F$ is *uniformly* distributed over $[0,1]$, since $\Pr[F(X_{(i)}) \le t] = \Pr[X_{(i)} \le F^{-1}(t)] = F(F^{-1}(t)) = t$ (see [14, pp. 36-37]). Thus, sequence P_F is very suitable for applying IS on it. That is, to search for target element y, split P_F on element $F(X_{(j_y)}) \approx \frac{F(y)-F(X_{(1)})}{F(X_{(n)})-F(X_{(1)})}$, and recursively apply IS to $P_F^- = \langle F(X_{(1)}), \ldots, F(X_{(j_y)})\rangle$, if $F(y) \le F(X_{(j_y)})n$, otherwise to $P_F^+ = P_F \setminus P_F^-$. However, this approach *also* tacitly assumes that the conditional distribution of the elements in subsequences P_F^-, P_F^+ remains unaffected and obeys Eq. (5).

subinterval $\left(X_{\left((v-1)\frac{n}{R(n)}\right)}, X_{\left(v\frac{n}{R(n)}\right)}\right] \subseteq [a,b]$. Thus, the key property in (5) can not be applied recursively conditioning only on the endpoints of each subinterval.

2.4. Discrete distributions with measurable probability of element collision

Consider the simple case of three stored elements (random variables) $X_1, X_2, X_3 \in [a,b]$ drawn according to some μ-random smooth distribution – the general case of n variables can be easily deduced from this case by a simple induction argument. These elements are identically and independently distributed. For each $i = 1, 2$ the corresponding REP[i] is a *new* random variable defined as $\text{REP}[1] \equiv X_{(1)} = \min\{X_1, X_2, X_3\}$, $\text{REP}[2] \equiv X_{(3)} = \max\{X_1, X_2, X_3\}$. The phenomenon that could make us suspicious that something is wrong is that each random element REP[i], $i = 1, 2$, in contrast to each random element X_1, X_2, X_3, *does not* follow the initial distribution μ. Since the elements REP[i], $i = 1, 2$ are the *minimum* and *maximum* elements of the subsequence P_v we realize that this should affect the distribution of the elements in subsequence P_v. Let us study the distribution of a random element X into the subinterval [REP[1], REP[2]].

$$\begin{aligned}\Pr[X = \lambda \mid \text{REP}[1] = a' \le X \le \text{REP}[2] = b'] &= \Pr[X = \lambda \mid X_{(1)} = a' \cap X_{(3)} = b'] \\ &= \frac{\Pr[X = \lambda \cap X_{(1)} = a' \cap X_{(3)} = b']}{\Pr[X_{(1)} = a' \cap X_{(3)} = b']},\end{aligned} \tag{6}$$

where $a' < \lambda < b'$. The event $\{X_{(1)} = a' \cap X_{(3)} = b'\}$ occurs if at least one of the following mutually disjoint events occurs:

$$\begin{aligned}&\{X_1 = a',\ X_2 = b',\ a' < X_3 < b'\}, \quad \{X_2 = a',\ X_1 = b',\ a' < X_3 < b'\},\\ &\{X_1 = a',\ X_3 = b',\ a' < X_2 < b'\}, \quad \{X_3 = a',\ X_1 = b',\ a' < X_2 < b'\},\\ &\{X_2 = a',\ X_3 = b',\ a' < X_1 < b'\}, \quad \{X_3 = a',\ X_2 = b',\ a' < X_1 < b'\}.\end{aligned} \tag{7}$$

Besides the events listed in (7), at least one of the following mutually disjoint events occurs (in the following, by $X_{i,j} = y$ we mean $X_i = y$ and $X_j = y$):

$$\begin{aligned}&\{X_{1,2} = a',\ X_3 = b'\}, \{X_{1,2} = b',\ X_3 = a'\},\ \{X_{1,3} = a',\ X_2 = b'\},\\ &\{X_{1,3} = b',\ X_2 = a'\}, \{X_{2,3} = a',\ X_1 = b'\},\ \{X_{2,3} = b',\ X_1 = a'\}\end{aligned} \tag{8}$$

Combining (7) and (8) the denominator of (6) equals:

$$\begin{aligned}\Pr[X_{(1)} = a' \cap X_{(3)} = b'] = 3\Pr[X = a']^2 \Pr[X = b'] + 3\Pr[X = a']\Pr[X = b']^2 \\ +6\Pr[X = a']\Pr[X = b']\Pr[a' < X < b']\end{aligned} \tag{9}$$

Similarly, the event $\{X = \lambda \cap X_{(1)} = a' \cap X_{(3)} = b'\}$, with $a' < \lambda < b'$, occurs if one of the following mutually disjoint events occurs:

$$\begin{aligned}&\{X_2 = \lambda,\ X_3 = a',\ X_1 = b'\}, \quad \{X_1 = \lambda,\ X_2 = a',\ X_3 = b'\},\\ &\{X_3 = \lambda,\ X_1 = a',\ X_2 = b'\}, \quad \{X_1 = \lambda,\ X_3 = a',\ X_2 = b'\},\\ &\{X_3 = \lambda,\ X_2 = a',\ X_1 = b'\}, \quad \{X_2 = \lambda,\ X_1 = a',\ X_3 = b'\}.\end{aligned} \tag{10}$$

From (10) the numerator of (6) equals:

$$\begin{aligned}&\Pr[X = \lambda \cap X_{(1)} = a' \cap X_{(3)} = b'] =\\ &6\Pr[X = a']\Pr[X = b']\Pr[a' < X < b']\end{aligned} \tag{11}$$

Combining (9) and (11) we get for $a' < \lambda < b'$ that the probability of $X = \lambda$ conditioning on the corresponding values of the minimum and maximum element is:

$$\Pr[X = \lambda | X_{(1)} = a' \cap X_{(3)} = b'] = \frac{\Pr[X = \lambda]}{\frac{\Pr[X=a']+\Pr[X=b']}{2} + \Pr[a' < X < b']} \tag{12}$$

Note, however, that (12) (which is the discrete analog of the randomness invariant described in Section 2.3 for continuous distributions) is different from the corresponding probability of $X = \lambda$, for $a' < \lambda < b'$, conditional on the endpoints of this subinterval, which is depicted in (13) below:

$$\Pr[X = \lambda | a' \le X \le b'] = \frac{\Pr[X = \lambda]}{\Pr[a' \le X \le b']} \tag{13}$$

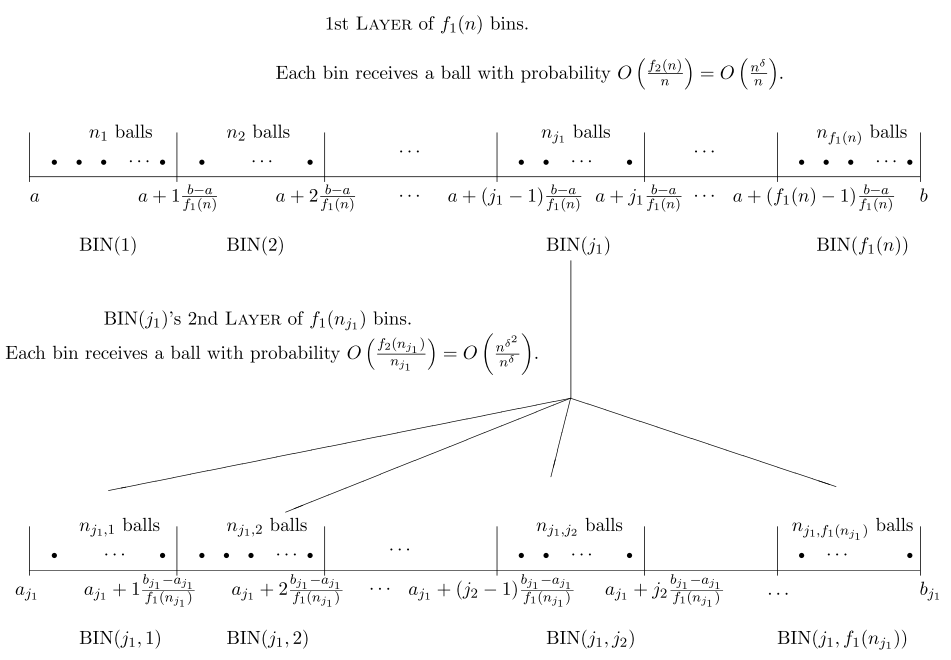

Fig. 2. The first interpolation step indicates within $O(1)$ time the index j_1 of BIN(j_1). Given that μ is $(f_1(n), f_2(n)) = (n^\alpha, n^\delta)$-smooth, no bin of this layer gets w.h.p. more that $O(n^\delta)$ balls. The second interpolation step indicates within $O(1)$ time the index j_2. Given that μ is $(f_1(n^\delta), f_2(n^\delta)) = ((n^\delta)^\alpha, n^{\delta^2})$-smooth, no bin of this layer gets w.h.p. more that $O(n^{\delta^2})$ balls.

In fact, we can easily see that, per point λ in this subinterval, (12) assigns greater probability mass than (13), since the denominator of (13) equals

$$\begin{aligned}\Pr[a' \le X \le b'] &= \Pr[X = a'] + \Pr[X = b'] + \Pr[a' < X < b'] \\ &> \frac{\Pr[X = a'] + \Pr[X = b']}{2} + \Pr[a' < X < b'] \end{aligned} \tag{14}$$

where the last expression is the denominator of (12).

