View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by University of Liverpool Repository

Random input helps searching predecessors

Djamal Belazzougui
CAPA, DTISI, CERIST, Algeria
dbelazzougui@cerist.dz

Alexis C. Kaporis
ICSD
University of the Aegean and Hellenic Open University, Greece
kaporisa@aegean.gr

Paul G. Spirakis
University of Liverpool and University of Patras
P.Spirakis@liverpool.ac.uk

CEUR-WS.org/Vol-2113/paper10.pdf

Abstract

A data structure problem consists of the finite sets: D of data, Q of queries, A of query answers, associated with a function $f : D \times Q \to A$. The data structure of file X is "static" ("dynamic") if we "do not" ("do") require quick updates as X changes. An important goal is to compactly encode a file $X \in D$, such that for each query $y \in Q$, function $f(X, y)$ requires the minimum time to compute an answer in A. This goal is trivial if the size of D is large, since for each query $y \in Q$, it was shown that $f(X, y)$ requires $O(1)$ time for the most important queries in the literature. Hence, this goal becomes interesting to study as a trade off between the "storage space" and the "query time", both measured as functions of the file size $n = |X|$. The *ideal* solution would be to use *linear* $O(n) = O(|X|)$ space, while retaining a *constant* $O(1)$ query time. However, if $f(X, y)$ computes the *static predecessor* search (find largest $x \in X : x \leq y$), then Ajtai [Ajt88] proved a negative result. By using just $n^{O(1)} = |X|^{O(1)}$ data space, then it is *not* possible to evaluate $f(X, y)$ in $O(1)$ time $\forall y \in Q$. The proof exhibited a bad distribution of data D, such that $\exists y^* \in Q$ (a "difficult" query y^*), that $f(X, y^*)$ requires $\omega(1)$ time. Essentially [Ajt88] is an existential result, resolving the *worst case* scenario. But, [Ajt88] left open the question: do we *typically*, that is, *with high probability* (w.h.p.) [1] encounter such "difficult" queries $y \in Q$, when assuming reasonable distributions *with respect to* (w.r.t.) queries and data? Below we make reasonable assumptions w.r.t. the distribution of the queries $y \in Q$, as well as w.r.t. the distribution of data $X \in D$. In two interesting scenarios studied in the literature, we resolve the *typical* (w.h.p.) query time.

Copyright © by the paper's authors. Copying permitted for private and academic purposes.

In: L. Ferrari, M. Vamvakari (eds.): Proceedings of the GASCom 2018 Workshop, Athens, Greece, 18–20 June 2018, published at http://ceur-ws.org

[1] In the paper given a random file of size $\Theta(n)$ an event occurs w.h.p. if its probability $\to 1$ as $n \to \infty$

First scenario: we assume *arbitrary* distribution of data $X \in D$, but, we restrict ourselves to *uniform* queries $y \in Q$. We show that Ajtai's result is not typical: by just using $O(n)$ space, w.h.p. $f(X,y)$ can be computed in $O(1)$ time. Then we extend the *w.h.p.* $O(1)$ time to *average* $O(1)$ time.
Second scenario: we assume *arbitrary* queries $y \in Q$, but we restrict to "real world" (f_1, f_2)-*smooth* [AM93, MT93, Wil85] distributions of data $X \in D$. **(i)** We show that: w.h.p. $f(X,y)$ can be computed in $O(1)$ time just using $O(n^{1+\delta})$ space for *any* constant $\delta > 0$. Also we extend the *w.h.p.* $O(1)$ time to *average* $O(1)$ time. **(ii)** We limit further the data space to $n^{1+\frac{1}{\log\log n}} = n^{1+o(1)}$, and, show that w.h.p. $f(X,y)$ can be computed in the slightly higher time $O(\log\log\log n)$. **(iii)** Finally, we limit even more the data space to $O(n)$, and, show that w.h.p. $f(X,y)$ can be computed in $O(\log\log n)$ time.

1 Introduction

The problem. Let X be a dynamic file of n keys, each stored in a memory cell with $\ell \leq b$ bits, $b = \log|U|$ is the word length of any key in the universe $U = \{1, \ldots, 2^b\}$. The ℓ bits must uniquely represent the n keys in file X, therefore $\ell \geq \log n$. The data size of X, the total of bits stored in the n memory cells, in our work must be $O(n^{1+\delta}), \forall\delta > 0$, therefore, as we show, ℓ must be $O(\log n)$. To prove time upper bounds in the RAM model of computation, we use $O(\log n)$ bits per cell, which makes realistic the assumption that cell content based operations take $O(1)$ time in the C language. Thus, in this context of the *cell probe* [Yao81] model that we use, it is *not* realistic if cells contain *real* numbers, or keys with arbitrarily large digit precision. We consider queries as: *Membership* `Memb`(y, X) "determine if $y \in X$, for any $y \in U$", *Insertion* `Ins`(y, X) "insert y into X", *Deletion* `Del`(y, X) "delete $y \in X$" and *Predecessor* `Pred`(y, X) "determine the largest $x \in X$ such that $x \leq y$, for an arbitrary $y \in U$". An intrinsic trade off arises between *size* and *time*, as the two examples suggest. If file X is static and we wish the minimum time but compromise space, then $\forall y \in U$ we can tabulate using $2^b = |U|$ cells the corresponding predecessor in X. Inspecting the appropriate cell we get the predecessor in $O(1)$ time. But, the size of cells equals the universe size $|U| = 2^b$, which may be arbitrarily larger than file size $|X|$. On the other example, if we wish the minimum space but compromise time, we can store in $O(n)$ cells the *ordered* keys in X and via binary search in $O(\log n)$ time, we get the predecessor $\forall y \in U$. Paragraph (i) below presents related work with no input distributional assumptions, it describes how prohibitive time lower bounds arise as space gets closer to linear. Then paragraph (ii) presents related work with input distributional assumptions, it shows that such time lower bounds can be surpassed as space becomes almost linear via careful probabilistic analyses.
Our goal. We show in our main results Theorem 2.1 (Section 2) and Theorem 3.1 (Section 3), that under reasonable assumptions about distributions w.r.t. queries and data, queries w.h.p. take close to $O(1)$ time, while space is at most $O(n^{1+\delta}), \forall\delta > 0$ (almost linear). To the best of our knowledge, these are the best space bounds w.r.t. the total bits stored such that $O(1)$ time can be achieved, as paragraph **(ii)** in the related work suggests.
Space vs time complexity results. **(i)** Let us start with the related work *without assumptions about query and data distributions*. Membership requires $O(1)$ time and $O(n)$ space [FKS84, Pag00]. Willard [Wil83] showed $O(n)$ space and $O(\log b) = O(\log\log|U|)$ time upper bound for predecessor search. Ajtai's [Ajt88] original time lower bound, revisited by Miltersen [Mil94], shows that there exists a file size n, as a function of the memory cell bits ℓ, such that predecessor search takes $\Omega(\sqrt{\ell})$ time, when space is limited to poly(n), or equivalently, there exists an ℓ function of n such that time becomes $\Omega(\sqrt[3]{n})$. Later [FW93] improved the time upper bound in [Wil83] to $O\left(\min\left\{\frac{\log n}{\log \ell}, \log \ell\right\}\right) = O(\sqrt{\log n})$ using $O(n)$ space, where here and in the related work of this paragraph $\ell = b = \log|U|$. The followup [MNSW98] generalized the lower bound in [Ajt88] to randomized algorithms. Beam and Fich [BF02] improved the corresponding lower bounds in [Ajt88], up to $\Omega\left(\frac{\log \ell}{\log\log \ell}\right)$ and $\Omega\left(\sqrt{\frac{\log n}{\log\log n}}\right)$ respectively. Similar lower bounds appeared in [Xia92]. Also [BF02] improved the time upper bounds to $O\left(\min\left\{\frac{\log n}{\log \ell}, \frac{\log \ell}{\log\log \ell}\right\}\right) = O(\sqrt{\frac{\log n}{\log\log \ell}})$ but using $\Omega(n^2)$ space, while posing the important question if

