Dynamic Elias-Fano Representation*

Giulio Ermanno Pibiri[1] and Rossano Venturini[2]

1 **Computer Science Department, University of Pisa, Pisa, Italy**
`giulio.pibiri@di.unipi.it`
2 **Computer Science Department, University of Pisa, Pisa, Italy**
`rossano.venturini@unipi.it`

Abstract

We show that it is possible to store a dynamic ordered set $\mathcal{S}(n, u)$ of n integers drawn from a bounded universe of size u in space close to the information-theoretic lower bound and yet preserve the asymptotic time optimality of the operations. Our results leverage on the *Elias-Fano* representation of $\mathcal{S}(n, u)$ which takes $\mathsf{EF}(\mathcal{S}(n, u)) = n\lceil \log \frac{u}{n} \rceil + 2n$ bits of space and can be shown to be less than half a bit per element away from the information-theoretic minimum.

Considering a RAM model with memory words of $\Theta(\log u)$ bits, we focus on the case in which the integers of $\mathcal{S}$ are drawn from a polynomial universe of size $u = n^\gamma$, for any $\gamma = \Theta(1)$. We represent $\mathcal{S}(n, u)$ with $\mathsf{EF}(\mathcal{S}(n, u)) + o(n)$ bits of space and: 1. support static predecessor/successor queries in $\mathcal{O}(\min\{1 + \log \frac{u}{n}, \log\log n\})$; 2. make $\mathcal{S}$ grow in an append-only fashion by spending $\mathcal{O}(1)$ per inserted element; 3. support random access in $\mathcal{O}(\log n / \log\log n)$ worst-case, insertions/deletions in $\mathcal{O}(\log n / \log\log n)$ amortized and predecessor/successor queries in $\mathcal{O}(\min\{1 + \log \frac{u}{n}, \log\log n\})$ worst-case time. These time bounds are optimal.

1998 ACM Subject Classification E.1 Data Structures, E.4 Coding and Information Theory, F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases succinct data structures, integer sets, predecessor problem, Elias-Fano

Digital Object Identifier 10.4230/LIPIcs.CPM.2017.30

1 Introduction

The problem we consider is the one of representing in compressed space a dynamic ordered set $\mathcal{S}$ of n integer keys, which is a fundamental textbook problem (see the introduction to parts III and V of [8]). In general, any self-balancing search tree data structure, e.g., AVL or Red-Black tree, solves the problem optimally in the comparison model, by implementing all operations in $\mathcal{O}(\log n)$ worst-case time and using linear space [8]. However, by exploiting the fact that the stored keys are integers drawn from a bounded universe of size u, the problem is known to admit more efficient solutions in terms of asymptotic time complexity while still retaining linear space [8, 23, 26, 30, 13, 14]. Classical examples include the van Emde Boas tree [26, 27, 28], x/y-fast trie [30] and the fusion tree [14], that was the first data structure able to surpass the information-theoretic lower bound, by exhibiting an optimal [13] amount of time per operation within a number of memory words proportional to the size of the input. Some efforts have been spent in trying to reduce the space requirements of the representation [16, 18, 25] but known compressed solutions do not closely match the information-theoretic lower bound of the underlying integer set.

* This work was partially supported by the EU H2020 Program under the scheme INFRAIA-1-2014-2015: *Research Infrastructures*, grant agreement #654024 *SoBigData: Social Mining & Big Data Ecosystem* and by the *Pegaso* Project, POR FSE 2014-2020.

In this paper we show that it is possible to *preserve the optimal bounds* for the operations under almost *optimal space requirements*. The key ingredient of our data structures is the *Elias-Fano* representation of monotone integer sequences [10, 11]. In particular, Elias-Fano encodes a monotone integer sequence $\mathcal{S}(n,u)$ in $\mathsf{EF}(\mathcal{S}(n,u)) = n\lceil \log \frac{u}{n} \rceil + 2n$ bits, which can be shown to be less than half a bit per element away from optimality [10], maintaining the capability of randomly access an integer in $\mathcal{O}(1)$ worst-case time. The query $\mathsf{Predecessor}$, which, given an integer x, returns $\max\{y \in \mathcal{S} : y < x\}$, is possible as well over the compressed sequence in $\mathcal{O}(1 + \log \frac{u}{n})$ worst-case time. These properties make Elias-Fano extremely efficient on crucial practical applications, e.g., inverted indexes compression, just to mention the most noticeable one. Since inverted indexes can indeed be regarded as being a collection of sorted integer sequences, recent works [29, 20] have shown that Elias-Fano exhibits the best time/space trade-off thanks to its efficient search capabilities and strong theoretical guarantees. For this specific application, the operation that has to be supported efficiently is $\mathsf{Successor}(x) = \min\{y \in \mathcal{S} : y \geq x\}$, which is commonly called $\mathsf{NextGEQ}$ (Next Greater or EQual) [29, 20]. Throughout the paper we adopt the classical nomenclature and discuss $\mathsf{Predecessor}(x)$ as it is well known that the twin query $\mathsf{Successor}(x)$ is solved in a similar way.

The natural question is whether it is possible to extend the *static* Elias-Fano representation to *dynamic* scenarios, in which integers can also be inserted/deleted in/from $\mathcal{S}$. To this end, we consider the case in which the n integers of $\mathcal{S}$ are drawn from a *polynomial universe* of size $u = n^\gamma$, for any $\gamma = \Theta(1)$. This is the classical operational setting as considered by Fredman and Saks [13] (list representation problem) and let us concentrate on the typical case of practical interest. In order to characterize the asymptotic complexity of the data structures described in the paper and review the literature, we use a RAM model with word size $w = \Theta(\log u)$ bits. We also adopt the usual trans-dichotomous assumption [14], making w grow with n as needed. We maintain $\mathcal{S}(n,u)$ using $\mathsf{EF}(\mathcal{S}(n,u)) + o(n)$ bits of space, hence introducing a *sublinear* space overhead with respect to its static Elias-Fano representation, and show how:

1. static predecessor/successor queries can be supported in $\mathcal{O}(\min\{1 + \log \frac{u}{n}, \log\log n\})$ worst-case time (note that the first term of the bound, i.e., $\mathcal{O}(1 + \log \frac{u}{n})$, is optimal only for polynomial universes of size $u = n^\gamma$ with $1 \leq \gamma \leq 1 + \log\log n / \log n$);
2. to extend $\mathcal{S}$ in an append-only fashion, i.e., by assuming that integers are inserted in the data structure in sorted order, using a constant amount of work per integer;
3. to maintain $\mathcal{S}$ in a fully dynamic way, supporting random access in $\mathcal{O}(\log n / \log\log n)$ worst-case, insertions/deletions in $\mathcal{O}(\log n / \log\log n)$ amortized and predecessor/successor queries in $\mathcal{O}(\min\{1 + \log \frac{u}{n}, \log\log n\})$ worst-case time.

2 Related Work

We organize the discussion of the related work in three parts. The first part concerns the review of the results about the static predecessor problem. The second one explains in details the (static) Elias-Fano representation of monotone integer sequences because it forms the backbone of our solutions. The last part finally describes the results closest to our work for the maintenance of a dynamic integer set.

2.1 Static Predecessor Problem

We could solve the static predecessor problem in $\mathcal{O}(1)$ worst-case by storing all results to every possible query using *perfect hashing* [12] in $\mathcal{O}(u)$ words of space. In order to not

trivialize the problem, assume we have a polynomial space budget, e.g., we deal with a data structure occupying $\mathcal{O}(n^{\mathcal{O}(1)})$ words.