We conclude that, when the probability of collisions is measurable, the net effect of choosing, as endpoints of subintervals, not deterministically obtained values is to *destroy* the randomness invariant (Section 2.3), which is crucial for simplifying the study of the smoothness of the distribution of the elements per subinterval.

3. The new Interpolation Search data structure

A possible solution to the aforementioned problem would be to replace the representatives (elements guiding the search) in each node of the tree with values that are deterministically obtained and are independent of the specific stored set. This is roughly the basic idea behind the data structure that we present in this section.

Consider a dynamic set S containing $O(n)$ elements drawn from the interval $[a, b]$, according to a continuous or discrete distribution μ. For ease of exposition we consider the case of $(f_1, f_2) = (n^\alpha, n^\delta)$-smooth density for $\alpha, \delta \in (0, 1)$, $\alpha + \delta \ge 1$ and discuss the larger class of $\left(\frac{n}{(\log\log n)^{1+\varepsilon}}, n^\delta\right)$-smooth densities (where $\delta \in (0, 1), \varepsilon > 0$) at the end of the section.

3.1. Description of the new data structure

Intuitively, the new data structure can be considered as a dynamic *balls into bins* random game, where balls correspond to elements and bins to suitably chosen subintervals of $[a, b]$. First, we describe the structure, which is a leaf-oriented[5] tree structure, then the supported operations (search and update), and finally we provide an algorithm to construct the data structure on a set of n elements.

Our data structure consists of Layers of bins; see Fig. 2. We can view the whole structure as a single bin at the 0th Layer containing n elements that represent the stored elements. The 1st Layer partitions the interval $[a, b]$ into $f_1(n)$ equal-length bins. We define as BIN(j_1), the j_1-th bin in the 1st Layer of bins, which corresponds to the subinterval

[5] All elements are stored in the leaves of the tree, while internal nodes contain only routing information.

$$\left[a+(j_1-1)\frac{b-a}{f_1(n)},\ a+j_1\frac{b-a}{f_1(n)}\right)=[a_{j_1},\ b_{j_1})\subset[a,\ b],\quad j_1=1,\ldots,f_1(n)$$

Only the last subinterval is closed from both sides.

In the following, the subscript i of j_i will denote the i-th Layer of bins. By slightly abusing notation, we assume that BIN(j_0), where $j_0=1$, corresponds to all elements of set S and will only be used when necessary. We assume that $a_{j_0}=a$ and $b_{j_0}=b$.

In the first layer we store in increasing sorted order the minimum element a_{j_1} of each subinterval. Thus, we store in an array the sequence $a_1,a_2,\ldots,a_{f_1(n)}$ to facilitate the interpolation search. Each such cell of the array also contains the number of elements within the respective bin. For example, BIN(1) is represented by the first cell of the array of the first layer which contains a_1 as well as the number of elements that fall in BIN(1). The same structure applies to all layers.

A random element $X\in S$ chosen according to μ is stored in BIN(j_1) iff X belongs to the subinterval $[a_{j_1},b_{j_1})$. The subset $S_{j_1}\subseteq S$ consists of all n_{j_1} elements that are stored in BIN(j_1), where $n_1+\ldots+n_{f_1(n)}=|S|=O(n)$, and $j_1=1,\ldots,f_1(n)$, where $f_1(n)=n^\alpha$.

The 2nd Layer of bins is constructed by recursively partitioning each BIN(j_1) of the 1st Layer into $f_1(n_{j_1})$ equal-length bins, i.e., BIN(j_1) containing n_{j_1} elements is partitioned into equal-length bins BIN(j_1,j_2), with corresponding indices $j_1=1,\ldots,f_1(n)=n^\alpha$, and $j_2=1,\ldots,f_1(n_{j_1})=(n_{j_1})^\alpha$. In this case, BIN($j_1,j_2$) corresponds to the subinterval

$$\left[a_{j_1}+(j_2-1)\frac{b_{j_1}-a_{j_1}}{f_1(n_{j_1})},a_{j_1}+j_2\frac{b_{j_1}-a_{j_1}}{f_1(n_{j_1})}\right)=\left[a_{j_1,j_2},\ b_{j_1,j_2}\right)\subset[a_{j_1},\ b_{j_1})\subset[a,b]$$

A random element $X\in S$ chosen according to μ is stored in BIN(j_1,j_2), iff X belongs to the subinterval $[a_{j_1,j_2},\ b_{j_1,j_2})$. The subset $S_{j_1,j_2}\subseteq S_{j_1}$ consists of all n_{j_1,j_2} elements that are stored in BIN(j_1,j_2), such that $n_{j_1,1}+\ldots+n_{j_1,f_1(n_{j_1})}=|S_{j_1}|=n_{j_1}$.

We proceed recursively with the subsequent Layers by further partitioning the bins of Layer i, $i\geq 2$, in order to construct the bins of Layer $i+1$.

This recursive process determines naturally a tree structure and it is carried out as long as a bin contains more than $\log^{1/\delta}n$ elements, where n is the initial number of elements. When a bin contains less than $\log^{1/\delta}n$ elements, it is not further partitioned and becomes a leaf of the structure. Additionally, in order to guarantee worst-case time complexities, we employ a second terminating condition according to which the recursive process is stopped as soon as the number of recursive layers exceeds $\log n$, irrespectively of the number of elements within these bins at recursive layer $\log n$.

The elements associated with each leaf bin are stored as a constant update balanced search tree [16,30]. This tree supports, position given,[6] update operations in constant worst-case time, while searching for an element costs $O(\log n)$ time, when the tree contains n elements. Thus, if the tree of a leaf bin contains $\log^{1/\delta}n$ elements, then a search operation is supported in $O(\frac{1}{\delta}\log\log n)=O(\log\log n)$ time.

Since our data structure consists of Layers of bins determining a tree structure, upon which interpolation search is executed, we call it the *Layered Interpolation Search Tree* (LIST). Furthermore, we call the maximum number of Layers (or alternatively the length of a path from a leaf bin to the root of the tree structure) as the *height* of LIST.

Searching in the LIST for a target element y is carried out as follows. Let BIN(j_1) be the bin of the 1st Layer, where the target element y may belong. This bin is dictated by the first interpolation step, with index j_1, satisfying

$$j_1=\left\lfloor\frac{(y-a)f_1(n_1)}{(b-a)}\right\rfloor+1. \tag{15}$$

Hence, BIN(j_1) corresponds to the subinterval

$$\left[a+(j_1-1)\frac{b-a}{f_1(n)},\ a+j_1\frac{b-a}{f_1(n)}\right)=[a_{j_1},b_{j_1})\subset[a,b]$$

for which it holds that $a_{j_1}\leq y<b_{j_1}$. The interval $[a_{j_1},b_{j_1})$ contains $n_{j_1}=O(n^\delta)$ elements, and is further partitioned into $f_1(n_{j_1})=O(n^{\delta\alpha})$ equally spaced bins BIN($j_1,1$), ..., BIN($j_1,f_1(n_{j_1})$); see Fig. 2. Let BIN(j_1,j_2) be the bin of the 2nd Layer, where the target element y may belong. This bin is dictated by the second interpolation step, with index $j_2=1,\ldots,f_1(n_{j_1})$, satisfying

$$j_2=\left\lfloor\frac{(y-a_{j_1})f_1(n_{j_1})}{(b_{j_1}-a_{j_1})}\right\rfloor+1. \tag{16}$$

Now BIN(j_1,j_2) corresponds to the interval

$$\left[a_{j_1}+(j_2-1)\frac{b_{j_1}-a_{j_1}}{f_1(n_{j_1})},\ a_{j_1}+j_2\frac{b_{j_1}-a_{j_1}}{f_1(n_{j_1})}\right)=[a_{j_1,j_2},b_{j_1,j_2})\subset[a_{j_1},b_{j_1})\subset[a,b]$$

[6] That is, we know the position of the element to be deleted or next to which the new element will be inserted. This can be accomplished by a search operation.

This process continues recursively until a leaf bin that contains y is reached. Then, we employ the searching procedure of the balanced search tree [16,30] that implements the leaf bin and we locate y.

Updating the LIST affects only its leaf bins. The update operation at a leaf bin is position given and its implementation is determined by the constant update search tree in the leaf-bin. This means that in the worst-case many elements could land in a leaf bin deteriorating the search time. However, the number of elements in the leaf bins are manageable in expectation as we show in Section 3.2. In addition, incremental global rebuilding [36] is employed on the LIST to maintain the size of its leaf bins on the long run.

This incremental global rebuilding [36] is spread over the updates performed on the LIST. Particularly, let S_0 be the set of stored elements at the most recent rebuilding and let U denote the set of updates (insertions/deletions of elements) since the most recent rebuilding. When the fraction $|U|/|S_0|$ exceeds some predefined constant, the rebuilding of the LIST is initiated by incrementally building a new LIST on its current set of elements that will replace eventually the old LIST. This rebuilding is spread over the next $O(|S_0|)$ updates.