their time bound can be achieved via linear space. The work of Sen and Venkatesh [SV08] yields the lower bounds $\Omega\left(\frac{\log n}{\log b}, \frac{\log \ell}{\log\log S}\right)$, with S the total file size and $b \geq \ell$. Finally, the work [PT06a] proved detailed optimal space vs time trade-offs, the followup [PT07] generalized to randomized algorithms. In particular, [PT06a] answered the important question in [BF02], showing that, as close to linear space as: $n^{1+1/\exp(\lg^{1-\varepsilon}\ell)}, \forall\varepsilon > 0$, it is achievable the $O\left(\frac{\log \ell}{\log\log \ell}\right)$ time, here $\lg x = \lceil \log_2(x+2) \rceil$. However, [PT06a] also proved that if space drops to $n \log^{O(1)} n$ the *van Emde Boas* (vEB) $O(\lg \ell)$ time can not be improved.

(ii) We proceed with related work that *assumes query and data distributions*. Peterson's experimental *Interpolation Search* (`IS`) [Pet57] for predecessors, assumed uniform n real keys stored in a static file X. The idea of `IS` is to use the input distribution towards estimating the expected location in the static file X of the predecessor of each query $y \in U$. Yao and Yao [YY76] proved `IS` takes $\Theta(\log\log n)$ average time, for static file X with n *real* keys uniformly distributed. Interestingly, their result and the subsequent ones presented below, surpass the time bounds mentioned in the above paragraph. However the space is *not* bounded up to $O(n)$ w.r.t. the total bits stored in the cells. Since this and subsequent results simplify their probabilistic analysis [KMSTTZ06, Sect. 2] by assuming *real* keys of arbitrary precision. In [GRG80, PR77, Gon77] several aspects of the `IS` are described and analyzed on (almost) uniform and static file X. Willard [Wil85] proved the same time assuming *regular* input distributions of real keys, significantly extending over the uniform input assumed before. Recently, [DJP04] considered non-random input data of a static file X that possess enough "pseudo-randomness" for effective `IS` to be applied. The study of a *dynamic* file X, with uniform insertions/deletions was initiated in [Fre83, IKR81]. In [Fre83] the dynamic data structure supports insertions/deletions of real keys in $O(n^\epsilon), \epsilon > 0$, time and $O(\log\log n)$ expected time for `IS`. The dynamic structure of [IKR81] has expected insertion $O(\log n)$ time for real keys, amortized insertion time $O(\log^2 n)$ and claimed that supports `IS` . Mehlhorn and Tsakalidis [MT93] analyzed a dynamic version of `IS`, the *Interpolation Search Tree (IST)* with $O(\log\log n)$ expected search/update time for the (f_1, f_2)*-smooth* continuous distributions of real keys, which are significantly larger than the regular ones in [Wil85]. Andersson and Mattson [AM93] further generalized and refined the notion of (f_1, f_2)*-smooth* continuous distributions w.r.t. real keys of [MT93], presented the *Augmented Sampled Forest*, extending the class of continuous input distributions with $\Theta(\log\log n)$ expected search time. Notably, they surpassed the lower bounds of the previous paragraph, achieving w.h.p. $O(1)$ time for *bounded* densities of real keys. Their use of real keys made elusive to bound the space w.r.t. the total bits stored. In [KMSTTZ03], with real keys as in [AM93], a *finger search* version of (`IS`) was studied with $O(1)$ worst-case update time (update position given) and w.h.p. $O(\log\log d)$ expected search time. The novelty here is that search initiates from a *leaf node* (pointed by a finger) of the search tree, d is the distance from the fingered leaf up to the searched key. Then [KMSTTZ06] initiated the probabilistic analysis for *discrete* (f_1, f_2)-smooth distributions with keys of limited bits. It alleviated the obstacles of previous work about retaining randomness per recursive step. However in [KMSTTZ06] no space upper bounds w.r.t. the total of bits where proven. Recently, [BHM13] reviews how distributional information improves time and space. Also [BFHM16] bounds the time w.r.t. the entropy of the input distribution, while bounds space w.r.t. the universe size. Finally a preliminary work [Cun15] assumes smooth input distribution w.r.t. queries and claims to achieve $O(\log\log n)$ time in $O(n)$ space.