Ajtai [1] proved the first $\omega(1)$ lower bound, claiming that $\forall w$, $\exists n$ that gives $\Omega(\sqrt{\log w})$ query time. Only ten years later Beame and Fich [3, 4] proved two strong bounds for any *cell-probe* data structure[1]. They proved that $\forall w$, $\exists n$ that gives $\Omega(\log w / \log\log w)$ query time and that $\forall n$, $\exists w$ that gives $\Omega(\sqrt{\log n / \log\log n})$ query time. They also gave a static data structure achieving $\mathcal{O}(\min\{\log w / \log\log w, \sqrt{\log n / \log\log n}\})$ which is, therefore, optimal. Building on a long line of research, Pătraşcu and Thorup [21, 22] finally proved the following *optimal* space-time trade-off for a *static* data structure taking $m = n2^a w$ bits of space, with $a = \log\frac{m}{n} - \log w$

$$\Theta\Big(\min\Big\{\log_w n, \log\frac{w-\log n}{a}, \frac{\log\frac{w}{a}}{\log(\frac{a}{\log n}\log\frac{w}{a})}, \frac{\log\frac{w}{a}}{\log(\log\frac{w}{a}/\log\frac{\log n}{a})}\Big\}\Big). \tag{1}$$

This lower bound holds for cell-probe, RAM, trans-dichotomous RAM, external memory and communication game models. The first branch of the trade-off indicates that, whenever we are in RAM or external memory with one integer fitting in one memory word, fusion trees are optimal, as these require $\mathcal{O}(\log_w n) = \mathcal{O}(\log n / \log w)$ query time. The second branch holds for *polynomial universes*, i.e., whenever $u = n^\gamma$, for any $\gamma = \Theta(1)$. In such case we have that $w = \Theta(\log u) = \gamma \log n$, therefore y-fast tries [30] and van Emde Boas trees [26, 27, 28] are optimal with query time $\mathcal{O}(\log\log u) = \mathcal{O}(\log\log n)$. Finally, the last two branches of the trade-off treat the case for *super-polynomial universes*. In particular, the third branch matches the lower bound by Beame and Fich [3, 4] that requires $n^{\mathcal{O}(1)}$ words of space; the fourth branch improves this space occupancy, showing that $n^{1+1/\exp(\log^{1-\epsilon}\log u)}$ words are sufficient, for any $\epsilon > 0$.

2.2 Static Elias-Fano Representation

The integer encoding we describe in this section was independently proposed by Peter Elias [10] and Robert Mario Fano [11], hence its name. Given a monotonically increasing sequence $\mathcal{S}(n, u)$ of n positive integers drawn from a universe of size u (i.e., $\mathcal{S}[i-1] \leq \mathcal{S}[i]$, for any $1 \leq i < n$, with $\mathcal{S}[n-1] \leq u$), we write each $\mathcal{S}[i]$ in binary using $\lceil \log u \rceil$ bits. Each binary representation is then split into two parts: a *high* part consisting in the first $\lceil \log n \rceil$ most significant bits that we call *high bits* and a *low* part consisting in the remaining $\ell = \lfloor \log\frac{u}{n} \rfloor$ bits that we similarly call *low bits*. Let us call h_i and ℓ_i the values of high and low bits of $\mathcal{S}[i]$ respectively. The Elias-Fano representation of $\mathcal{S}$ is given by the encoding of the high and low parts. The array $L = [\ell_0, \ldots, \ell_{n-1}]$ is stored in fixed-width and represents the encoding of the low parts. Concerning the high bits, we represent them in *negated unary*[2] using a bit vector of $n + \lceil \frac{u}{2^\ell} \rceil \leq 2n$ bits as follows. We start from a $\mathsf{0}$-valued bit vector H and set the bit in position $h_i + i$, for all $i \in [0, n)$. The effect is that now the k-th unary integer m of H indicates that m integers of $\mathcal{S}$ have high bits equal to k. Finally the Elias-Fano representation of $\mathcal{S}$ is given by the concatenation of H and L and overall takes

$$\mathsf{EF}(\mathcal{S}(n, u)) = n\left\lceil \log\frac{u}{n} \right\rceil + 2n \text{ bits}. \tag{2}$$

[1] In the cell-probe computational model, described by Yao [31], computation is for free given that we only take into account word reads. It is not a very realistic model of computation, but it is useful to prove lower bounds because it is a stronger model than RAM and trans-dichotomous RAM.

[2] The negated unary representation of an integer x, is the bitwise NOT of its unary representation $U(x)$. An example: $U(5) = \mathsf{00001}$ and $\mathsf{NOT}(U(5)) = \mathsf{11110}$.

While we can opt for an arbitrary split ranging from 0 to $\lceil \log u \rceil$ into high and low parts, it can be shown that the value $\ell = \lfloor \log \frac{u}{n} \rfloor$ minimizes the overall space occupancy of the encoding [10]. As the information theoretic lower bound for a monotone sequence of n elements drawn from a universe of size u is $\lceil \log \binom{u+n}{n} \rceil \approx n \log \frac{u+n}{n} + n \log e$ bits, it can be shown that less than half a bit is wasted per element by the space bound in (2) [10]. Since we set a bit for every $i \in [0, n)$ in H and each h_i is extracted in $\mathcal{O}(1)$ time from $\mathcal{S}[i]$, it follows that $\mathcal{S}$ gets encoded with Elias-Fano in $\Theta(n)$ time.

Despite the simplicity of the encoding, it is possible to randomly access an integer from a sequence encoded with Elias-Fano *without* decompressing it. We refer to this operation as $\mathsf{Access}(i)$ in the following, which returns the i-th (smallest) element of the sequence. The operation is supported using an auxiliary data structure that is built on bit vector H, able to efficiently answer $\mathsf{Select}_1(i)$ queries, that return the position in H of the i-th 1 bit. This auxiliary data structure is *succinct* in the sense that it is negligibly small compared to $\mathsf{EF}(\mathcal{S}(n, u))$, requiring only $o(n)$ additional bits [7, 29].

Using the Select_1 primitive, it is possible to implement $\mathsf{Access}(i)$, which returns $\mathcal{S}[i]$ for any $i \in [0, n)$, in $\mathcal{O}(1)$. We basically have to re-link together the high and low bits of an integer, previously split up during the encoding phase. The low bits ℓ_i are trivial to retrieve as we need to read the range of bits $[i\ell, (i+1)\ell)$ from L. Note that we also need to store the quantity ℓ: a global redundancy of $\mathcal{O}(\log u)$ bits is sufficient. The retrieval of the high bits deserve, instead, a bit more care. Since we write in negated unary how many integers share the same high part, we have a bit set for every integer of S and a zero for every distinct high part. Therefore, to retrieve the high bits of the i-th integer, we need to know how many zeros are present in $H[0, \mathsf{Select}_1(i))$. This quantity is evaluated on H in $\mathcal{O}(1)$ as $\mathsf{Rank}_0(\mathsf{Select}_1(i)) = \mathsf{Select}_1(i) - i$. Notice, therefore, that the succinct rank/select data structure does not have to support Rank. Finally, linking the high and low bits is as simple as: $\mathsf{Access}(i) = ((\mathsf{Select}_1(i) - i) \ll \ell) \vee \ell_i$, where $\ll$ is the left shift operator and $\vee$ is the bitwise OR.

The query $\mathsf{Successor}(x)$ is supported in $\mathcal{O}(1 + \log \frac{u}{n})$ time[3], as follows. Let h_x be the high bits of x, i.e., its first $\lceil \log n \rceil$ most significant bits. Then $p_1 = \mathsf{Select}_0(h_x) - h_x$ represents the number of integers in $\mathcal{S}$ whose high bits are less than h_x. On the other hand, $p_2 = \mathsf{Select}_0(h_x + 1) - h_x - 1$ gives us the position at which the elements having high bits greater than h_x start. These two preliminary operations take $\mathcal{O}(1)$. We can now determine the successor of x by binary searching in this interval which may contain up to u/n integers. The algorithm for $\mathsf{Predecessor}(x)$ runs in a similar way. In particular, it could be that $\mathsf{Predecessor}(x)$ lies before the interval $[p_1, p_2)$: in this case $\mathcal{S}[p_1 - 1]$ is the element to return.