Finally, we describe a method to construct a LIST on a sorted sequence of n given elements. This is necessary for the global rebuilding technique, since it requires an explicit way to construct the data structure on a set of elements.

Let S be the set of n elements on which the LIST will be built. We partition S into $m = n/\log n$ buckets and let the smallest element of each bucket be its *representative*. Let S' be the set of representatives. We build top-down (from the root towards the leaves) a LIST on S' as follows. In the root of the LIST, we partition the $[a, b]$ interval into $f_1(m) = m^\alpha$ subintervals, each corresponding to a bin. Then, we distribute all elements in S' among these bins and this procedure continues recursively. This distribution is carried out by a simple scan of the sorted sequence of elements in S'. In each recursion, we construct the array corresponding to the bins (nodes) filling it up with the lower bound of the corresponding subinterval as well as with the number of elements (representatives) that lie in this bin. The recursive layering will end either when the path on the LIST from the root to some node reaches length $\log m$, or when a node (bin) has $\log^{1/\delta} m$ representatives. Finally, each element of S' in a leaf bin maintains a pointer to the corresponding bucket of elements of S that contains $\log n$ in total elements. Note, that these buckets contain all elements of S, since LIST is leaf-oriented.

3.2. Analysis of the new data structure

The careful reader should have noticed that the endpoints selected as representatives in each subtree are independent of the particular characteristics of the input distribution μ, thus confronting the weakness of the constructions in all the previous approaches. This crucial randomness invariance property of our new data structure is proved in the next lemma.

Lemma 1. *Consider an arbitrary bin BIN$(j_1, \ldots, j_i)$ and let $[a_{j_1,\ldots,j_i},\ b_{j_1,\ldots,j_i})$ be its corresponding subinterval of the ith* Layer *of bins. Then, the $n_{j_1,\ldots,j_i}$ elements in BIN$(j_1, \ldots, j_i)$ are μ-randomly distributed in the subinterval $[a_{j_1,\ldots,j_i},\ b_{j_1,\ldots,j_i})$.*

Proof of Lemma 1. Consider the 1st Layer of the $f_1(n)$ distinct bins BIN(j_1), $j_1 = 1, \ldots, f_1(n)$. The interval $[a, b]$ is deterministically partitioned into $f_1(n)$ equal-length subintervals I_{j_1}, $j_1 = 1, \ldots, f_1(n)$, according to the function f_1 which is independent of the distribution μ. Each subinterval I_{j_1} corresponds to BIN(j_1). Each element $X \in S$, where $n = |S|$ is the number of elements currently stored in the data structure, is μ-randomly distributed over $[a, b]$, and belongs to an arbitrary subinterval I_{j_1} independently and with probability $p_{j_1} = \Pr[X \in I_{j_1}]$. The number of elements n_{j_1} currently stored in BIN(j_1) is a Binomial[7] $B(n, p_{j_1})$ random variable, $j_1 = 1, \ldots, f_1(n)$, sharply concentrated to its expected value. Now, conditioning on the number n_{j_1} of elements that BIN(j_1) contains, each element X in it is distributed over subinterval I_{j_1} according to the *conditional*, on this subinterval, probability density $\Pr[X = x \mid X \in I_{j_1}] = \frac{\Pr[X=x \,\cap X \in I_{j_1}]}{\Pr[X \in I_{j_1}]} = \frac{\mu(x)}{p_{j_1}}$, since $\{X = x\} \subset \{X \in I_{j_1}\}$.

Given n_{j_1}, BIN(j_1) is further partitioned into $f_1(n_{j_1})$ distinct bins. Each subinterval I_{j_1,j_2} corresponds to BIN(j_1, j_2), $j_2 = 1, \ldots, f_1(n_{j_1})$. Let $p_{j_1,j_2} = \Pr[X \in I_{j_1,j_2}]$. Each element X of the n_{j_1} possible is μ-randomly distributed over I_{j_1}, and belongs to an arbitrary subinterval I_{j_1,j_2} independently and with probability $p_{j_2|j_1} = \Pr[X \in I_{j_1,j_2} \mid X \in I_{j_1}] = \frac{\Pr[X \in I_{j_1,j_2} \cap X \in I_{j_1}]}{\Pr[X \in I_{j_1}]} = \frac{p_{j_1,j_2}}{p_{j_1}}$, since $\{X \in I_{j_1,j_2}\} \subset \{X \in I_{j_1}\}$. The size n_{j_1,j_2} of BIN(j_1, j_2) is a Binomial $B(n_{j_1},\ p_{j_2|j_1})$ random variable sharply concentrated to its expected value. Now, conditioning on the number n_{j_1,j_2} of elements that BIN(j_1, j_2) contains, each element X in it is distributed over the subinterval I_{j_1,j_2} according to the *conditional* on this subinterval probability density $\Pr[X = x \mid X \in I_{j_1,j_2} \cap I_{j_1}] = \Pr[X = x \mid X \in I_{j_1,j_2}]$, since $\{X = x\} \subset \{X \in I_{j_1,j_2}\} \subset \{X \in I_{j_1}\}$.

Applying this argument inductively, we conclude that each element in an arbitrary bin remains μ-random given its corresponding subinterval. □

The next theorem shows that w.h.p. all bins have the desired upper bound on the number of elements within them, i.e., a child bin has size at most $f_2(t)$ where t is the number of elements of its father bin.

[7] The Binomial distribution $B(n, p)$ is the discrete probability distribution of the number of "success"-es in n independent experiments, when per experiment the outcome is "success" with probability p, or "failure" otherwise. Let X be a random variable denoting the number of "success"-es. Then, $\Pr[X = i] = \binom{n}{i} p^i (1-p)^{(n-i)}$, and $E[X] = np$.

Theorem 1. *Consider an arbitrary bin* $\text{BIN}(j_0,\ldots,j_i)$ *of the* i*-th* LAYER *of bins and let* $\text{BIN}(j_0,\ldots,j_{i-1})$ *be its father in the tree structure. Then for* $\tau = 3$*,*

$$\Pr\left[n_{j_0,\ldots,j_i} > \tau\cdot(n_{j_0,\ldots,j_{i-1}})^{\delta}\right] < \frac{1}{n}$$

which tends to zero as $n\to\infty$*.*

Proof of Theorem 1. By Lemma 1, there are $n_{j_0,\ldots,j_{i-1}}$ elements *μ-randomly distributed* into the subinterval $[a_{j_0,\ldots,j_{i-1}}, b_{j_0,\ldots,j_{i-1}})$, which is partitioned into $f_1(n_{j_1,\ldots,j_{i-1}})$ equal-length subintervals one of which corresponds to $\text{BIN}(j_0,\ldots,j_i)$. We prove that the probability, that $\text{BIN}(j_0,\ldots,j_i)$ receives more than $\tau f_2(n_{j_1,\ldots,j_{i-1}})$ elements, approaches 0 exponentially fast on the number of elements $n_{j_1,\ldots,j_{i-1}}$.

By the smoothness properties (1) and (2), $\text{BIN}(j_0,\ldots,j_i)$ receives a μ-random element independently of all other elements with probability $\leq \frac{f_2(n_{j_0,\ldots,j_{i-1}})}{n_{j_0,\ldots,j_{i-1}}}$ (for simplicity, we drop the constant β). Therefore, this bin is expected to contain $f_2(n_{j_0,\ldots,j_{i-1}}) = (n_{j_0,\ldots,j_{i-1}})^{\delta}$ elements. More specifically, the number $n_{j_0,\ldots,j_i}$ of elements it receives, is a random variable statistically dominated by the Binomial distribution

$$B\left(n_{j_0,\ldots,j_{i-1}},\ \frac{f_2(n_{j_0,\ldots,j_{i-1}})}{n_{j_0,\ldots,j_{i-1}}}\right) = B\left(n_{j_0,\ldots,j_{i-1}},\ \frac{(n_{j_0,\ldots,j_{i-1}})^{\delta}}{n_{j_0,\ldots,j_{i-1}}}\right).$$

We upper bound the probability that $\text{BIN}(j_0,\ldots,j_i)$ deviates significantly more than its expected load $(n_{j_0,\ldots,j_{i-1}})^{\delta}$. This is a standard Chernoff bound argument.

$$\Pr\left[n_{j_0,\ldots,j_i} > \tau\cdot(n_{j_0,\ldots,j_{i-1}})^{\delta}\right] < \left(\frac{e^{\tau}}{(1+\tau)^{1+\tau}}\right)^{(n_{j_0,\ldots,j_{i-1}})^{\delta}}$$

To prove that there does not exist a bin that has this property among all bins in the LIST we employ the union bound. First, we compute a crude upper bound on the number of bins in the LIST. Since one of the stopping conditions require that a bin must have $\log^{1/\delta} n \geq \log n$ elements, it follows that the maximum number of bins per layer is $\frac{n}{\log n}$ so that the recursive process may continue to the next layer. In addition, the other stopping condition requires that the maximum depth of a node is $\log n$, which means that the maximum number of layers is that much. Finally, each bin may have at most n^{α} children (this is true only for the root). All in all, a crude upper bound on the number of bins in a LIST of n elements is $\frac{n}{\log n} n^{\alpha}\log n$. For ease of exposition and since $\alpha < 1$, we choose n^2 as an upper bound on the number of bins, which suffices for the purpose of this proof. (Note that most of these bins are empty and are not stored. This is made clear in Lemma 3 that provides the actual bounds on the construction time as well as on the space usage.)