Smooth data distributions. [MT93] introduced the class of (f_1, f_2)-smooth real distributions, subsequently generalized by [AM93]. For appropriate f_1, f_2 parameters the class contains *regular* [Wil85], as well as *bounded* (an extension of uniform) distributions [GRG80, PR77, Gon77]. Recently [KMSTTZ06] introduced the analogous Definition 1.1 for *discrete* distributions.

Definition 1.1 ([KMSTTZ06]). An unknown *discrete* probability distribution $\mathbb{P}$ is (f_1, f_2)-smooth w.r.t. the interval $[a,\ b]$ that corresponds to the discrete universe $U = \{a, \ldots, b\}$ of the input keys, if $\exists\beta > 0 : \forall a \leq c_1 < c_2 < c_3 \leq b$ and $n \in \mathbb{N}$, for a $\mathbb{P}$-random key y to hold:

$$P\left[c_2 - \frac{c_3 - c_1}{f_1(n)} \leq y \leq c_2 | c_1 \leq y \leq c_3\right] = \sum_{x_i = c_2 - \frac{c_3 - c_1}{f_1(n)}}^{c_2} \mathbb{P}[c_1,\ c_3](x_i) \leq \beta \frac{f_2(n)}{n}$$

$\mathbb{P}[c_1, c_3](x_i) = 0, \forall x_i < c_1, x_i > c_3$ & $\mathbb{P}[c_1,\ c_3](x_i) = \mathbb{P}(x_i)/\sum_{x_j \in [c_1,\ c_3]} \mathbb{P}(x_i)$, $\forall x_i \in [c_1,\ c_3]$.

In Definition 1.1, function f_1 partitions an arbitrary subinterval $[c_1, c_3] \subseteq [a, b]$ into f_1 equal parts, each of length $\frac{c_3 - c_1}{f_1} = O(\frac{1}{f_1})$. That is, f_1 measures how *fine* is the partitioning. 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

$[c_2 - \frac{c_3-c_1}{f_1}, c_2] \subseteq [c_1, c_3]$. Actually, *any* probability distribution is $(f_1, \Theta(n))$-smooth, for a suitable choice of β. The most general class of unknown (f_1, f_2)-smooth input distributions (Definition 1.1), so that $O(\log\log n)$ expected search times is achievable by IS, was defined in [AM93] with f_1, f_2 parameters as: $f_1(n) = \frac{n}{\log^{1+\epsilon}\log n}$, $f_2(n) = n^\alpha, \alpha < 1$. In our Theorem 3.1 (Section 3) we consider more general parameters: $f_1(n) = n^\gamma$ and $f_2(n) = n^\alpha$ with constants $0 < \alpha < 1, \gamma > 0$.

2 Uniform *queries*, arbitrary *data* distribution: $O(1)$ time & $O(n)$ space

Theorem 2.1. *In a static set X stored by an adversary, regardless of the distribution of the data D, (almost) uniform queries take $O(1)$ time w.h.p..*

Proof. It suffices to assume that the queries are uniformly distributed, while an adversary selects from the universe U and stores in file X the n keys in the *worst possible* way. Our approach is simple: divide the universe U into $n\log^c n$ equally sized buckets (c is constant) and store a bitmap in which we set to 1 only non-empty buckets. This bitmap has size m with redundancy $m/\log^c m$ with $O(1)$ rank queries [Pat08]. It will contain $\leq n$ ones (as only $\leq n = |X|$ buckets can be non empty) and $n\log^c n - n \leq$ zeros, thus it can be coded in $O(n)$ space while supporting predecessor queries in $O(1)$ time. In this bitmap, if a query key lands in an empty bucket i we immediately get its predecessor which is the largest key in bucket $j < i$, the closest non empty bucket before empty bucket i. If a query is chosen uniformly at random, then it lands w.h.p. $\geq 1 - \frac{n}{n\log^c n} = 1 - \frac{1}{\log^c n} = 1 - o(1)$ in an empty bucket and the query is answered in $O(1)$ time. Only when a query lands in a non empty bucket (which happens with the remaining probability $\leq 1/\log^c n = o(1)$) we do need to do more complicated things (up to $O\left(\sqrt{\frac{\log n}{\log\log n}}\right) = o(\log^c n)$ time, see Section 4.3 for this worst-case guarantee) to answer the predecessor query in $\omega(1)$ time. In turn, this yields $O(1)$ average time. □

3 Arbitrary *queries*, smooth *data* distribution: $O(1)$ time & space $O(n^{1+\delta}), \forall\delta > 0$

In this section we prove our main Theorem 3.1 using the data structure DS illustrated in Figure 1.

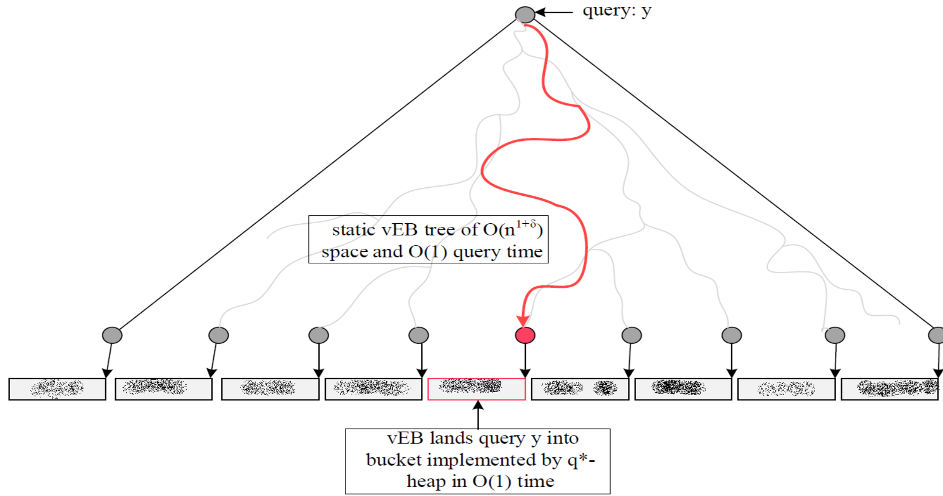

Figure 1: The upper static vEB tree uses at most $O(n^{1+\delta}), \forall\delta > 0$, space: each node has $O(\log n)$ bits. This tree in $O(1)$ time lands each query $y \in Q$, handed at the root, into the appropriate leaf node associated to a bucket. Each bucket is a dynamic q^*-*heap*. Thus, this landed q^*-*heap* takes $O(1)$ query time: w.h.p. has $\Theta(\log n)$ load.