2.3 Dynamic Problems

We now review the most important results concerning the maintenance of a *dynamic* set of integers/binary strings, following the chronological order of their proposal.

The van Emde Boas tree is a recursive data structure that maintains $\mathcal{S}$ in $\mathcal{O}(u)$ words of space and supports the operations: Search which tests whether a given integer is present or not in $\mathcal{S}$, Insert/Delete and $\mathsf{Predecessor}$/$\mathsf{Successor}$ all in $\mathcal{O}(\log w)$ worst-case time [26, 27, 28]. Willard [30] improved the space bound to $\mathcal{O}(n)$ words by introducing the *y-fast trie* that supports Search and $\mathsf{Predecessor}$/$\mathsf{Successor}$ queries in $\mathcal{O}(\log w)$ worst-case time, Insert/Delete in amortized $\mathcal{O}(\log w)$ time.

[3] We report the bound as $\mathcal{O}(1 + \log \frac{u}{n})$, instead of $\mathcal{O}(\log \frac{u}{n})$, to cope with the case $n = u$.

The work by Fredman and Saks [13] is useful to understand which lower bounds apply to the problem we consider in the paper. They described the list representation problem, i.e., how to maintain $\mathcal{S}$ under the triad of operations Access/Insert/Delete, and proved that it can be solved in $\Omega(\log n/\log\log n)$ amortized time per operation if $w \leq \log^{\gamma} n$ for some γ. No space bound is posed on such problem. Their lower bound does not apply to dynamic predecessor queries and holds for the cell-probe computational model [31]. Extending the result to the dynamic predecessor problem, they proved that any cell-probe data structure representing $\mathcal{S}$ using $(\log u)^{\mathcal{O}(1)}$ bits per memory cell and $n^{\mathcal{O}(1)}$ worst-case time for insertions, requires $\Omega(\sqrt{\log n/\log\log n})$ worst-case query time. They also proved that on a RAM, the dynamic predecessor problem can be solved in $\mathcal{O}(\min\{\log\log n \cdot \log w/\log\log w, \sqrt{\log n/\log\log n}\})$, using $\mathcal{O}(n)$ words. This bound was matched by Andersson and Thorup [2] with the so-called *exponential search tree*. This data structure has an optimal bound of $\mathcal{O}(\sqrt{\log n/\log\log n})$ worst-case time for searching and updating $\mathcal{S}$, using polynomial space. Raman, Raman and Rao [24] also addressed the list representation problem[4] for arrays of length n by providing two solutions. Their first data structure supports Access in $\mathcal{O}(1)$ and Insert/Delete in $\mathcal{O}(n^{\epsilon})$ worst-case time for any fixed positive $\epsilon < 1$; the second data structure implements all the three operations in $\mathcal{O}(\log n/\log\log n)$ amortized time. Both data structures use $o(n)$ bits of redundancy and the time bounds are optimal.

Fredman and Willard [14] showed that dynamic predecessor queries can be answered in $\mathcal{O}(\log n/\log\log n)$ time by using the *fusion tree*. This data structure is a B-tree with branching factor $B = \Theta(\log n)$ that stores in each internal node a *fusion node*, a small data structure able of answering predecessor queries in $\mathcal{O}(1)$ for sets up to $w^{1/5}$ integers. Updating a fusion node takes, however, $\mathcal{O}(B^4)$ time. The overall space of the data structure is $\mathcal{O}(n)$ words. The work by Pătraşcu and Thorup [23] has recently shown that it is possible to "dynamize" the fusion node, by supporting Insert and Delete in $\mathcal{O}(1)$. As a result, they have proposed a data structure representing $\mathcal{S}$ in $\mathcal{O}(n)$ words and optimal $\mathcal{O}(\log n/\log w)$ running time for the operations Insert, Delete, Predecessor, Successor, Rank and Select.

We also mention a few additional results, that will be useful in the following. Bille *et al.* [5] recently combined the static solution of Demaine and Pătraşcu [9] with the one by Pătraşcu and Thorup [23] to support *dynamic prefix sums* over an array of size n in optimal $\mathcal{O}(\log n/\log(w/\delta))$ time per operation and linear space, where δ is the number of bits needed to encode the quantity that we sum to the elements of the array. Though not devised for integer sets, the *extended CRAM* (Compressed Random Access Memory) data structure described by Jansson, Sadakane and Sung [17] allows a string $\mathcal{S}$ of length n to be stored using its k-th order empirical entropy $nH_k(\mathcal{S})$ plus a redundancy of $\mathcal{O}(n\log\sigma(k\log\sigma + (k+1)\log\log n)/\log n)$ bits for every $0 \leq k < \log_{\sigma} n$, where σ is the size of the alphabet, in such a way that Insert/Delete of characters and Access to any consecutive $\log_{\sigma} n$ bits are all supported in optimal $\mathcal{O}(\log n/\log\log n)$ worst-case time. We will exploit the part of this work dedicated to the memory management. Grossi *et al.* [15] improved the previous space bound by using $nH_k(\mathcal{S}) + \mathcal{O}(n\log\log n/\log_{\sigma} n)$ bits and maintaining the asymptotic optimality for all operations. The paper by Navarro and Nekrich [19] illustrates a data structure supporting Access, Rank/Select queries, as well as symbol insertions/deletions on $\mathcal{S}$ in optimal $\mathcal{O}(\log n/\log\log n)$ time and taking $nH_0(\mathcal{S}) + \mathcal{O}(n + \sigma(\log\sigma + \log^{1+\epsilon} n))$ bits of space. Of particular interest for our purposes, is the data structure described in Appendix A.1 concerning the organization of data in small blocks. The high-level idea is to maintain a

[4] In their paper [24], the authors refer to the list representation problem, as introduced by Fredman and Saks [13], as the *dynamic array* problem. Also, the operation Access is named Index.

tree of constant height with node degree $\log^{\delta} n$, for some $0 < \delta < 1$, and leaves containing $o(\log n)$ elements each. As each internal node can fit in one machine word, the tree supports basic search operations in $\mathcal{O}(1)$ time by using a small pre-computed table. In Section 5 we will make use of a similar data structure, in order to handle mini blocks of sorted integers, which avoids the use of pre-computed tables.

3 Static Predecessor Queries in Optimal Time

In this section we are interested in determining the optimal running time of Predecessor for the Elias-Fano space bound in (2). As mentioned in Section 1, our focus is on polynomial universes, i.e., $u = n^{\gamma}$ for any $\gamma = \Theta(1)$, for which the second branch of the time/space trade-off in (1) becomes optimal. The following theorem shows that adding $o(n)$ bits of redundancy to $\mathsf{EF}(\mathcal{S}(n, u))$ is enough to support Predecessor queries in optimal time.