By the union bound we get that the probability that there exists a bin $\text{BIN}(j_0,\ldots,j_i)$ with more than $\tau\cdot(n_{j_0,\ldots,j_{i-1}})^{\delta}$ elements is

$$\Pr\left[\exists\ \text{BIN}(j_0,\ldots,j_i) : n_{j_0,\ldots,j_i} > \tau\cdot(n_{j_0,\ldots,j_{i-1}})^{\delta}\right] < n^2\left(\frac{e^{\tau}}{(1+\tau)^{1+\tau}}\right)^{(n_{j_0,\ldots,j_{i-1}})^{\delta}}$$

The minimum value of $n_{j_0,\ldots,j_{i-1}}$ is $\log^{1/\delta} n$. In addition, by choosing $\tau = 3$ we have that $\frac{e^3}{4^4} < 1/8$, and hence we get the following simpler bound

$$\Pr\left[\exists\ \text{BIN}(j_0,\ldots,j_i) : n_{j_0,\ldots,j_i} > \tau\cdot(n_{j_0,\ldots,j_{i-1}})^{\delta}\right] < \frac{1}{n}$$

which tends to zero as $n\to\infty$.

Consequently, the probability to encounter at least one bin with *large* size vanishes polynomially fast. This completes the proof of the theorem. □

We now turn to establish the search bound of our data structure. We show that the desired bound holds with high probability.

Lemma 2. *The searching time for a target element* y *in LIST is proportional to its height and both are w.h.p.* $O(\log\log n)$.

Proof of Lemma 2. Assume that at a time instant, there is a total of n elements stored in the data structure. The search for y requires a single interpolation search per layer until we have reached a leaf bin in which we search for y. This search naturally defines a path from the root to the leaf bin containing y (or its predecessor). Let this bin be $\text{BIN}(j_1,\ldots,j_k)$.

Then, by Theorem 1, it holds w.h.p. that for all nodes on the path $n_{j_0,\ldots,j_i} = O\left((n_{j_1,\ldots,j_{i-1}})^{\delta}\right)$, $i = 1,\ldots,k$. As a result, it holds that $n_{j_0,\ldots,j_i} = O\left(n^{\delta^i}\right)$.

It follows that after at most $k = O(\log_{1/\delta} \log n)$ interpolation steps, we can find a *sparse* enough bin denoted as $\text{BIN}(j_1, \ldots, j_k)$ with load of at most $\log^{1/\delta} n$. Clearly, k is the height of LIST. Implementing each such leaf bin as a constant update balanced search tree [16,30], we can find the target element y within $O\left(\log\left(\log^{1/\delta} n\right)\right) = O(\log\log n)$ time. Since each of the totally k interpolation steps requires a single step to compute the bin in which y lies, the overall search time is $O(k) + O(\log\log n) = O(\log\log n)$. □

The following lemma states the complexities for the construction and space consumption of the LIST.

Lemma 3. *A LIST on a sorted sequence of μ-random elements of cardinality n can be constructed in $O(n)$ worst-case time occupying $O(n)$ space.*

Proof of Lemma 3. Constructing the sorted sequence S' of representatives from S requires $O(n)$ time, since it is just a linear scan of the sorted sequence S. Recall that the size of S' is $m = n/\log n$. Distributing the representatives in S' into bins in the first layer requires linear time since S' is sorted. The same holds in all other layers as well. This is because in each layer there cannot be more than m bins, since all other bins should be empty and the total number of representatives to be distributed is m. Thus, the total time to distribute the representatives in a layer is $O(m)$. By the stopping condition, the height of the LIST is at most $\log m$. As a result, the total construction times is $O(m \log m)$. Substituting m we get that the total construction time is $O(n)$. Finally, since the used space cannot be larger than the construction time, we also get that the space usage is $O(n)$. □

We are now ready for the main result of this section.

Theorem 2. *Consider a set of n (not necessarily distinct) elements produced by a sequence of μ-random insertions and random deletions, where μ is a (n^α, n^δ)-smooth density, for any arbitrary $\alpha, \delta \in (0,1)$ with $\alpha + \delta \geq 1$. Then, there exists a dynamic interpolation search data structure which achieves w.h.p. $O(\log\log n)$ search time. The space usage of the data structure is $\Theta(n)$, the worst-case update time (position given) is $O(1)$, and the worst-case search time is $O(\log n)$.*

Proof of Theorem 2. The LIST is maintained by incrementally performing global reconstructions [36]. Let S_0 be the set of stored elements at the most recent reconstruction. By Lemma 3, this reconstruction takes $c|S_0|$ worst-case time, for some appropriate constant c.

The insertions and deletions of elements are carried out between consecutive reconstructions. To insert/delete an element, we insert/delete it to/from the appropriate leaf bin of the LIST. Since the linear work spent during reconstruction can be spread out in the updates in such a way that a rebuilding cost of $O(1)$ is spent at each update, and since the leaf bins can be updated in $O(1)$ worst-case time (given the update position), we conclude that the worst-case update time cost in maintaining the LIST is $O(1)$.

The search procedure is composed of two phases. The first one refers to the traversal of a root to a leaf-bin path, which by Lemma 2 has w.h.p. length equal to $O(\log\log n)$. The second phase is the search inside the leaf-bin. Assume that the leaf-bin has size b. Then, the balanced search tree guarantees that the time complexity for the search in the leaf-bin is $O(\log b)$. By the proof of Lemma 2 and the respective stopping condition, b is $O\left(\log^{1/\delta} n\right)$ and as a result the time needed to search in the leaf-bin is $O(\log\log n)$ w.h.p. Thus, the search procedure can be carried out in $O(\log\log n)$ w.h.p.

Finally, in the worst case, the size of a leaf-bin can be as high as n, and thus the worst-case complexity for searching in a leaf-bin will be as high as $O(\log n)$. Taking into account that the maximum root-to-leaf-bin path is $\log n$ the worst-case time complexity follows. □

It is easy to verify that every part of our analysis remains valid if we replace the function $f_1(n) = n^\alpha$ with the function $f_1(n) = \frac{n}{(\log\log n)^{1+\varepsilon}}$, where $\varepsilon > 0$. Hence, our structure can handle within the same time and space complexities the larger class of $\left(\frac{n}{(\log\log n)^{1+\varepsilon}}, n^\delta\right)$-smooth densities, yielding the following corollary to Theorem 2.

Corollary 1. *Consider a set of n (not necessarily distinct) elements produced by a sequence of μ-random insertions and random deletions, where the density μ is $\left(\frac{n}{(\log\log n)^{1+\varepsilon}}, n^\delta\right)$-smooth, for any arbitrary $0 < \delta < 1$ and $\varepsilon > 0$. Then, there exists a dynamic interpolation search data structure which achieves w.h.p. $O(\log\log n)$ search time. The space usage of the data structure is $\Theta(n)$, the worst-case update time (position given) is $O(1)$, and the worst-case search time is $O(\log n)$.*

3.3. Comparison

The difference of our data structure with those in [3,32] is in the absence of REP arrays. These arrays guarantee that when we move to a child of a node whose subtree contains N nodes, then this child node will be the root of a subtree

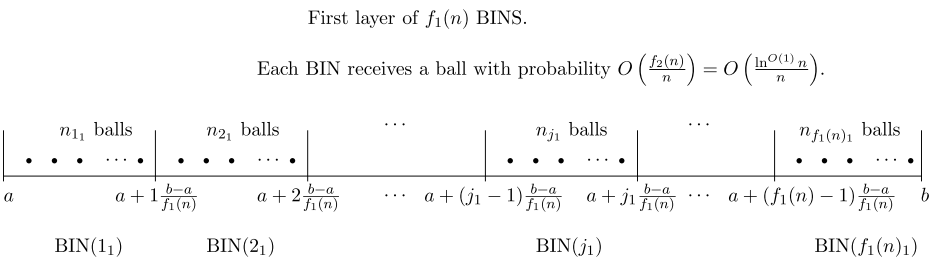

Fig. 3. The unique interpolation step indicates in $O(1)$ time the index $j_1 = \lfloor \frac{(y-a)f_1(n)}{(b-a)} \rfloor + 1$ of BIN(j_1) that element y may belong. Given that μ is $\left(\frac{n}{g}, \ln^{O(1)} n\right)$-smooth, no bin gets w.h.p. more than $\log^{O(1)} n$ elements.

containing $\sqrt{N}$ nodes. In our case, this is not guaranteed (it is easy to come up with a setting where all elements are in a very small region and thus the height of our tree structure without pruning would be large). However, assuming that the input elements are generated by a smooth distribution, it is *very unlikely* that this bad scenario will happen, since we prove that the height of our tree structure is doubly logarithmic with high probability. Our data structure is in a sense "similar" to other data structures that partition the space (e.g., quadtrees). Indeed, our structure partitions the universe until each region has a bounded number of elements. On the other hand, the use of REP arrays allows for a partition according to the number of elements (like e.g., in range trees), thus guaranteeing that each partition has geometrically less elements.