Theorem 3.1. *Consider arbitrary queries $y \in Q$. Data $X \in D$ are drawn from unknown f_1, f_2-smooth distribution (Definition 1.1), with $f_1(n) = n^\gamma$, $f_2(n) = n^\alpha$ with constants $0 < \alpha < 1, \gamma > 0$. For any positive constant $\delta > 0$ the following hold:* **a.** *it requires $O(n^{1+\delta})$ space and time to build and supports $\Omega(n^{1+\delta})$ queries, where* Insrt(y, X) *and* Del(y, X) *can occur in arbitrary order, subject to the load constraint that $\Theta(n) = C_{\max}n$ keys exist in the* DS *over all query steps.* **b.** DS *supports $O(1)$ query time w.h.p..* **c.** DS *supports $O(\log\log\log n)$ query time w.h.p., using only $n^{1+\frac{1}{\log\log n}} = n^{1+o(1)}$ space (construction time).* **d.** DS *supports $O(\log\log n)$ query*

time w.h.p., using $O(n)$ space (built time). **e.** DS *supports $O(1)$ w.h.p. amortized query time.* **f.** DS *supports $O(\sqrt{\frac{\log n}{\log\log n}})$ worst-case query time. The above w.h.p. $O(1)$ query times extend to $O(1)$ average times.*

During the *preprocessing phase* that takes $O(n^{1+\delta})$ time, in Section 3.1 we build the *upper static* vEB tree shown in Figure 1 and prove time & space bounds in Corollary 3.4. During the *operational phase* that consists of $\Theta(n^{1+\delta})$ queries, the role of this upper static vEB tree is to place each query $y \in Q$ into the appropriate lower dynamic bucket (pointed by a leaf), that in turn answers the query $y \in Q$ by implementing a q^*-*heap*. More precisely, in Section 3.2 we construct the *lower dynamic buckets* illustrated in Figure 1, which are implemented as q^*-*heaps* with time & space bounds proved in Corollary 3.6. Hence, Theorem 3.1 directly follows by combining Corollary 3.4 and Corollary 3.6. For the worst-case time, we use two dynamic predecessor search data structures [BF02] with worst case $O(\sqrt{\frac{\log n}{\log\log n}})$ query and update times and linear space, see Section 4.3.

3.1 The upper static vEB tree in Figure 1

Construction of the upper static vEB tree in Figure 1. As it is folklore in the literature of dynamic IS, during the *preprocessing phase*, we draw n keys from $U = \{0, \ldots, 2^b\}$, according to an unknown f_1, f_2-*smooth* distribution (Definition 1.1), with parameters as in Theorem 3.1: $f_1(n) = n^\gamma$, $f_2(n) = n^\alpha$, where $0 < \alpha < 1, \gamma > 0$ are constants. We order increasingly the n keys:

$$\mathcal{X} = \{x_1 \leq \ldots \leq x_n\} \tag{1}$$

We select $\rho - 1 < n$ representative keys (order statistics):

$$\mathcal{R} = \{r_0, \ldots, r_\rho\} \tag{2}$$

from (1), setting $r_0 = 0, r_\rho = 2^b$, for each $0 < i < \rho$ the (bucket) representative key $r_i \in \mathcal{R}$ in (2) is the key $x_{i\alpha(n)n} \in \mathcal{X}$ in (1). These consecutive keys in $\mathcal{R}$ in (2) are the endpoints of $\rho - 1$ consecutive buckets that constitute the lower dynamic part of the data structure (Figure 1). The value $\alpha(n)n = \Theta(\log n)$ is tuned in Section 4.2 (the proof of Property 3.2) and Section 4.1 (the proof of Property 3.5). Intuitively, these $\rho - 1$ keys in $\mathcal{R}$ are sufficiently "$\alpha(n)n$-apart" such that in each part of partition $\mathcal{P}$ (Property 3.2 below) to appear w.h.p. *at most one* representative from $\mathcal{R}$. This follows from Property 3.2 (proved in Section 4.2), since each part in $\mathcal{P}$ w.h.p. contains $< \alpha(n)n$ keys from $\mathcal{X}$.

Property 3.2. *If we partition by $\mathcal{P}$ the interval $[0, 2^b]$, with endpoints dictated by the universe $U = \{0, \ldots, 2^b\}$ of the discrete input keys, into $|\mathcal{P}| = n^{C_1}, C_1 = O(1)$,* equal sized *parts, then w.h.p. each part contains* at most one *representative from $\mathcal{R}$ in (2).*

Thus we *uniquely* index with partition $\mathcal{P}$ in Property 3.2 each representative key $r_i \in \mathcal{R}$ in (2) using $\log|\mathcal{P}| = C_1 \log n = O(\log n)$ bits. Let $\widetilde{\mathcal{R}}$ this indexing of $\mathcal{R}$ in (2) obtained by partition $\mathcal{P}$. We store $\widetilde{\mathcal{R}}$ as the nodes of the upper static vEB data structure (Figure 1), as described in [PT06b, Lem. 17].
During the *operational phase* that consists of $\Omega(n^{1+\delta}), 0 < \delta < 1$ queries, we use the upper static vEB (Figure 1) and for each query w.r.t. key $y \in Q$ we determine in $O(1)$ time the i_y-th lower dynamic bucket (Figure 1) with endpoints in $\widetilde{\mathcal{R}}$ that satisfy: $\widetilde{r}_{i_y} \leq y < \widetilde{r}_{i_y+1}$. This is done by running query $\texttt{Pred}(y, \widetilde{\mathcal{R}})$ in the upper vEB tree that returns $\widetilde{r}_{i_y}$ in $O(1)$ time.
Space & time bounds for the upper static vEB tree in Figure 1. These are derived by Theorem 3.3 and Corollary 3.4 below.