▶ **Theorem 1.** *There exists a data structure representing an ordered set $\mathcal{S}(n, u)$ of n integers drawn from a polynomial universe of size $u = n^{\gamma}$, for any $\gamma = \Theta(1)$, that takes $\mathsf{EF}(\mathcal{S}(n, u)) + o(n)$ bits of space and supports* Access *in $\mathcal{O}(1)$ worst-case and* Predecessor/Successor *queries in optimal $\mathcal{O}(\min\{1 + \log \frac{u}{n}, \log\log n\})$ worst-case time.*

We resort on the time/space trade-off (1) by Pătraşcu and Thorup [21, 22]. In our case, $a = \log(\lceil \log \frac{u}{n} \rceil + 2)$ and $w = \Theta(\log u) = \gamma \log n$. In such setting, the second term of the trade-off becomes $\log \frac{w - \log n}{a} = \log((\gamma - 1)\log n / \log(\lceil \log \frac{u}{n} \rceil + 2) = \mathcal{O}(\log\log n)$. This proves that y-fast tries and van Emde Boas trees are optimal for static Predecessor queries within the Elias-Fano space bound. However, such bound only depends on n, whereas the plain Elias-Fano bound for Predecessor of $\mathcal{O}(1 + \log \frac{u}{n})$, introduced in Subsection 2.2, depends on both n and u. On the other hand, the relation $u = n^{\gamma}$ relates the two parameter by means of the constant $\gamma = \Theta(1)$. It is clear that varying γ one of the two bounds becomes optimal. Indeed, comparing $1 + \log \frac{u}{n}$ with $\log\log n$, we have that $1 + \log \frac{u}{n} \leq \log\log n$ whenever $u \leq \frac{n}{2} \log n$, i.e., when $n^{\gamma - 1} \leq \frac{1}{2} \log n$. From this last condition we derive that the plain Elias-Fano is faster than van Emde Boas whenever $1 \leq \gamma \leq 1 + \frac{\log\log n}{\log n}$. In this case the static Elias-Fano representation does *not* need to be augmented. When, instead, $\gamma > 1 + \frac{\log\log n}{\log n}$, the query time $\mathcal{O}(\log\log n)$ is optimal and *exponentially better* than plain Elias-Fano. Therefore, $\mathcal{O}(\min\{1 + \log \frac{u}{n}, \log\log n\})$ is an accurate characterization of the Predecessor time bound.

We are left to describe a data structure matching the bound of $\mathcal{O}(\log\log n)$, within $o(n)$ bits of additional space. We divide $\mathcal{S}$ into $\lceil n / \log^2 u \rceil$ blocks of $\log^2 u$ integers each (the last block may contain less integers). We can solve Predecessor queries in a block in $\mathcal{O}(\log\log u) = \mathcal{O}(\log\log n)$ time by applying binary search. Now, we need a data structure on top of $\mathcal{S}$ that allows us to identify the proper block in $\mathcal{O}(\log\log n)$ time. Call the first element of a block its lower bound. We attach to $\mathcal{S}$ an y-fast trie storing the lower bounds of the blocks. More precisely, each leaf in the y-fast trie holds the lower bound of a block and its position in $\mathcal{S}$. The integers stored in the y-fast trie are $\lceil n / \log^2 u \rceil$, therefore its space is $\mathcal{O}(\frac{n}{\log^2 u} \log u) = o(n)$ bits. To identify the block where the predecessor of x lies in, we answer a partial Predecessor(x) query among the integers stored in the y-fast trie in $\mathcal{O}(\log\log n)$ worst-case time. The position p in $\mathcal{S}$ of the block's lower bound, associated to the identified partial answer, indicates that the search must continue in the block $\mathcal{S}[p, \min\{p + \log^2 u, n\})$.

Concluding this section, observe that the time bound for Predecessor queries is always at most $\mathcal{O}(\log\log n)$ except when $1 \leq \gamma \leq 1 + \frac{\log\log n}{\log n}$: in this case, the plain Elias-Fano

representation beats the time bound of $\mathcal{O}(\log\log n)$. Therefore, in what follows we report the bound as $\mathcal{O}(\min\{1+\log\frac{u}{n}, \log\log n\})$ but discuss the case for $\gamma > 1 + \frac{\log\log n}{\log n}$.

4 Extensible Elias-Fano Representation

When the integers are inserted in sorted order, we obtain an efficient *extensible* representation as these can only be added at the end of the sequence by means of an Append operation. This is a scenario of practical interest as it is the operational setting of append-only inverted indexes, e.g., the one of Twitter [6].

▶ **Theorem 2.** *There exists a data structure representing an ordered set $\mathcal{S}(n,u)$ of n integers, drawn from a polynomial universe of size $u = n^\gamma$, for any $\gamma = \Theta(1)$, that takes $\mathsf{EF}(\mathcal{S}(n,u)) + o(n)$ bits of space and supports:* Append *in $\mathcal{O}(1)$ amortized,* Access *in $\mathcal{O}(1)$ worst-case and* Predecessor/Successor *queries in optimal $\mathcal{O}(\min\{1+\log\frac{u}{n}, \log\log n\})$ worst-case time.*

We maintain an array B of size m in which integers are appended uncompressed. This array acts as a buffer, which is periodically encoded with Elias-Fano in $\Theta(m)$ time and dumped, so that new integers can be successfully appended. Each compressed representation of the buffer is appended in an array of blocks encoded with Elias-Fano. More precisely, when B becomes full we encode with Elias-Fano its corresponding *differential* buffer, i.e., the buffer whose values are $B[i] - B[0]$, $0 \leq i < m$. Each time the buffer is compressed, we append in another array C the pair $\langle \mathsf{base}, \mathsf{low\_bits}\rangle = \langle B[0], \lceil \log(B[m-1]/m)\rceil\rangle$, i.e., the buffer lower bound value and the number of bits needed to encode the average gap of the Elias-Fano representation of the buffer.

Apart from the space taken by the compressed blocks, the space of the data structure is given by the following contributions:

- $(m+1)\log u$ bits for the buffer B of uncompressed integers and its size;
- $\mathcal{O}(\lceil \frac{n}{m}\rceil \log n)$ bits for pointers to rank/select data structures, low and high bit arrays;
- $\mathcal{O}(\lceil \frac{n}{m}\rceil \log u)$ bits for the array C.

Summing up, the redundancy is $\mathcal{O}((m+1+\lceil \frac{n}{m}\rceil)\log u)$ bits. We use a buffer of size $m = \log^2 u$ and, as done in Section 3, we index the buffer lower bounds in an y-fast trie. More precisely, each leaf of the fast trie stores a buffer lower bound and the index of the compressed block to which the lower bound belongs to. The values stored in the y-fast trie are $\lceil n/\log^2 u\rceil$, thus requiring $o(n)$ bits of space. The redundancy $\mathcal{O}((m+1+\lceil \frac{n}{m}\rceil)\log u)$ becomes $o(n)$ bits for $n = \omega(\log^3 u)$, which is already satisfied by requiring that $\gamma = \Theta(1)$.

To take into account the space taken by the representation of the blocks, we use the property that splitting a block encoded with Elias-Fano into two sub-blocks *never* increases the cost of representation of the block. This is possible because each sub-block can be encoded with a universe *relative* to the sub-block, which is smaller than the original block universe, by subtracting to each integer the lower bound of the sub-block. The following property can be easily extended to work with an arbitrary number of splits.

▶ **Property 1.** *Consider a monotone sequence $\mathcal{S}$ of n integers. Let $\mathcal{S}[i,j)$ indicate the range of $\mathcal{S}$ delimited by endpoints i and j. Then for any i, k and j such that $0 \leq i < k < j < n$, we have $\mathsf{EF}(\mathcal{S}[i,k)) + \mathsf{EF}(\mathcal{S}[k,j)) \leq \mathsf{EF}(\mathcal{S}[i,j))$.*