4. Constant search time

In this section, we consider a variant of our data structure consisting of only *one* LAYER of $f_1 = f_1(n)$ bins indexed as BIN$(1), \ldots,$ BIN$(f_1(n))$, and show that it achieves $O(1)$ expected search time for a wide class of smooth distributions.

Assume that the structure contains n elements. For some constant $0 < r < 1$, we study an arbitrary sequence of rn insert (or delete) operations on this structure. In each operation, $j = 1, \ldots, rn$, a new element $X \in [a, b]$ obeying an unknown (discrete or continuous) $(f_1(n), f_2(n)) = \left(\frac{n}{g}, \ln^{O(1)} n\right)$-smooth distribution μ is inserted or a random existing element is deleted, where $g \geq 1$ is a constant. The structure always contains $\Theta(n)$ elements, since after the rn-th update we reconstruct the structure to initiate a new sequence of updates in the same manner as we did in Section 3.1.

The class μ of $\left(\frac{n}{g}, \ln^{O(1)} n\right)$-smooth distributions includes that of bounded ($(n, 1)$-smooth) densities, for which $O(1)$ expected search time was known [3], as well as all those for which a $o(\log\log n)$ expected search time could be achieved [3]. For instance, the density $\mu[0, 1](x) = -\ln x$ is $(n/(\log^* n)^{1+\varepsilon}, \log^2 n)$-smooth, and an expected search time complexity of $\Theta(\log^* n)$ was given in [3]. Notice that by definition the $(n/(\log^* n)^{1+\varepsilon}, \log^2 n)$-smooth class of distributions is contained within the $(n, \log^2 n)$-smooth class. Our results in this section imply $O(1)$ search time for all the aforementioned densities.

We shall distinguish between discrete and continuous probability distributions, although the results in the two cases do not differ too much, since the former case has a simpler and more direct approach than the latter. In both cases, we will make use of a search tree data structure called *q*-heap* [59], implemented on a unit-cost RAM with a word length of w bits – which assumes that multiplication and the standard AC^0 operations (addition, subtraction, comparison, bitwise Boolean operations and shifts) are performed in constant time on $O(w)$-bit operands. 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.

4.1. Discrete probability distributions

The discrete $\left(\frac{n}{g}, \ln^{O(1)} n\right)$-smooth distribution may produce elements with measurable probability of collisions, that is, not necessarily distinct elements. This implies that the elements are produced with some (sufficiently large but) fixed precision, and therefore we can safely assume that we work under the word RAM model of computation in this case.

Theorem 3. *There exists a dynamic interpolation search data structure, which in the word RAM model attains $O(1)$ search time w.h.p. for μ-random insertions and random deletions, where μ is a discrete $\left(\frac{n}{g}, \ln^{O(1)} n\right)$-smooth density and n denotes the number of stored elements for an arbitrary constant $g \geq 1$. The space usage of the data structure is $\Theta(n)$, and the worst-case update time is $O(1)$.*

Proof of Theorem 3. Our approach is very simple. It follows straightforwardly from the proof of Theorem 1 in that during each step $j = 1, \ldots, rn$, *no bin* of the 1st LAYER gets w.h.p. more than $\log^{O(1)} n$ elements. That is, the whole LIST data structure reduces to a single LAYER; see Fig. 3. In particular, during each step j, our goal is to search w.h.p. in $O(1)$ time for an arbitrary element y.

Each μ-random element X is stored in BIN(j_1), if $X \in \left[a+(j_1-1)\frac{b-a}{f_1(n)},\ a+j_1\frac{b-a}{f_1(n)}\right] \subseteq [a,\ b],\ \ j_1=1,\ldots,f_1(n)$. According to Ineq. (1), X is stored independently in BIN(j_1) with probability p_{j_1}, which is $p_{j_1}=O\left(\frac{f_2(n)}{n}\right)=O\left(\frac{\ln^{O(1)}n}{n}\right)$, $j_1=1,\ldots,f_1(n)$. Then, the number of elements n_{j_1} stored in BIN(j_1) is a Binomial $B\left(O(n),p_{j_1}\right)$ random variable, sharply concentrated to its expected value, which is $O\left(np_{j_1}\right)=O\left(\ln^{O(1)}n\right)$. This means that the occupancy number n_{j_1} of BIN(j_1) may deviate by a *constant* factor from its expectation with exponentially small probability. Since there are only $f_1(n)$ bins, which are polynomially many, then – exactly as we did in the proof of Theorem 1 – we upper bound the probability that at least one bin gets more than $\log^{O(1)}n$ elements and show that it vanishes as n approaches infinity.

We conclude that during each step $j=1,\ldots,rn$, no bin gets w.h.p. more than $\log^{O(1)}n$ elements. Each BIN(j_1), $j_1=1,\ldots,f_1(n)$ is implemented as a q^*-heap and as a result we can search for element y in it in $O(1)$ time. We can also determine in O(1) time the bin BIN(j_1) that y may belong using the interpolation relation $j_1=\left\lfloor\frac{(y-a)f_1(n)}{(b-a)}\right\rfloor+1$. Taking into account that updates within a q^*-heap take $O(1)$ time and that the structure is incrementally reconstructed every rn updates, we have established the theorem. □

4.2. Continuous probability distributions

In the case where the $\left(\frac{n}{g},\ln^{O(1)}n\right)$-smooth distribution is continuous, we follow the assumption adopted throughout this paper, i.e., that the produced elements are distinct (and thus have arbitrary precision), and hence we work on the unit-cost real RAM model.

Theorem 4. *There exists a dynamic interpolation search data structure, which in the unit-cost real RAM model attains $O(1)$ expected search time or $O(\phi(n))$ search time w.h.p. for μ-random insertions and random deletions, where μ is a continuous $\left(\frac{n}{g},\ln^{O(1)}n\right)$-smooth density for a constant $g\geq 1$, n denotes the number of stored elements, and $\phi(n)$ is any slowly growing function of n. The space usage of the data structure is $\Theta(n)$, and the worst-case update time is $O(1)$.*

Proof of Theorem 4. Our approach in this case follows exactly the one described in Section 4.1, with the following exception regarding the treatment of the actual elements. Since the actual elements are (at least theoretically) numbers of infinite precision, we cannot use a q^*-heap to store them. To store and manipulate the actual elements, we follow the approach in [29]. Each BIN(j_1), $j_1=1,\ldots,f_1(n)$, is again implemented as a q^*-heap, but now it stores a truncated version of the actual elements along with pointers to a sorted list L that contains the actual elements. In particular, for each actual element x, its truncated version $\tilde{x}$, up to a sufficiently large precision, is stored in some bin 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 reals $x_1\neq\ldots\neq x_k$ in L coincide when truncated to $\widetilde{x}$. We call these k real elements the *chain of* $\widetilde{x}$ in L, and we maintain pointers from each such x_i, $1\leq i\leq k$, to $\widetilde{x}$, while it suffices to maintain just one pointer from $\widetilde{x}$ to one of the reals x_i. In other words, we treat bins as an "indexing structure" to the actual elements. The update and search operations are now handled as follows.

Searching for element y involves firstly locating the bin BIN(j_1) that y may belong. This is done in $O(1)$ worst-case time using the interpolation relation $j_1=\left\lfloor\frac{(y-a)f_1(n)}{(b-a)}\right\rfloor+1$. Then, a search within BIN(j_1) is carried out that returns, in $O(1)$ worst-case time (since each bin is implemented as a q^*-heap), an element $\widetilde{z_0}$ which is equal to $\tilde{y}$ (the truncated version of y), or its predecessor. Recall that there may be several actual elements $z_1\neq\cdots\neq z_k\neq z_0$ (the chain of $\widetilde{z_0}$) that have the same truncated version $\widetilde{z_0}$, i.e., $\widetilde{z_1}=\widetilde{z_2}=\cdots=\widetilde{z_k}=\widetilde{z_0}$. The elements z_i can be identified in $O(k)$ time by following the pointer from $\widetilde{z_0}$ to one of those elements, say z_j, and then locating (one by one) the elements z_i for which $\widetilde{z_i}=\widetilde{z_j}=\widetilde{z_0}$, $0\leq i\leq k$. For each $0\leq i\leq k$, z_i is compared to y and either a match is found or the largest element less than y is returned (predecessor).

Deleting an element y is carried out as follows. First, we locate y by calling the search operation. We delete y from L in $O(1)$ worst-case time, by manipulating a constant number of pointers, and look at its predecessor and successor elements in L. If any of these two elements point to $\widetilde{y}$ in bin BIN(j_1) (i.e., if any of these elements points to the same truncated version as y did), then we do nothing. Otherwise, we follow the pointer to $\widetilde{y}$ and remove $\widetilde{y}$ from BIN(j_1) in $O(1)$ worst-case time, since the bins are organized as q^*-heaps.

Inserting an element y is carried out similarly. First, we call the search procedure for y, which returns an element x in L next to which y will be inserted. Then, y is inserted in L next to x in $O(1)$ worst-case time, since only a constant number of pointers need to be manipulated. Then, by using the pointer of x to $\widetilde{x}$, we determine the bin into which $\widetilde{y}$ will be inserted. This insertion is carried out in $O(1)$ worst-case time, due to the q^*-heap organization of the bins. Finally, a pointer is established from $\widetilde{y}$ to y.