Theorem 3.3. [PT06b, Lem. 17] *Let κ be any positive integer. Given any set $S \subseteq [0, m-1]$ of n keys, we can build in $O(n2^\kappa \log m)$ time a static data structure which occupies $O(n2^\kappa \log m)$ bits of space and solves the static predecessor search problem in $O(\log(\frac{\log m - \log n}{\kappa}))$ time per query.*

In our case and terminology, we map the characteristic function of the set $\widetilde{\mathcal{R}}$ to a bitstring S with length $|S| = m$ that equals the number of possible values of the keys stored in $\widetilde{\mathcal{R}}$. To figure out how large is, recall that each key $\widetilde{r}_i \in \widetilde{\mathcal{R}}$ is uniquely indexed by $\mathcal{P}$ (Property 3.2) with $C_1 \log n$ bits. Hence the number m of possible values of the keys in $\widetilde{\mathcal{R}}$ are $|S| = m = 2^{C_1 \log n}$. Thus by setting $\kappa = \delta \log n$ we can build the predecessor data structure on the $\widetilde{\mathcal{R}}$, it takes $O(n^{1+\delta})$ space and answers to queries in time $O(\log(\frac{\log m - \log n}{\kappa})) = O(\frac{C_1 - 1}{\delta}) = O(1)$.

Corollary 3.4. *Setting $\kappa = \delta \log n$, the predecessor data structure built on the $\widetilde{\mathcal{R}}$ takes $O(n^{1+\delta})$ space and answers to queries in time $O(\log(\frac{\log m - \log n}{\kappa})) = O(\log(\frac{C_1 - 1}{\delta})) = O(1)$. By setting $\kappa = \frac{\log n}{\log\log n}$, the data structure built on $\widetilde{\mathcal{R}}$ takes $O(n^{1+1/\log\log n}) = O(n^{1+o(1)})$ space and answers to queries in time $O(\log(\frac{\log m - \log n}{\kappa})) = O(\log(\frac{(C_1-1)\log n}{\frac{\log n}{\log\log n}})) = O(\log\log\log n)$. Finally by setting $\kappa = 1$, the data structure built on $\widetilde{\mathcal{R}}$ takes $O(n)$ space and answers to queries in time $O(\log(\frac{\log m - \log n}{\kappa})) = O(\log((C_1 - 1)\log n)) = O(\log\log n)$.*

3.2 The lower dynamic structure of q^*-heaps in Figure 1

In the lower dynamic structure in Figure 1, the i-th bucket is implemented as a q^*-heap and has endpoints $\widetilde{r}_i, \widetilde{r}_{i+1} \in \widetilde{\mathcal{R}}$, which are indexed with $C_1 \log n$ bits using $\mathcal{P}$ (Property 3.2) that truncates the representatives in $r_i, r_{i+1} \in \mathcal{R}$ in (2). Each query $y \in Q$ is landed by the upper static vEB tree into the i_y-th lower dynamic bucket with endpoints satisfying: $\widetilde{r}_{i_y} \leq y < \widetilde{r}_{i_y+1}$. The i_y-th bucket is located by running query $\texttt{Pred}(y, \widetilde{\mathcal{R}})$ in the upper vEB tree that returns $\widetilde{r}_{i_y}$ in $O(1)$ time. Given that Property 3.5 (proved in Section 4.1) holds, we get in Corollary 3.6 the space & time bounds.

Property 3.5. *During the* operational phase *that consists of $\Omega(n^{1+\delta}), 0 < \delta < 1$ queries, each such lower dynamic bucket (Figure 1) w.h.p. has load: $C_3 \log n \leq M \leq C_5 \log n$, with $C_3, C_5 = O(1)$.*

Corollary 3.6 (Corollary 3.2, [Wil92])**.** *Assume that in a database of n elements, we have available the use of pre-computed tables of size $o(n)$. Then for sets of arbitrary cardinality $M \leq n$, it is possible to have available variants of q^*-heaps using $O(M)$ space that have a worst-case time of $O(1 + \frac{\log M}{\log\log n})$ for doing member, predecessor, and rank searches, and that support an amortized time $O(1 + \frac{\log M}{\log\log n})$ for insertions and deletions.*

To apply Corollary 3.6 in our case, the database is the dynamic file X with n elements, so for our purposes it suffices to use pre-computed table of size $o(n)$. Also, the sets of arbitrary cardinality M in our case are the ρ lower dynamic buckets (Figure 1), each with endpoints: $[\widetilde{r}_i, \widetilde{r}_{i+1})$, implemented as q^*-heaps. By Property 3.5 (proved in Section 4.1), each such bucket has w.h.p. load within: $C_3 \log n \leq M \leq C_5 \log n$ with $C_3, C_5 = O(1)$. It follows that w.h.p. each of the q^*-heap time is $O(1 + \frac{\log M}{\log\log n}) = O(1 + \frac{\log(C_5 \log n)}{\log\log n}) = O(1)$ and the space per q^*-heap is $O(M) = O(C_5 \log n) = O(\log n)$. Also, the exponentially small failure probability when multiplied to the worst-case $O(\sqrt{\frac{\log n}{\log\log n}})$ time proved in Section 4.3 yields $O(1)$ average query time.