Proof. Let m and u be respectively size and universe of the sub-sequence $\mathcal{S}[i,j)$, and, similarly, let m_1, m_2, u_1, u_2 be the sizes and universes of the two sub-sequences $\mathcal{S}[i,k)$ and $\mathcal{S}[k,j)$ respectively. We have that $m = m_1 + m_2$ and $u = u_1 + u_2$. From Subsection 2.2, we know that $\mathsf{EF}(\mathcal{S}[i,j))$ takes $m\phi + m + \lceil \frac{u}{2^\phi}\rceil$. Similarly $\mathsf{EF}(\mathcal{S}[i,k)) = m_1\phi_1 + m_1 + \lceil \frac{u_1}{2^{\phi_1}}\rceil$

and $\mathsf{EF}(\mathcal{S}[k,j)) = m_2\phi_2 + m_2 + \lceil \frac{u_2}{2^{\phi_2}} \rceil$. $\mathsf{EF}(\mathcal{S}[i,k))$ and $\mathsf{EF}(\mathcal{S}[k,j))$ are minimized by setting $\phi_1 = \lfloor \log \frac{u_1}{m_1} \rfloor$ and $\phi_2 = \lfloor \log \frac{u_2}{m_2} \rfloor$ respectively [10], therefore, by replacing ϕ_1 and ϕ_2 with ϕ, we have that $\mathsf{EF}(\mathcal{S}[i,k)) + \mathsf{EF}(\mathcal{S}[k,j)) \leq m_1\phi + m_2\phi + m_1 + m_2 + \lceil \frac{u_1}{2^\phi} \rceil + \lceil \frac{u_2}{2^\phi} \rceil = m\phi + m + \lceil \frac{u}{2^\phi} \rceil = \mathsf{EF}(\mathcal{S}[i,j))$. ◀

The operations are supported as follows. Since we compress the buffer each time it fills up (by taking $\Theta(m)$ time), Append is performed in $\mathcal{O}(1)$ amortized time. Appending new integers in the buffer accumulates a credit of $\mathcal{O}(\log^2 u)$ which largely pays the amortized cost $\mathcal{O}(\log \log u)$ of inserting a buffer lower bound into the y-fast trie. To Access the i-th integer, we retrieve the element x in position $i - jm$ from the compressed block of index $j = \lfloor \frac{i}{m} \rfloor$. This is done in $\mathcal{O}(1)$ worst-case time, since we know how many low bits are required to perform the access by reading $C[j]$.low_bits. We finally return the integer $x + C[j]$.base. Predecessor queries are supported similarly as in the description of Theorem 1. Given the integer x, we first resolve a partial Predecessor(x) query in the y-fast trie to identify the index j of the compressed block in which the predecessor is located. Then we return $C[j]$.base + Predecessor($x - C[j]$.base) by binary searching the block of index j in $\mathcal{O}(\log \log u) = \mathcal{O}(\log \log n)$ worst-case time.

From Theorem 2, the following corollary easily follows.

▶ **Corollary 3.** *There exists a data structure representing an ordered set $\mathcal{S}(n,u)$ of $n = \omega(\log^2 u)$ integers drawn from a universe of size u that takes $\mathsf{EF}(\mathcal{S}(n,u)) + o(n)$ bits of space and supports* Append *and* Access *operations in $\mathcal{O}(1)$ worst-case time.*

Without using the y-fast trie we are able to achieve a worst-case running time for the Append operation in Corollary 3 by using a classical de-amortization argument (note, however, that Predecessor queries are not supported in optimal time anymore). We maintain two buffers, B_1 and B_2, instead of one. When one is full we use the other to store the elements that must be appended. Suppose B_1 is full. For each of the successive m Append operations, we compress one element from B_1 and append the new integer in B_2. These two steps require $\mathcal{O}(1)$ worst-case time each.

5 Dynamic Elias-Fano Representation

In this section we describe how the static Elias-Fano representation can be turned into an efficient *dynamic* data structure, i.e., supporting Access, Insert, Delete, Minimum, Maximum, Predecessor and Successor in optimal time and taking $\mathsf{EF}(\mathcal{S}(n,u)) + o(n)$ bits of space.

As already discussed in Subsection 2.3, Fredman and Saks [13] proved that $\mathcal{O}(\frac{\log n}{\log \log n})$ amortized time is optimal for any data structure maintaining a set of integers subject to Access, Insert and Delete (list representation problem). Their result holds when $w \leq \log^\gamma n$ for some γ, which covers the case of polynomial universes $u = n^\gamma$ since $\gamma \leq \log^{\gamma-1} n$, for any $\gamma \geq 1$ and $n \geq 2$. We operate, therefore, in the same setting as Theorems 1 and 2, considering integers drawn from a polynomial universe of size $u = n^\gamma$, for any $\gamma = \Theta(1)$. In this setting, Pătraşcu and Thorup [21] showed that $\mathcal{O}(\log \log n)$ query time of y-fast tries and van Emde Boas trees is optimal for the dynamic predecessor problem too.

▶ **Theorem 4.** *There exists a data structure representing an ordered set $\mathcal{S}(n,u)$ of n integers drawn from a polynomial universe of size $u = n^\gamma$, for any $\gamma = \Theta(1)$, that takes $\mathsf{EF}(\mathcal{S}(n,u)) + o(n)$ bits of space and supports:* Access *in $\mathcal{O}(\log n / \log \log n)$ worst-case;* Insert/Delete *in $\mathcal{O}(\log n / \log \log n)$ amortized;* Minimum/Maximum *in $\mathcal{O}(1)$ and* Predecessor/Successor *queries in $\mathcal{O}(\min\{1 + \log \frac{u}{n}, \log \log n\})$ worst-case time. These time bounds are optimal.*

In what follows, we first describe the layout of the data structure and then analyze its space and time complexities.

5.1 Data Structure Description

We begin our description by showing how to handle a dynamic collection of mini blocks in succinct space, which is a key tool to obtain the full dynamic data structure. This result builds on an idea from [19], Appendix A.1.

5.1.1 Maintaining a Sorted Collection of Mini Blocks

Let $\mathcal{C}$ be a collection of $k = \mathcal{O}(\text{polylog}\, n)$ blocks of sorted integers, with the following properties. The blocks of $\mathcal{C}$ form a *total order*, i.e., $u_j \leq f_{j+1}$, for all $j = 1, \ldots, k-1$, where f_j and u_j indicate, respectively, the first and last element of the j-th block in the total order. Each block supports random access to its elements in constant time and is of size $\Theta(b) = \rho b$ with $\frac{1}{2} \leq \rho \leq 2$ and $b = \mathcal{O}(\text{polylog}\, n)$.

▶ **Lemma 5.** *The total order of the blocks of $\mathcal{C}$ can be maintained by using a data structure that takes $\mathcal{O}(\text{polylog}\, n \cdot \log\log n)$ bits of space and supports the following operations in $\mathcal{O}(\log\log n)$ worst-case time:* Search(x) *which returns a pointer to the block containing the integer x;* Access(i) *which returns the i-th integer of the total order;* Insert/Delete *of a block.*

Pointers of $\mathcal{O}(\log\log n)$ bits to blocks are stored in the leaves of a τ-ary tree $\mathcal{T}$, with $\tau = \Theta(\log^{\sigma} n)$ for some $0 < \sigma < 1$. Given that we have $\mathcal{O}(\text{polylog}\, n)$ leaves, the height of $\mathcal{T}$ is constant and equal to $\mathcal{O}(1/\sigma)$. $\mathcal{T}$ operates as a B-tree, in which internal nodes have $\Theta(\tau) = \rho\tau$ children.

Logically, we divide the information stored at each *internal node* into two levels of representation. For each of the two levels we store $\Theta(\tau)$ pairs, where the i-th pair maintains information about the sub-tree rooted in the i-th child. The pairs are stored following the order of the upper bounds of the blocks indexed in the sub-trees rooted in the node's children. In the lower level, each pair contains a pointer to the sub-tree rooted in the child and the size of such sub-tree. The $\Theta(\tau)$ children sizes are kept in prefix sums to enable binary search. In the upper level, each pair contains a pointer to the *right-most* block indexed in the sub-tree rooted in its child and the size of such sub-tree. Each *leaf* holds, of course, only the lower level of information. Each node uses $\mathcal{O}(\tau(\log\log n + \log \text{polylog}\, n)) = \mathcal{O}(\tau \log\log n) = o(\log n)$ bits, thus fitting in (less than) a machine word. The space taken by whole data structure is, therefore, $\mathcal{O}(\tau^{\mathcal{O}(1/\sigma)} \log\log n) = \mathcal{O}(\text{polylog}\, n \cdot \log\log n)$ bits.