It is clear from the above description that the time of a search operation is dominated by the expected length of the chain of an element, and that insertions and deletions take $O(1)$ worst-case time, position given.

In [29], it is proved that for elements drawn according to a general class of smooth distributions, which includes the class of $\left(\frac{n}{g},\ln^{O(1)}n\right)$-smooth densities (for a constant $g\geq 1$): (i) the length of an arbitrary chain is $O(1)$ in expectation, or $O(\phi(n))$ w.h.p., where $\phi(n)$ is any slowly growing function of n (e.g., the inverse Ackermann function [49]); (ii) $O(\log n)$

bits suffice to represent the truncated version of any actual element – therefore, we can employ a q^*-heap with a word length of $O(\log n)$ bits for the representation of the truncated elements. This implies that the search operation takes $O(1)$ expected time, or $O(\phi(n))$ time w.h.p. Note that there is a trade-off with respect to function $\phi(n)$. On the one hand, the length of the chain (and thus the efficiency of the search procedure) is getting larger with $\phi(n)$. On the other hand, the probability that its length is bounded by $\phi(n)$ is of the form $O(1-1/\phi(n))$, which means that this probability gets higher with $\phi(n)$. Thus, the choice of $\phi(n)$ is based solely on this trade-off. The preceding discussion establishes the theorem. □

5. Handling power law and binomial distributions

As shown in Section 2, the efficiency of an arbitrary interpolation step dictating a subtree rooted at a node v highly relies in how sparsely the total of n_v elements belonging to the subtree are distributed in its associated subinterval $[a_v, b_v]$. This sparsity fails for power law and binomial distributions, as we show in this section. Nevertheless, we are able to show that a search time of $O(\log\log n)$, with high probability, can be achieved for these distributions.

Similarly to Section 4, we assume that the data structure contains initially n elements drawn from a universe $\mathcal{U}$. After a sequence of rn updates, for a constant $0 < r < 1$, the structure is replaced by a new one (re-initialized) which is constructed incrementally (as described in Section 3.1). In this way, the space usage of the structure remains always $\Theta(n)$. Note that n constantly changes between initializations due to the updates but only by a constant factor. The additional cost to updates due to the incremental reconstruction is $O(1)$ in the worst-case.

5.1. Power law distribution

We first study the case where the subtree located after an arbitrary interpolation step stores elements distributed in its associated subinterval according to a power law distribution. A random element X is drawn according to a *power law* distribution [33, Sec. 2] with *unknown* parameters $c, \gamma > 0$, when $\Pr[X \geq x] \sim c \cdot x^{-\gamma}$.

Let $0 < \alpha < 1$ be any constant. Consider the subsets of elements $I_1 = \{1, \ldots, n^\alpha - 1\} \subseteq \mathcal{U}$ and $I_2 = \mathcal{U} \setminus I_1$. The probability mass of the subset of elements in I_1 equals

$$\Pr[X \in I_1] = 1 - \Pr[X \in I_2] = 1 - \Pr[X \geq n^\alpha] = 1 - \frac{c}{(n^\alpha)^\gamma} = \omega\left(\frac{n^\delta}{n}\right) \tag{17}$$

with $0 < \delta < 1$. According to Ineq. (2), I_1 is *not* (n^α, n^δ)-smooth, for any constant $0 < \alpha < 1$, hence interpolation search is not suitable on I_1. Nevertheless, we are able to achieve a search time of $O(\log\log n)$ w.h.p. for power law distributions, as we show in the next theorem.

Theorem 5. *Consider a set of n (not necessarily distinct) elements produced by a sequence of power law insertions and random deletions. Then, there exists a dynamic interpolation search data structure which achieves w.h.p. $O(\log\log n)$ search time. The space usage of the data structure is $\Theta(n)$ and the worst-case update time (position given) is $O(1)$.*

Proof of Theorem 5. At this point, it is intuitively helpful to assume that parameters $c, \gamma > 0$ of the power law distribution are *known* to us. Later, we exhibit the robustness of the structure by removing this assumption.

Let the subsets of elements $I_1 = \{1, \ldots, n^\alpha - 1\} \subseteq \mathcal{U}$ (for any constant $0 < \alpha < 1$) and $I_2 = \mathcal{U} \setminus I_1$, as defined above.

Observe, that $I_2 = \mathcal{U} \setminus I_1$ can be arbitrarily sparse, as a function of α. Since $\Pr[X \in I_2] = \Pr[X \geq n^\alpha] = \frac{c}{(n^\alpha)^\gamma}$, we set $\alpha = \frac{1}{\gamma \ln(n)} \ln\left(\frac{cn}{\ln(n)}\right) \to \frac{1}{\gamma}$ and as $n \to \infty$ we get $\Pr[X \in I_2] = \Pr[X \in \{n^{\frac{1}{\gamma}}, \ldots, |\mathcal{U}|\}] = \frac{\ln n}{n}$. We call $n^\alpha \to n^{\frac{1}{\gamma}}$ (as $n \to \infty$) the *splitting element* of $\mathcal{U}$. That is, a random element $X \in \mathcal{U}$ falls above the splitting element into $I_2 = \{n^{\frac{1}{\gamma}}, \ldots, |\mathcal{U}|\}$ with probability $\frac{\ln n}{n}$, or, it falls below the splitting element into $I_1 = \{1, \ldots, n^{\frac{1}{\gamma}} - 1\}$ with probability $1 - \frac{\ln n}{n}$. Observe that $|I_1| = |\{1, \ldots, n^\alpha - 1\}| \to |\{1, \ldots, n^{\frac{1}{\gamma}} - 1\}| = n^{\frac{1}{\gamma}} - 1$, as $n \to \infty$. That is, if the target element y belongs to I_1, then it can be searched amongst $n^\alpha - 1 \to n^{\frac{1}{\gamma}} - 1$ possible elements. Therefore, if $y \in I_1$, we can employ the van Emde Boas structure [54,55], which yields a searching time complexity $O\left(\log\log\left(n^{\frac{1}{\gamma}}\right)\right) = O(\log\log n)$. On the other hand, a random element $X \in \mathcal{U}$ is possible to belong to $I_2 = \{n^{\frac{1}{\gamma}}, \ldots, |\mathcal{U}|\}$ with probability $\frac{\ln n}{n}$. Then, we can view I_2 as a bucket, where each of the $rn = O(n)$ update operations is associated to an element in this bucket with probability $\frac{\ln n}{n}$. Therefore, during all the $rn = O(n)$ update operations the load of this bucket is a Binomial $B\left(rn, \frac{\ln n}{n}\right)$ random variable and w.h.p. remains concentrated to its expectation $\Theta(\ln n)$. In turn, we can employ a q^*-*heap* for this bucket that achieves $O(1)$ time per update operation.

We now turn to the case where the parameters $c, \gamma > 0$ are *unknown* to us. We show that the initialization phase of the data structure (consisting of n random power law insertions) helps to approximate the above subsets defined by the splitting element $n^\alpha \to n^{\frac{1}{\gamma}}$, without affecting considerably the efficiency of the operations.

We sort increasingly the values of the n random elements inserted during the initialization phase, that is, $x_{(1)} \leq x_{(2)} \leq \ldots \leq x_{(n)}$ breaking ties arbitrarily. Then, we select as an approximation of the splitting element $n^{\alpha} \to n^{\frac{1}{\gamma}}$ the element

$$x^* = x_{(n-\ln n)}$$

That is, x^* appears in the above ordering exactly $\ln n$ positions before the least frequent element $x_{(n)}$. In this way, we set as $I_1^* = \{1, \ldots, x^* - 1\}$ and $I_2^* = \mathcal{U} \setminus I_1^* = \{x^*, \ldots, |\mathcal{U}|\}$. Now, the argument boils down to show that w.h.p. $\Pr[X \in I_2^*] = \Theta\left(\frac{\ln n}{n}\right)$.

Let $w(n) = \frac{\ln n}{n}$ and $p(n) = \Pr[X \in I_2^*]$. When n random elements are drawn, the number of elements in I_2^* is a Binomial $B(n, p(n))$ random variable. Hence, the probability that $w(n)n = \ln n$ random elements are in I_2^* is[8]:

$$\binom{n}{w(n)n} p(n)^{w(n)n} (1 - p(n))^{(1-w(n))n} \sim \left[\left(\frac{p(n)}{w(n)}\right)^{w(n)} \left(\frac{1-p(n)}{1-w(n)}\right)^{1-w(n)} \right]^n \tag{18}$$

Eq. (18) is a convex function of $w(n), p(n)$ and its maximum value equals 1 iff $w(n) = p(n)$. Therefore (18) approaches 0, exponentially in n, *iff*

$$p(n) = \omega(w(n)) \Leftrightarrow \Pr[X \in I_2^*] = \omega\left(\frac{\ln n}{n}\right)$$

or

$$p(n) = o(w(n)) \Leftrightarrow \Pr[X \in I_2^*] = o\left(\frac{\ln n}{n}\right)$$

We conclude that

$$\Pr[X \in I_2^*] = \Theta\left(\frac{\ln n}{n}\right). \text{ Since } I_1^* = \mathcal{U} \setminus I_2^*, \text{ then } \Pr[X \in I_1^*] = 1 - \Theta\left(\frac{\ln n}{n}\right)$$

Then, we can view I_2^* as a bucket, where each of the $rn = O(n)$ update operations is associated to an element in this bucket with probability $\Theta\left(\frac{\ln n}{n}\right)$. Therefore, during all the $rn = O(n)$ update operations, the load of this bucket is a Binomial $B\left(rn, \Theta\left(\frac{\ln n}{n}\right)\right)$ random variable and w.h.p. remains concentrated to its expectation $\Theta(\ln n)$. In turn, we can employ a q^**-heap* for this bucket that achieves $O(1)$ time per update operation.