4 Technical properties

4.1 Proof of Property 3.5

In the lower dynamic structure in Figure 1, the i-th bucket is implemented as a q^*-heap and has endpoints $\widetilde{r}_i, \widetilde{r}_{i+1} \in \widetilde{\mathcal{R}}$, which are indexed with $C_1 \log n$ bits using $\mathcal{P}$ (Property 3.2) that truncates the representatives in $r_i, r_{i+1} \in \mathcal{R}$ in (2). Let $q_i(n)$ be the *i-bucket probability mass* of the unknown smooth (Definition 1.1) distribution $\mathbb{P}$ over the part with endpoints these representatives in $r_i, r_{i+1} \in \mathcal{R}$ in (2). That is, $q_i(n) = \sum_{y \in U : r_i \leq y < r_{i+1}} \mathbb{P}(y)$. Furthermore, by the construction of the ordering in (2), these r_i, r_{i+1} are $\alpha(n)n = \Theta(\log n)$-apart, than is, during the initialization of the data structure, exactly $\alpha(n)n = \Theta(\log n)$ random keys appear between these r_i, r_{i+1}. Now, assume the *i-bucket bad scenario* that this probability mass $q_i(n)$ is either $q_i(n) = \omega(\alpha(n))$ or $q_i(n) = o(\alpha(n))$. But, the probability of this *i-bucket bad scenario* is dominated by the Binomial distribution:

$$\binom{n}{\alpha(n)n} q_i(n)^{\alpha(n)n} (1 - q_i(n))^{(1-\alpha(n))n} \sim \left[\left(\frac{q_i(n)}{\alpha(n)} \right)^{\alpha(n)} \left(\frac{1 - q_i(n)}{1 - \alpha(n)} \right)^{1-\alpha(n)} \right]^n \to 0, n \to \infty \tag{3}$$

which vanishes exponentially in n if $q_i(n) = \omega(\alpha(n))$ or if $q_i(n) = o(\alpha(n))$. Note that there are $\rho < n$ such bad scenaria (the possible buckets) hence the union bound gives:

$$n \times \left[\max_{i \in [\rho]} \left\{ \left(\frac{q_i(n)}{\alpha(n)} \right)^{\alpha(n)} \left(\frac{1 - q_i(n)}{1 - \alpha(n)} \right)^{1-\alpha(n)} \right\} \right]^n \to 0, \tag{4}$$

which also vanishes if $q_i(n) = \omega(\alpha(n))$ or if $q_i(n) = o(\alpha(n))$. Thus, w.h.p. each bucket has probability mass $\Theta(\alpha(n)) = \Theta(\frac{\log n}{n})$. From exponentially strong tail bounds, w.h.p. no bucket contains less than $C_3 \log n$ keys and more than $C_5 \log n$ per query step, for appropriate constants $C_3, C_5 = \Theta(1)$.

4.2 Proof of Property 3.2

We partition the universe into $n^{C_1}, C_1 = O(1)$, equal parts, such each part in $\mathcal{P}$ gets expected load $\leq \log n < C_5 \log n$ during each query step $t = 1, \ldots, \Omega(n^{1+\delta})$ of the operational phase. The result follows by exponential tail bounds w.r.t. the load of the parts in $\mathcal{P}$, since by Property 3.5, each pair $r_i, r_{i+1} \in \mathcal{R}$ in (2) is at least $C_5 \log n$ apart, with constant C_5 appropriately high.
We construct $\mathcal{P}$ recursively, noting that the endpoints of each $\mathcal{P}$ part is a deterministic function of the parameters f_1, f_2 as assumed in our main Theorem 3.1. Also in Theorem 3.1 we assume that that per query step $t = 1, \ldots, \Omega(n^{1+\delta})$ the total number of stored keys are $< C_{\max} n = \nu$ with $C_{\max}$ a constant. In the 1st recursive step, $\mathcal{P}$ partitions the interval $[0, 2^b]$, with endpoints dictated by the universe $U = \{0, \ldots, 2^b\}$ of the discrete input keys, into $f_1(\nu) \geq f_1(n_t)$ equally sized parts. Smoothness (Definition 1.1) imply that each $\mathcal{P}$ part gets $\leq \beta \frac{f_2(n_t)}{n_t} \leq \beta \frac{f_2(\nu)}{n_t}$ probability mass per update step t (f_2 is increasing w.r.t. n). Here β is a constant (Definition 1.1) depending only on the characteristics of unknown distribution. Hence, during each query step t, each $\mathcal{P}$ part gets $\leq \beta \frac{f_2(\nu)}{n_t} \times n_t = \beta \nu^\alpha$ keys in expectation of the $n_t \leq \nu$ keys currently stored in X. Moreover, the number of input keys distributed on any such $\mathcal{P}$ part has exponentially small probability of deviating from its expectation $\leq \beta \nu^\alpha$. We will not take into account constant β in the subsequent deterministic partitioning without loss of generality. This partitioning is applied recursively within each such $\mathcal{P}$ part, until we reach a sufficiently small $\mathcal{P}$ part with expected number of keys $\leq \log n$ per query step $t = 1, \ldots, \Omega(n^{1+\delta})$. Let h be the number of such recursive partitions, then, it suffices:

$$\nu^{\alpha^h} \leq \log n \rightarrow h \leq \frac{\ln\left(\frac{\ln \ln(n)}{\ln(\nu)}\right)}{\ln \alpha} < \frac{\ln\left(\frac{\ln \ln(n)}{\ln(n)}\right)}{\ln \alpha} = -\log_\alpha(\ln(n)) \quad (5)$$

where the strict inequality follows from $n < \nu$. We upper bound the number of $\mathcal{P}$ parts obtained by the h recursions in (5). In the 1-st partition of the universe U, the number of $\mathcal{P}$ parts is $f_1(\nu) = f_1(\nu^{\alpha^0}) = (\nu^{\alpha^0})^\gamma = \nu^{\alpha^0 \cdot \gamma}$. In the 2-nd recursive partition, each $\mathcal{P}$ part will be further partitioned into $f_1(\nu^\alpha) = f_1(\nu^{\alpha^1}) = (\nu^{\alpha^1})^\gamma = \nu^{\alpha^1 \cdot \gamma}$ subparts. In general, in the $(i+1)$-th recursive partition an arbitrary $\mathcal{P}$ part will be divided into $f_1(\nu^{\alpha^i}) = (\nu^{\alpha^i})^\gamma = \nu^{\alpha^i \cdot \gamma}$ subparts. Taking into account Eq. (5), in the final level of recursive partition, the total number $|\mathcal{P}|$ of subparts is