We now detail how the operations are implemented. To support Search(x), i.e., determining the block where the integer x is comprised, we percolate $\mathcal{T}$, locating the correct child at each node in $\mathcal{O}(\log \tau) = \mathcal{O}(\log\log n)$ by binary searching on blocks' upper bounds. Specifically, if the upper bounds of the i-th block is needed for comparison for some $1 \leq i \leq \Theta(\tau)$, we access the block following the pointer (to the right-most block) of the i-th pair stored in the upper level of the node and we retrieve the upper bound in $\mathcal{O}(1)$, given that we also know the size of the block. When we have to insert/delete an integer, we identify the proper block of the total order in/from which the integer must be inserted/deleted in $\mathcal{O}(\log\log n)$ time (as described for the Search operation) and update the pairs along the path from the root in constant time, as these pairs fits in $o(\log n)$ bits overall. If a split or merge of a block happens, it is handled as usual and solved in a constant number of $\mathcal{O}(1)$-time operations. During an Access(i) query, we follow the proper root-to-leaf path in $\mathcal{T}$. The traversal of the data structure does not need to access the blocks directly, but instead uses their sizes to

determine the correct child at each level. By binary searching the sizes, we traverse the data structure in $\mathcal{O}(\log \log n)$ time. During the traversal of the path we also compute the sum Δ of the sizes of the preceding blocks by summing to the current value of Δ, at each level, the value stored in the $(j-1)$-th pair of the lower level if the j-th child is traversed. Finally we retrieve the $(i-\Delta)$-th integer from the identified block in $\mathcal{O}(1)$, as the blocks of $\mathcal{C}$ support random access.

5.1.2 Full Data Structure Layout

Let ℓ be $\log n / \log \log n$ for the rest of the paper. We logically divide the sorted sequence $\mathcal{S}(n,u)$ into mini blocks of $\Theta(\ell) = \rho\ell$ integers each. We organize the dynamic layout into two levels.

Lower level. We group $\mathcal{O}(\log^2 n)$ consecutive mini blocks together and index such collection using the data structure $\mathcal{T}$ described in Lemma 5. We refer to this collection as a "block" and say that $\mathcal{T}$ stores a block of $\mathcal{O}(\log^2 n)$ mini blocks. The set $\{\mathcal{T}_j\}_{j=1}^{k'}$, with $k' = n/\mathcal{O}(\ell \log^2 n)$, of all such data structures forms the *lower level* of the dynamic layout. Each $\mathcal{T}_j$ also stores the lower bound f_j of its block and the number of low bits required by its Elias-Fano representation in $\Theta(\log u)$ bits, so that we can subtract f_j to all the integers belonging to the mini blocks of $\mathcal{T}_j$.

Upper level. The set $\{f_j\}_{j=1}^{k'}$ of all the lower bounds of the blocks are indexed using an y-fast trie. The sizes of the blocks are maintained, instead, using the dynamic prefix sums data structure described in [5], which is a B-tree in which each node stores a dynamic prefix sums data structure operating on a small set of integers in $\mathcal{O}(1)$ time. In particular we use the operation $\mathsf{Update}(i,\Delta)$ of $\mathcal{P}$ as implemented in [5], which sums to the i-th integer of the data structure the quantity Δ (that fits in δ bits) and runs in optimal $\mathcal{O}(\log n/\log(w/\delta))$ worst-case time. In our setting this operation is supported in $\mathcal{O}(\ell)$ given that $\delta = \Delta = 1$.

These two data structures, respectively named $\mathcal{Y}$ and $\mathcal{P}$ in the following, form the *upper level* of the dynamic layout. The j-th leaf of $\mathcal{Y}$ and $\mathcal{P}$ stores a $\mathcal{O}(\log n)$-bit pointer to the data structure $\mathcal{T}_j$ in the lower level.

To handle the memory allocation for the mini blocks, we employ a different technique to manage the high and low part of their Elias-Fano representation. Recall from Subsection 2.2 that, given a sequence $\mathcal{S}(n,u)$, the high part of $\mathsf{EF}(\mathcal{S}(n,u))$ consists in a bitvector of at most $2n$ bits, whereas the low part is given by a vector of n $\lceil \log \frac{u}{n} \rceil$-bit integers. In our case, the high part of each mini block requires at most $2\ell = \mathcal{O}(w)$ bits and is stored using the data structure of Theorem 6 from [17] that allows to address and allocate the high part of a mini block in $\mathcal{O}(1)$ worst-case time. The low part of a mini block is instead stored using the data structure of Corollary 3 from [24] that supports Access in $\mathcal{O}(1)$ and $\mathsf{Insert}/\mathsf{Delete}$ in $\mathcal{O}(\ell^\epsilon)$ worst-case time for any fixed positive $\epsilon < 1$.

5.2 Space Analysis

The space required by the introduced layout will be clearly given by the contribution of:

- the data structures $\mathcal{Y}$ and $\mathcal{P}$ used in the upper level and the data structures $\mathcal{T}$ of Lemma 5 used in the lower level;
- the cost of representation of the mini blocks encoded with Elias-Fano;
- the overhead given by the mini blocks memory management.

In the following we separately analyze each contribution.

The space taken by the data structures $\mathcal{Y}$ and $\mathcal{P}$ in the upper level is $\mathcal{O}(\frac{n}{\ell \log^2 n} \log u) = o(n)$ bits. All the data structures $\mathcal{T}$ of Lemma 5 require $\mathcal{O}(\frac{n}{\ell \log^2 n} \log^2 n \log\log n) = o(n)$ bits too.

We now analyze the space taken by the encoding of the mini blocks. Since the universe of representation of a mini block could be as large as the one of its comprising block, i.e., u, storing the lower bounds of the mini blocks in order to use reduced universes (as already done for the blocks), would require $\mathcal{O}(\frac{n}{\ell} \log u)$ bits, which is too much. In what follows we show that it is not necessary to re-map the mini blocks using Property 1, hence these are kept encoded with the universe relative to their comprising block, if we carefully set the number of bits required to represent each *low part* in the Elias-Fano space bound (2). As pointed out previously, each low part in the Elias-Fano representation of a sequence $\mathcal{S}(n, u)$ is encoded using $\lceil \log \frac{u}{n} \rceil$ bits, which is the number of bits needed to encode the average gap u/n of $\mathcal{S}$. The number of bits for the average gap of a block is therefore $\lceil m \rceil = \lceil \log \frac{u}{\ell \log^2 n} \rceil$.

The idea is to choose a number of bits $\lceil m' \rceil$ for the encoding of the average gap of the mini blocks such that $\lceil m' \rceil = \lceil m \rceil$ for a sufficiently long sequence of p insertions/deletions. After p insertions/deletions have been performed, we rebuild the mini blocks using $\lceil m \rceil$ bits for the average gap. In other words, we want to guarantee that encoding the mini blocks with $\lceil m' \rceil$ bits for the average gap, instead of $\lceil m \rceil$, does *not* introduce any extra space. Since m' lies in the interval $[l, r] = [\log \frac{u}{\ell \log^2 n + p}, \log \frac{u}{\ell \log^2 n - p}]$, m' must be chosen in order to satisfy $\lceil m \rceil - 1 < m' < \lceil m \rceil$, which indeed implies $\lceil m' \rceil = \lceil m \rceil$. Precisely, we satisfy this condition by fixing $m' = m \pm \theta$ with $\lceil m \rceil - l < \pm\theta < \lceil m \rceil - r + 1$. To derive this condition, we distinguish three possible cases.