It only remains to show that the size of $I_1^* = \{1, \ldots, x^* - 1\}$ is close to $|I_1| \to n^{\frac{1}{\gamma}} - 1$, so we can still employ a van Emde Boas tree [54,55]. But this is easy, since w.h.p. it holds $\Pr[X \in I_2^*] = \Theta\left(\frac{\ln n}{n}\right)$ which gives $\Pr[X \in I_2^*] = \Pr[X \geq x^*] = \frac{c}{(x^*)^{\gamma}} = \Theta\left(\frac{\ln n}{n}\right)$, with solution $x^* = n^{\frac{1}{\gamma \ln n} \ln\left(\frac{cn}{\Theta(\ln n)}\right)} \to n^{\frac{1}{\gamma}}$. □

5.2. Binomial distribution

We now turn to the case where the subtree located after an arbitrary interpolation step stores elements distributed in its associated subinterval according to a Binomial Distribution. For the rest of this section, let $B(|\mathcal{U}|, p)$ be a Binomial distribution w.r.t. a universe of discrete elements $\mathcal{U}$. Assume that $p|\mathcal{U}| = O(n)$, with n being the size of our data structure, while p is *unknown* to us.

Consider the "dense" subset $I_1 = \{\max\{0, (1-\sigma)|\mathcal{U}|p\}, \ldots, (1+\sigma)|\mathcal{U}|p\} \subseteq \mathcal{U}$, "centered" at the mean value $|\mathcal{U}|p$ with "radius" $\sigma\, p|\mathcal{U}| = O(n)$, for $\sigma > 0$. According to Ineq. (2), I_1 is *not* (n^{α}, n^{δ})-smooth for any constants $0 < \alpha, \delta < 1$, since during $\Theta(n)$ operations (insert/delete/search) $\Omega(n)$ random elements w.h.p. are drawn from the subset I_1. Hence interpolation search is not suitable on I_1. Nevertheless, we are able to achieve a search time of $O(\log\log n)$ w.h.p. for Binomial distributions, as we show in the next theorem.

Theorem 6. *Consider a set of n (not necessarily distinct) elements produced by a sequence of $B(|\mathcal{U}|, p)$-random insertions and random deletions. Then, there exists a dynamic interpolation search data structure which achieves w.h.p. $O(\log\log n)$ search time. The space usage of the data structure is $\Theta(n)$ and the worst-case update time (position given) is $O(1)$.*

Proof of Theorem 6. For ease of exposition, let us first assume that p is known to us; later, we shall discuss how to eliminate this assumption.

Let $I_1 = \{\max\{0, (1-\sigma)|\mathcal{U}|p\}, \ldots, (1+\sigma)|\mathcal{U}|p\} \subseteq \mathcal{U}$ be a "dense" subset, "centered" at the mean value $|\mathcal{U}|p$ with "radius" $\sigma\, p|\mathcal{U}| = O(n)$, for $\sigma > 0$. Note that $|I_1| \leq 2\sigma\, p|\mathcal{U}| = O(n)$, while $I_2 = \mathcal{U} \setminus I_1$ can be arbitrarily large w.r.t. n, since the universe

[8] We use that $n! \sim \left(\frac{n}{e}\right)^n \sqrt{2\pi n}$ and ignore inverse polynomial multiplicative terms.

size $|\mathcal{U}|$ can be much larger than n. However, tail bounds [34, Theorems 4.1 & 4.2] establish that $I_2 = \mathcal{U} \setminus I_1$ behaves as a "sparse" bucket. For each insertion, a random element $X > (1+\sigma)|\mathcal{U}|p$ occurs with probability that is at most

$$\left[\frac{e^{\sigma}}{(1+\sigma)^{(1+\sigma)}}\right]^{|\mathcal{U}|p}$$

Similarly, a random element $X < (1-\sigma)|\mathcal{U}|p$ has probability at most

$$e^{-\frac{\sigma^2|\mathcal{U}|p}{2}}$$

Adding the above, during $rn = \Theta(n)$ update operations (insert/delete), the union bound gives that the probability to have at least one element in I_2 is

$$rn \cdot \left[\frac{e^{\sigma}}{(1+\sigma)^{(1+\sigma)}} + e^{-\frac{\sigma^2}{2}}\right]^{|\mathcal{U}|p} \to 0, \quad \forall \sigma > e_0 = 2.7182$$

since $\frac{e^{\sigma}}{(1+\sigma)^{(1+\sigma)}} + e^{-\frac{\sigma^2}{2}} < 1, \forall \sigma > e_0$.

Hence, if we *know* p we can safely construct a van Emde Boas tree [54,55] with universe the interval $I_1 = \{0, \ldots, (1+e_0)|\mathcal{U}|p\}$ and w.h.p. achieve $O(\log\log n)$ time per operation, while for the remaining universe $I_2 = \mathcal{U} \setminus I_1$ *no element* w.h.p. is drawn from it.

We now turn to the case where p is *unknown* and the only known fact is that the binomial distribution has expectation $|\mathcal{U}|p = O(n)$. The idea is to overestimate I_1 by an interval $I_1^* \supseteq I_1$, computed by the *initialization* phase (consisting of n random insertions), while w.h.p. not affecting considerably the efficiency of the supported operations.

This can be done by ordering increasingly the n random observations, $X_{(1)} \leq \ldots \leq X_{(n)}$, and note from the above calculations that w.h.p. $|\mathcal{U}|p < X_{(n)} < (1+e_0)|\mathcal{U}|p$. Hence, we can overestimate $(1+e_0)|\mathcal{U}|p$ by $(1+e_0)X_{(n)}$, which does not affect the efficiency of the supported operations, since it still holds that $O(|\mathcal{U}|p) = O(n)$ and safely set $I_1^* = \{0, \ldots, (1+e_0)X_{(n)}\}$ and $I_2^* = \mathcal{U} \setminus I_1^*$. □

6. Conclusions and discussion

We have presented a simple dynamic data structure that employs interpolation search to answer predecessor search queries for elements in a μ-random set of size n, where μ is a smooth density function. Our data structure uses $O(n)$ space and retrieves any element within $O(\log\log n)$ time w.h.p. It makes no use of REP arrays, since we have proved that such arrays *destroy* important statistical properties of the stored elements and thus the probabilistic analysis of known data structures [3,27,29,32] that use such arrays does not hold for the general case where duplicate elements may exist. As a by-product of our main structure we have also presented a data structure achieving $O(1)$ search time (or almost $O(1)$ for continuous distributions) w.h.p. This result holds for the class of $\left(\frac{n}{g}, \ln^{O(1)} n\right)$-smooth distributions for a constant $g \geq 1$, for which $O(1)$ expected search time was known, being as well the first *high probability* result, since all previously known results concern only the average search time. Finally we were able to show that a slight modification of our data structure achieves $O(\log\log n)$ time with high probability for power law and binomial distributions. No previous IS structure achieves such a time bound for these distributions.

Another data structure that is extensively studied on random inputs is the trie [8,19]. To the best of our knowledge (a more interested reader can find nice expositions and extended bibliography of this subject in [9,15,48]), all trie-related probabilistic approaches deal only with the static case, as also noted in [35]. In particular, the authors in [35] provide an experimental evaluation of the dynamic level compressed tries without providing a rigorous analysis (to the best of our knowledge no such analysis exists yet). Concerning search time, the trie depth is $\sim \log n$ for such general random input distributions as the ones we study here. Remarkably, the work in [4] achieves $O(\log^* n)$ search time for uniform real elements. Finally, no space bounds (w.r.t. total number of bits) have been studied, mainly due to the assumption of real input elements of infinite length, an assumption made to facilitate the trie probabilistic analysis.

Declaration of competing interest

The authors declare that they do not have any financial, general, and institutional competing interests.

Acknowledgments

We are indebted to Lefteris Kirousis for various helpful discussions and to the anonymous referees for their insightful comments that helped us to improve the presentation.