$$\prod_{i=0}^{h} f_1\left(\nu^{\alpha^i}\right) = \prod_{i=0}^{h} (\nu^{\alpha^i})^\gamma < \prod_{i=0}^{h} \nu^{\alpha^i \cdot \gamma} = \nu^{\sum_{i=0}^{h} \alpha^i \cdot \gamma} = \nu^{\gamma \frac{\alpha^{h+1}-1}{\alpha-1}} = e^{\gamma \frac{\alpha}{\alpha-1}} \nu^{\frac{\gamma}{1-\alpha}} < \nu^{\frac{\gamma}{1-\alpha}} < C_{\max}^{\frac{\gamma}{1-\alpha}} n^{\frac{\gamma}{1-\alpha}}$$

with $\nu > n$ and α the smooth parameter in Theorem 3.1. Hence, the total number of bits needed for each such final part of $\mathcal{P}$ is at most $\log(|\mathcal{P}|) = \frac{\gamma}{1-\alpha} \log n + \log\left(C_{\max}^{\frac{\gamma}{1-\alpha}}\right) \leq C_1 \log n$.

4.3 Proof of $O(\sqrt{\frac{\log n}{\log \log n}})$ worst-case time

1. In the first predecessor data structure which we note by B_1 we initially insert all the numbers in interval $[1, \rho]$. Then B_1 will maintain the set of non empty buckets during the operational phase. The role of B_1 is to ensure the correctness of predecessor queries when some buckets get empty.
2. The second predecessor data structure which we note by B_2 is initially empty. The role of B_2 is to store all the overflowing elements which could not be stored in the q^*-heaps corresponding to their buckets.
Handling overflows. In addition, we maintain an array of ρ counters where each counter c_i associated with bucket i stores how many keys are stored inside that bucket and assume that the capacity of the q^*-heap associated with each bucket is $C = \Theta(\log n)$. Now at initialisation, all the buckets have initial load $\Theta(\log n)$ and all the keys of any bucket i are stored in the corresponding q^*-heap. Then at operational phase the insertions/deletions of keys belonging to a given bucket are directed to the corresponding q^*-heaps unless it is overflown. More precisely when trying to insert a key x into a bucket number i and we observe that $c_i \geq C$, we instead insert the key x in B_2. Symmetrically, when deleting a key x from a bucket i when $c_i \geq C$ proceeds as follows : If the key x is found into the q^*-heap, we delete it from there and additionally look into B_2 for any key belonging to bucket i and transfer it to q^*-heap of bucket i (that is delete it from B_2 and insert it into the q^*-heap). If the key x to be deleted is not found in the q^*-heap, we instead try to delete the key from B_2. By using this strategy we ensure that any insertion/deletion in any bucket takes at worst $O(\sqrt{\frac{\log n}{\log \log n}})$ time and still $O(1)$ time w.h.p.. Queries

can also be affected by overflown buckets. When the predecessor of a key x is to be searched in an overflown bucket i (that is when a predecessor search lands into bucket i with $c_i > C$) and the key is not found in the corresponding q^*-heap, then the key x is searched also in B_2 in time $O(\sqrt{\frac{\log n}{\log\log n}})$. The event of an overflowing bucket is very rare, so the time of queries remains $O(1)$ w.h.p..

Handling empty buckets. The data structure B_1 will help us handle a subtle problem which occurs when we have some empty buckets. Suppose that we have a non empty bucket i followed by a range of empty buckets $[i+1, j]$. Then the answer to any predecessor search directed towards any bucket $k \in [i+1, j]$ should return the largest element in bucket i. Thus in the highly unlikely case that a predecessor search lands in an empty bucket k (which is checked by verifying that $c_k = 0$) we will need to be able to efficiently compute the largest non empty bucket index i such that $i < k$ and this can precisely be done by querying B_1 for the value k which obviously will return i as B_1 is a predecessor data structure which stores precisely the index of non empty buckets and i is the largest non empty bucket index preceding k. This last step takes $O(\sqrt{\frac{\log \rho}{\log\log \rho}}) = O(\sqrt{\frac{\log n}{\log\log n}})$ time. What remains is to show how to maintain B_1. For that we only need to insert a bucket i into B_1 whenever it gets non empty after it was empty or to delete it from B_1 whenever it gets empty after it was non empty and those two events (a bucket becoming empty or a bucket becoming non empty) are expected to be rare enough that the time bound for any insertion/deletion remains $O(1)$ w.h.p..

References

[Ajt88] M. Ajtai. A lower bound for finding predecessors in Yao' s cell probe model. *Combinatorica*, 8:235–247, 1988.

[AFK84] M. Ajtai, M. Fredman and J. Komlós. Hash functions for priority queues. *Information and Control*, 63:217–225, 1984.

[AM93] A. Andersson and C. Mattsson. Dynamic Interpolation Search in $o(\log\log n)$ Time. In: *Proceedings of 20th Colloquium on Automata, Languages and Programming (ICALP '93)*, *Lecture Notes in Computer Science*, 700:15–27, 1993.

[BF99] P. Beame and F. E. Fich. Optimal Bounds for the Predecessor Problem. *STOC '99*, pp. 295–304.

[BF02] P. Beame and F. Fich. Optimal Bounds for the Predecessor problem and related problems. *Journal of Computer and Systems Sciences*, 65(1):38–72, 2002.

[BKZ77] P. van Emde Boas, R. Kaas and E. Zijlstra. Design and Implementation of an Efficient Priority Queue. *Mathematical Systems Theory*, 10:99–127, 1977.

[BFHM16] P. Bose, R. Fagerberg, J. Howat and P. Morin. Biased Predecessor Search. *Algorithmica*, 76(4):1097–1105, 2016.

[BHM13] P. Bose, J. Howat and P. Morin. A History of Distribution-Sensitive Data Structures. *Space-Efficient Data Structures, Streams, and Algorithms - Papers in Honor of J. I. Munro on the Occasion of His 66th Birthday*, *Lecture Notes in Ccomputer Science*, 8066:133–149, 2013.

[Cun15] V. Cunát. Predecessor problem on smooth distributions. At *https://arxiv.org/abs/1511.08598*, 2015.