1. $[l, r] \subset [\lceil m \rceil - 1, \lceil m \rceil)$. In this case the condition $\lceil m \rceil - 1 < m' < \lceil m \rceil$ is already satisfied. The other two cases are symmetric.
2. $\lceil l \rceil = \lceil m \rceil - 1$. In this case we set $m' = m + \theta$. To let $\lceil m \rceil - 1 < m' < \lceil m \rceil$ holds, θ must be at least $\lceil m \rceil - l$ and at most $\lceil m \rceil + 1 - r$.
3. $\lceil r \rceil = \lceil m \rceil + 1$. In this case we set $m' = m - \theta$. To let $\lceil m \rceil - 1 < m' < \lceil m \rceil$ holds, θ must be at least $r - \lceil m \rceil - 1$ and at most $l - \lceil m \rceil$.

Cases 2. and 3. together yield the condition $\lceil m \rceil - l < \pm\theta < \lceil m \rceil - r + 1$.

Finally, we have to determine the proper number p of insertions/deletions before triggering the rebuilding of the mini blocks in order to attain to optimal insert/delete amortized time $\mathcal{O}(\ell)$. As blocks are of size $\Theta(\ell \log^2 n)$, p is chosen to be $\mathcal{O}(\log^2 n)$.

The techniques used to manage the memory allocation for the mini blocks introduce an overall redundancy of $o(n)$ bits. Precisely, the data structure of Theorem 6 from [17] has an overhead of $\mathcal{O}(w^4 + \frac{n}{\log n} \log^2 w) = o(n)$ bits, while the one of Corollary 3 from [24] uses $o(n)$ bits by choosing a proper positive $\epsilon < 1$.

In conclusion, by the above discussion and the use of Property 1, the space taken by the mini blocks can be safely upper bounded by $\mathsf{EF}(\mathcal{S}(n, u))$ and the redundancy sums up to $o(n)$ bits, so that the whole data structure requires $\mathsf{EF}(\mathcal{S}(n, u)) + o(n)$ bits of space.

5.3 Operations

In this subsection we describe how the operations of Theorem 4 are implemented. As stated before, ℓ is a short-hand for $\log n / \log\log n$.

To Access the i-th integer, we first resolve $\mathsf{Search}(i)$ on $\mathcal{P}$ in $\mathcal{O}(\ell)$: $\mathsf{Search}(i) = j$ indicates that the j-th block contains the i-th integer given that $\mathsf{Sum}(j-1) < i \leq \mathsf{Sum}(j)$, where $\mathsf{Sum}(j)$ equals the sum of the sizes of the first j blocks. We then follow the pointer stored in

the j-th leaf of $\mathcal{P}$, which points to the data structure $\mathcal{T}_j$. We finally Access the integer x of index $i - \mathsf{Sum}(j-1)$ from $\mathcal{T}_j$ in $\mathcal{O}(\log\log n)$ and return $x + f_j$. The overall complexity is, therefore, $\mathcal{O}(\ell)$. To Insert/Delete an integer x, we perform the following steps: 1. identify the proper data structure $\mathcal{T}_j$ by resolving a partial $\mathsf{Successor}(x)$ query on $\mathcal{Y}$ in $\mathcal{O}(\log\log n)$ and following the pointer retrieved at the identified leaf of $\mathcal{Y}$; 2. identify the correct mini block by $\mathsf{Search}(x - f_j)$ in $\mathcal{T}_j$ in $\mathcal{O}(\log\log n)$; 3. Insert/Delete $x - f_j$ in $\mathcal{T}_j$ by rebuilding the proper mini block in $\Theta(\ell)$; 4. update $\mathcal{P}$ in $\mathcal{O}(\ell)$. During the third step, split or merge of a mini block can happen and it is handled in $\mathcal{O}(\ell)$ worst-case time by the data structure $\mathcal{T}_j$; rebuilding of the mini blocks can happen as pointed out in the previous section and it is handled in $\mathcal{O}(\ell)$ amortized time. If split/merge of a block happens, the lower bound of the block is inserted/removed from $\mathcal{Y}$ in $\mathcal{O}(\log\log n)$ time. The overall complexity is, therefore, $\mathcal{O}(\ell)$ amortized. The query $\mathsf{Predecessor}(x)$ is supported as follows ($\mathsf{Successor}(x)$ runs in a similar way). We identify the proper data structure $\mathcal{T}_j$ in $\mathcal{O}(\log\log n)$ by answering a partial $\mathsf{Predecessor}(x)$ query on $\mathcal{Y}$ and following the pointer retrieved at the identified leaf of $\mathcal{Y}$. Then we identify the proper mini block by $\mathsf{Search}(x - f_j)$ in $\mathcal{T}_j$ in $\mathcal{O}(\log\log n)$ time. We finally return $f_j + \mathsf{Predecessor}(x - f_j)$ by binary searching on the identified mini block. The overall complexity is $\mathcal{O}(\log\log n)$ worst-case. The minimum and maximum elements of $\mathcal{S}$ are stored uncompressed using $\Theta(\log u)$ bits and returned when requested in $\mathcal{O}(1)$. Upon insertion/deletions these are updated as needed.

6 Conclusions

In this paper we have shown how the Elias-Fano representation of a monotone integer sequence $\mathcal{S}$ can be adapted to obtain optimal data structures in terms of query time and almost optimal in terms of space. In particular, when integers are drawn from a polynomial universe of size $u = n^\gamma$, for any $\gamma = \Theta(1)$, our data structures take the same asymptotic space of the plain, static, Elias-Fano representation, i.e., $\mathsf{EF}(\mathcal{S}(n,u)) + o(n)$ bits and support: 1. static Predecessor/Successor queries in optimal worst-case time $\mathcal{O}(\min\{1 + \log\frac{u}{n}, \log\log n\})$ (Section 3); 2. a $\mathcal{O}(1)$ worst-case amount of work for Append when integers are inserted in sorted order (Section 4); 3. Access in optimal $\mathcal{O}(\log n/\log\log n)$ worst-case time, Insert/Delete in optimal $\mathcal{O}(\log n/\log\log n)$ amortized time, Predecessor/Successor queries in optimal $\mathcal{O}(\min\{1 + \log\frac{u}{n}, \log\log n\})$ worst-case time (Section 5).

As a last note, we observe that the data structure described in Section 5 allows us to support all operations in time $\mathcal{O}(\log\log u)$ when *non*-polynomial universes are considered, i.e., when n and u are not necessarily related by means of the formula $u = n^\gamma$ for any $\gamma = \Theta(1)$. In this setting, the data structure of Lemma 5 will take $\mathcal{O}(\mathrm{polylog}\, u \cdot \log\log u)$ bits and operate in $\mathcal{O}(\log\log u)$ time. In order to guarantee an overall redundancy of $o(n)$ bits, we let mini blocks be of size $\Theta((\log\log u)^2)$ and group $\mathcal{O}(\log^2 u)$ consecutive mini blocks into a block. The high part of a mini block fits into one machine word, whereas we can insert/delete a low part in $\mathcal{O}((\log\log u)^{2\epsilon})$ for Corollary 3 of [24], which is $\mathcal{O}(\log\log u)$ as soon as $\epsilon < \frac{1}{2}$. Therefore, the following corollary matches the asymptotic time bounds of y-fast tries and van Emde Boas trees but in almost optimally compressed space.