References

[1] G.M. Adel'son-Vel'skii, E.M. Landis, An algorithm for the organization and information, Dokl. Akad. Nauk SSSR 146 (1962) 263–266 (in Russian); English translation in Sov. Math. 3 (1962) 1259–1262.
[2] A.V. Aho, J.E. Hopcroft, J.D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, 1974.
[3] A. Andersson, C. Mattsson, Dynamic Interpolation Search in $o(\log\log n)$ time, in: Proc. 20th Colloquium on Automata, Languages and Programming, ICALP 1993, in: LNCS, vol. 700, 1993, pp. 15–27.
[4] A. Andersson, S. Nilsson, Improved behaviour of tries by adaptive branching, Inf. Process. Lett. 46 (6) (1993) 295–300.
[5] A. Anderson, M. Thorup, Tight(er) worst-case bounds on dynamic searching and priority queues, in: Proc. 32nd ACM Symposium on Theory of Computing, STOC 2001, 2000, pp. 335–342.
[6] A. Anderson, M. Thorup, Dynamic ordered sets with exponential search trees, J. ACM 54 (3) (2007) 1–40.
[7] P. Beame, F. Fich, Optimal bounds for the predecessor problem and related problems, J. Comput. Syst. Sci. 65 (1) (2002) 38–72.
[8] R. de la Briandais, File searching using variable length keys, in: Western Joint Computer Conference, AFIPS Press, 1959.
[9] N. Broutin, Shedding New Light on Random Trees, PhD thesis, McGill Univ., 2007.
[10] F.W. Burton, G.N. Lewis, A robust variation of Interpolation Search, Inf. Process. Lett. 10 (4–5) (1980) 198–201.
[11] S.A. Cook, Linear time simulation of deterministic two-way push-down automata, in: Proc. IFIP Congress, 1971, pp. 75–80.
[12] S.A. Cook, R.A. Reckhow, Time bounded random access machines, J. Comput. Syst. Sci. 7 (1973) 354–375.
[13] E. Demaine, T. Jones, M. Pătraşcu, Interpolation Search for non-independent data, in: Proc. 15th ACM-SIAM Symposium on Discrete Algorithms, SODA 2004, 2004, pp. 522–523.
[14] W. Feller, An Introduction to Probability Theory and Its Applications, vol. II, 2nd edition, Wiley, New York, 1971.
[15] P. Flajolet, The ubiquitous digital tree, in: Theoretical Aspects of Computer Science, STACS 2006, in: LNCS, vol. 3884, 2006, pp. 1–22.
[16] R. Fleischer, A simple balanced search tree with $O(1)$ worst case update time, Int. J. Found. Comput. Sci. 7 (1996) 137–149.
[17] K.E. Foster, A statistically based interpolation binary search, TR, Winthrop College, SC.
[18] G. Frederickson, Implicit data structures for the dictionary problem, J. ACM 30 (1) (1983) 80–94.
[19] E.H. Fredkin, Trie memory, Commun. ACM 3 (9) (1960) 490–499.
[20] M.L. Fredman, D.E. Willard, Surpassing the information theoretic bound with fusion trees, J. Comput. Syst. Sci. 47 (1993) 424–436.
[21] G. Gonnet, Interpolation and Interpolation-Hash Searching, PhD thesis, University of Waterloo, 1977.
[22] G. Gonnet, L. Rogers, J. George, An algorithmic and complexity analysis of interpolation search, Acta Inform. 13 (1980) 39–52.
[23] G. Graefe, B-tree indexes, Interpolation Search, and skew, in: Proc. 2nd ACM International Workshop on Data Management on New Hardware, DaMoN 2006, 2006, 5.
[24] T. Hagerup, Sorting and searching on the word RAM, in: Theoretical Aspects of Computer Science, STACS'98, in: LNCS, vol. 1373, 1998, pp. 366–398.
[25] S. Huddleston, K. Mehlhorn, A new data structure for representing sorted lists, Acta Inform. 17 (1982) 157–184.
[26] A. Itai, A. Konheim, M. Rodeh, A sparse table implementation of priority queues, in: Proc. 8th Colloquium on Automata, Languages and Programming, ICALP'81, in: LNCS, vol. 115, 1981, pp. 417–431.
[27] A.C. Kaporis, C. Makris, S. Sioutas, A. Tsakalidis, K. Tsichlas, C. Zaroliagis, Improved bounds for finger search on a RAM, in: European Symposium on Algorithms, ESA 2003, in: LNCS, vol. 2832, 2003, pp. 325–336.
[28] A. Kaporis, C. Makris, S. Sioutas, A. Tsakalidis, K. Tsichlas, C. Zaroliagis, Dynamic Interpolation Search revisited, in: Proc. 33rd Colloquium on Automata, Languages and Programming, ICALP 2006, in: LNCS, vol. 4051, 2006, pp. 382–394.
[29] A. Kaporis, C. Makris, S. Sioutas, A. Tsakalidis, K. Tsichlas, C. Zaroliagis, Improved bounds for finger search on a RAM, Algorithmica 66 (2) (2013) 249–286.
[30] C. Levcopoulos, M.H. Overmars, A balanced search tree with $O(1)$ worst case update time, Acta Inform. 26 (1988) 269–277.
[31] Y. Manolopoulos, Y. Theodoridis, V. Tsotras, Advanced Database Indexing, Kluwer Academic Publishers, 2000.
[32] K. Mehlhorn, A. Tsakalidis, Dynamic Interpolation Search, J. ACM 40 (3) (1993) 621–634.
[33] M. Mitzenmacher, A brief history of generative models for power law and lognormal distributions, Internet Math. 1 (2) (2004) 226–251.
[34] R. Motwani, P. Raghavan, Randomized Algorithms, Cambridge University Press, 1995.
[35] S. Nilsson, M. Tikkanen, An experimental study of compression methods for dynamic tries, Algorithmica 33 (1) (2002) 19–33.
[36] M. Overmars, J. Leeuwen, Worst case optimal insertion and deletion methods for decomposable searching problems, Inf. Process. Lett. 12 (4) (1981) 168–173.
[37] M. Pătraşcu, M. Thorup, Time-space trade-offs for predecessor search, in: Proc. 38th ACM Symposium on Theory of Computing, STOC 2006, 2006, pp. 232–240.
[38] Y. Perl, L. Gabriel, Arithmetic Interpolation Search for alphabet tables, IEEE Trans. Comput. 41 (4) (1992) 493–499.
[39] Y. Perl, A. Itai, H. Avni, Interpolation Search – a $\log\log N$ search, Commun. ACM 21 (7) (1978) 550–554.
[40] Y. Perl, E.M. Reingold, Understanding the complexity of the Interpolation Search, Inf. Process. Lett. 6 (6) (1977) 219–222.
[41] W.W. Peterson, Addressing for random storage, IBM J. Res. Dev. 1 (4) (1957) 130–146.
[42] F. Preparata, M. Shamos, Computational Geometry, Springer, 1985.
[43] N. Santorno, J.B. Sidney, Interpolation binary search, Inf. Process. Lett. 20 (1985) 179–181.
[44] A. Schönhage, On the power of random access machines, in: Proc. 6th Colloquium on Automata, Languages and Programming, ICALP'79, in: LNCS, vol. 71, 1979, pp. 520–529.
[45] R. Sedgewick, Open problems in the analysis of sorting and searching algorithms, in: Workshop on the Probabilistic Analysis of Algorithms, Princeton, May 1997.
[46] J.C. Shepherdson, H.E. Sturgis, Computability of recursive functions, J. ACM 10 (2) (1963) 217–255.
[47] K.J. Supowit, E.M. Reingold, Divide and conquer heuristics for minimum weighted Euclidean matching, SIAM J. Comput. 12 (1) (1983) 118–143.
[48] W. Szpankowski, Average Case Analysis of Algorithms on Sequences, Wiley, 2001.
[49] R.E. Tarjan, Efficiency of a good but not linear set-union algorithm, J. ACM 22 (2) (1975) 215–225.
[50] R.E. Tarjan, A class of algorithms which require nonlinear time to maintain disjoint sets, J. Comput. Syst. Sci. 18 (2) (1979) 110–127.
[51] R.E. Tarjan, Updating a balanced search tree in $O(1)$ rotations, Inf. Process. Lett. 16 (5) (1983) 253–257.
[52] R.E. Tarjan, Data Structures and Network Algorithms, SIAM, 1983.
[53] M. Thorup, On RAM priority queues, SIAM J. Comput. 30 (1) (2000) 86–109.
[54] P. van Emde Boas, Preserving order in a forest in less than logarithmic time and linear space, Inf. Process. Lett. 6 (1977) 80–82.
[55] P. van Emde Boas, R. Kaas, E. Zijlstra, Design and implementation of an efficient priority queue, Math. Syst. Theory 10 (1977) 99–127.
[56] D.E. Willard, Log-logarithmic worst-case range queries are possible in space $\Theta(N)$, Inf. Process. Lett. 17 (2) (1983) 81–84.
[57] D.E. Willard, Searching unindexed and nonuniformly generated files in $\log\log N$ time, SIAM J. Comput. 14 (4) (1985) 1013–1029.

Please cite this article in press as: A. Kaporis et al., Dynamic Interpolation Search revisited, Inf. Comput. (2019), https://doi.org/10.1016/j.ic.2019.104465

[58] D.E. Willard, Applications of the fusion tree method to computational geometry and searching, in: Proc. 3rd ACM-SIAM Symposium on Discrete Algorithms, SODA'92, 1992, pp. 286–295.
[59] D.E. Willard, Examining computational geometry, van Emde Boas trees, and hashing from the perspective of the fusion tree, SIAM J. Comput. 29 (3) (2000) 1030–1049.
[60] A.C. Yao, F.F. Yao, The complexity of searching an ordered random table, in: Proc. 17th IEEE Symposium on Foundations of Computer Science, FOCS'76, 1976, pp. 173–177.