[DJP04] E. Demaine, T. Jones and M. Patrascu. Interpolation Search for Non-Independent Data. *SODA '04*, pp. 522–523.

[Fel71] W. Feller. *An introduction to probability theory and its applications. Vol. II.* Second edition. John Wiley & Sons, Inc., New York-London-Sydney, 1971.

[Fre83] G. Frederickson. Implicit Data Structures for the Dictionary Problem. *Journal of the ACM*, 30(1):80–94, 1983.

[FKS84] M. Fredman, J. Komlos and E. Szemeredi. Storing a sparse table with O(1)worst case access time. *Journal of the ACM*, 31:538–544, 1984.

[FW90] M. L. Fredman and D. E. Willard. Blasting through the Information Theoretic Barrier with fusion trees. In: *Proceedings of the 26th Annual ACM Symposium on Theory of Computing (STOC '90)*, pp. 1–7, Baltimore, Maryland, USA.

[FW93] M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47:424–436, 1993.

[Gon77] G. Gonnet. *Interpolation and Interpolation-Hash Searching.* PhD Thesis, University of Waterloo, Waterloo, 1977.

[GRG80] G. Gonnet, L. Rogers and J. George. An Algorithmic and Complexity Analysis of Interpolation Search. *Acta Informatica*, 13:39–52, 1980.

[Gra06] G. Graefe. B-tree indexes, interpolation search, and skew. In: *Proceedings of the Workshop on Data Management on New Hardware (DaMoN '06)*, Chicago, Illinois, USA, June 25, 2006.

[IKR81] A. Itai, A. Konheim and M. Rodeh. A Sparse Table Implementation of Priority Queues. In: *Proceedings of the 8th ICALP, Lecture Notes in Computer Science*, 115:417–431, 1981.

[KMSTTZ03] A.C. Kaporis, C. Makris, S. Sioutas, A. Tsakalidis, K. Tsichlas and C. Zaroliagis. Improved bounds for finger search on a RAM. *ESA '03, Lecture Notes in Computer Science*, 2832:325–336, 2003.

[KMSTTZ06] A.C. Kaporis, C. Makris, S. Sioutas, A. Tsakalidis, K. Tsichlas and C. Zaroliagis. Dynamic Interpolation Search Revisited. *ICALP '06.*

[MT93] K. Mehlhorn and A. Tsakalidis. Dynamic Interpolation Search. *Journal of the ACM*, 40(3):621–634, 1993.

[Mil94] P. B. Miltersen. Lower bounds for Union-Split-Find related problems on random access machines. In: *Proceeding of the 26th Annual ACM Symposium on Theory of Computing (STOC '06)*, pp. 625–634, Montreal, Quebec, Canada, 1994.

[MNSW95] P. B. Miltersen, N. Nisan, S. Safra and A. Wigderson. On data structures and asymmetric communication complexity. *STOC '95*, pp. 103–111.

[MNSW98] P. B. Miltersen, N. Nisan, S. Safra, A. Wigderson. On Data Structures and Asymmetric Communication Complexity. *Journal of Computer and System Sciences*, 57(1):37–49, 1998.

[Pag00] R. Pagh. Faster deterministic dictionaries. In: *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '00)*, pp. 487–493, 2000.

[Pat08] M. Pătraşcu. Succincter. *FOC'S 08*, pp. 305–313.

[PT06a] M. Pătraşcu and M. Thorup. Time-space trade-offs for predecessor search. In: *Proceedings of the 38th Annual ACM Symposium on Theory of Computing (STOC' 06)*, Seattle, WA, USA, pp. 232–240, 2006.

[PT06b] M. Pătraşcu and M. Thorup. Time-Space Trade-Offs for Predecessor Search. At *https://arxiv.org/abs/cs/0603043*, 2006.

[PT07] M. Pătraşcu and M. Thorup. Randomization does not help searching predecessors. In: *Proceedings of the 18th ACM-SIAM Symposium on Discrete Algorithms (SODA '07)*, New Orleans, Louisiana, USA, pp. 555–564, 2007.

[PG92] Y. Perl, L. Gabriel. Arithmetic Interpolation Search for Alphabet Tables. *IEEE Transactions on Computers*, 41(4):493–499, 1992.

[PIA78] Y. Perl, A. Itai and H. Avni. Interpolation Search – A $\log \log N$ Search. *Communications of the ACM*, 21(7):550–554, 1978.

[PR77] Y. Perl, E. M. Reingold. Understanding the Complexity of the Interpolation Search. *Information Processing Letters*, 6(6):219–222, 1977.

[Pet57] W. W. Peterson. Addressing for Random Storage. *IBM Journal of Research and Development*, 1(4):130–146, 1957.

[Sen03] P. Sen. Lower bounds for predecessor searching in the cell probe model. *IEEE Conference on Computational Complexity*, pp. 73–83, 2003.

[SV08] P. Sen, S. Venkatesh. Lower bounds for predecessor searching in the cell probe model. *Journal of Computer and System Sciences*, 74:364–385, 2008.

[Yao81] A. C. Yao. Should tables be sorted? *Journal of the ACM*, 28(3):615–628, 1981.

[YY76] A. C. Yao and F. F. Yao. The Complexity of Searching an Ordered Random Table. In: *Proceedings of the 17th IEEE Symposium on Foundations of Computer Science (FOCS '76)*, pp. 173–177, 1976.

[Wil83] D. E. Willard. Log-Logarithmic Worst-Case Range Queries are Possible in Space $\Theta(N)$. *Information Processing Letters*, (17)2:81–84, 1983.

[Wil85] D. E. Willard. Searching Unindexed and Nonuniformly Generated Files in $\log\log N$ Time. *SIAM Journal on Computing*, 14(4):1013–1029, 1985.

[Wil92] D. E. Willard. Applications of the Fusion Tree Method to Computational Geometry and Searching. In: *Proceedings of the 3rd ACM-SIAM Symposium on Discrete Algorithms (SODA '92)*, pp.286–295, 1992.

[Xia92] B. Xiao. *New bounds in cell probe model.* PhD thesis, UC San Diego, 1992.