▶ **Corollary 6.** *There exists a data structure representing an ordered set $\mathcal{S}(n,u)$ of n integers drawn from a universe of size u that takes $\mathsf{EF}(\mathcal{S}(n,u)) + o(n)$ bits of space and supports:* Access *and* Predecessor/Successor *queries in $\mathcal{O}(\log\log u)$ worst-case;* Insert/Delete *in $\mathcal{O}(\log\log u)$ amortized and* Minimum/Maximum *in $\mathcal{O}(1)$.*

References

1. Miklós Ajtai. A lower bound for finding predecessors in Yao's cell probe model. *Combinatorica*, 8(3):235–247, 1988. doi:10.1007/BF02126797.
2. Arne Andersson and Mikkel Thorup. Dynamic ordered sets with exponential search trees. *J. ACM*, 54(3):13, 2007. doi:10.1145/1236457.1236460.
3. Paul Beame and Faith E. Fich. Optimal bounds for the predecessor problem. In Jeffrey Scott Vitter, Lawrence L. Larmore, and Frank Thomson Leighton, editors, *Proceedings of the 31st Annual ACM Symposium on Theory of Computing (STOC 1999)*, pages 295–304. ACM, 1999. doi:10.1145/301250.301323.
4. Paul Beame and Faith E. Fich. Optimal bounds for the predecessor problem and related problems. *J. Comput. Syst. Sci.*, 65(1):38–72, 2002. doi:10.1006/jcss.2002.1822.
5. Philip Bille, Patrick Hagge Cording, Inge Li Gørtz, Frederik Rye Skjoldjensen, Hjalte Wedel Vildhøj, and Søren Vind. Dynamic relative compression, 2015. arXiv:1504.07851.
6. Michael Busch, Krishna Gade, Brian Larson, Patrick Lok, Samuel Luckenbill, and Jimmy Lin. Earlybird: Real-time search at Twitter. In Anastasios Kementsietsidis and Marcos Antonio Vaz Salles, editors, *Proceedings of the 28th IEEE International Conference on Data Engineering (ICDE 2012)*, pages 1360–1369. IEEE Computer Society, 2012. doi:10.1109/ICDE.2012.149.
7. David Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, Canada, 1996. URL: http://hdl.handle.net/10012/64.
8. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009. URL: http://mitpress.mit.edu/books/introduction-algorithms.
9. Erik D. Demaine and Mihai Pătraşcu. Tight bounds for the partial-sums problem. In J. Ian Munro, editor, *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2004)*, pages 20–29. SIAM, 2004. URL: http://dl.acm.org/citation.cfm?id=982792.982796.
10. Peter Elias. Efficient storage and retrieval by content and address of static files. *J. ACM*, 21(2):246–260, 1974. doi:10.1145/321812.321820.
11. Robert Mario Fano. On the number of bits required to implement an associative memory. Technical Report Memorandum 61, Computer Structures Group, MIT, Cambridge, MA, 1971. URL: http://csg.csail.mit.edu/pubs/memos/Memo-61/Memo-61.pdf.
12. Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst-case access time. *J. ACM*, 31(3):538–544, 1984. doi:10.1145/828.1884.
13. Michael L. Fredman and Michael E. Saks. The cell probe complexity of dynamic data structures. In David S. Johnson, editor, *Proceedings of the 21st Annual ACM Symposium on Theory of Computing (STOC 1989)*, pages 345–354. ACM, 1989. doi:10.1145/73007.73040.
14. Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.*, 47(3):424–436, 1993. doi:10.1016/0022-0000(93)90040-4.
15. Roberto Grossi, Rajeev Raman, Srinivasa Rao Satti, and Rossano Venturini. Dynamic compressed strings with random access. In Fedor V. Fomin, Rusins Freivalds, Marta Z. Kwiatkowska, and David Peleg, editors, *Proceedings of the 40th International Colloquium on Automata, Languages, and Programming (ICALP 2013)*, volume 7965 of *LNCS*, pages 504–515. Springer, 2013. doi:10.1007/978-3-642-39206-1_43.
16. Ankur Gupta, Wing-Kai Hon, Rahul Shah, and Jeffrey Scott Vitter. Compressed data structures: Dictionaries and data-aware measures. *Theor. Comput. Sci.*, 387(3):313–331, 2007. doi:10.1016/j.tcs.2007.07.042.

17 Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. CRAM: Compressed random access memory. In Artur Czumaj, Kurt Mehlhorn, Andrew M. Pitts, and Roger Wattenhofer, editors, *Proceedings of the 39th International Colloquium on Automata, Languages, and Programming (ICALP 2012)*, volume 7391 of *LNCS*, pages 510–521. Springer, 2012. doi:10.1007/978-3-642-31594-7_43.

18 Veli Mäkinen and Gonzalo Navarro. Rank and select revisited and extended. *Theor. Comput. Sci.*, 387(3):332–347, 2007. doi:10.1016/j.tcs.2007.07.013.

19 Gonzalo Navarro and Yakov Nekrich. Optimal dynamic sequence representations. In Sanjeev Khanna, editor, *Proceedings of the 24th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2013)*, pages 865–876. SIAM, 2013. doi:10.1137/1.9781611973105.62.

20 Giuseppe Ottaviano and Rossano Venturini. Partitioned Elias-Fano indexes. In Shlomo Geva, Andrew Trotman, Peter Bruza, Charles L. A. Clarke, and Kalervo Järvelin, editors, *Proceedings of the 37th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2014)*, pages 273–282. ACM, 2014. doi:10.1145/2600428.2609615.

21 Mihai Pătraşcu and Mikkel Thorup. Time-space trade-offs for predecessor search. In Jon M. Kleinberg, editor, *Proceedings of the 38th Annual ACM Symposium on Theory of Computing (STOC 2006)*, pages 232–240. ACM, 2006. doi:10.1145/1132516.1132551.

22 Mihai Pătraşcu and Mikkel Thorup. Randomization does not help searching predecessors. In Nikhil Bansal, Kirk Pruhs, and Clifford Stein, editors, *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2007)*, pages 555–564. SIAM, 2007. URL: http://dl.acm.org/citation.cfm?id=1283383.1283443.

23 Mihai Pătraşcu and Mikkel Thorup. Dynamic integer sets with optimal rank, select, and predecessor search. In Boaz Barak, editor, *Proceedings of the 55th IEEE Annual Symposium on Foundations of Computer Science (FOCS 2014)*, pages 166–175. IEEE Computer Society, 2014. doi:10.1109/FOCS.2014.26.

24 Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct dynamic data structures. In Frank K. H. A. Dehne, Jörg-Rüdiger Sack, and Roberto Tamassia, editors, *Proceedings of the 7th International Workshop on Algorithms and Data Structures (WADS 2001)*, volume 2125 of *LNCS*, pages 426–437. Springer, 2001. doi:10.1007/3-540-44634-6_39.

25 Kunihiko Sadakane and Roberto Grossi. Squeezing succinct data structures into entropy bounds. In Clifford Stein, editor, *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2006)*, pages 1230–1239. SIAM, 2006. doi:10.1145/1109557.1109693.

26 Peter van Emde Boas. Preserving order in a forest in less than logarithmic time. In Daniel J. Rosenkrantz, editor, *Proceedings of the 16th Annual Symposium on Foundations of Computer Science (FOCS 1975)*, pages 75–84. IEEE Computer Society, 1975. doi:10.1109/SFCS.1975.26.

27 Peter van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Inf. Process. Lett.*, 6(3):80–82, 1977. doi:10.1016/0020-0190(77)90031-X.

28 Peter van Emde Boas, Robert Kaas, and Erik Zijlstra. Design and implementation of an efficient priority queue. *Math. Syst. Theory*, 10:99–127, 1977. doi:10.1007/BF01683268.

29 Sebastiano Vigna. Quasi-succinct indices. In Stefano Leonardi, Alessandro Panconesi, Paolo Ferragina, and Aristides Gionis, editors, *Proceedings of the 6th ACM International Conference on Web Search and Data Mining (WSDM 2013)*, pages 83–92. ACM, 2013. doi:10.1145/2433396.2433409.

30 Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Inf. Process. Lett.*, 17(2):81–84, 1983. doi:10.1016/0020-0190(83)90075-3.

31 Andrew Chi-Chih Yao. Should tables be sorted? *J. ACM*, 28(3):615–628, 1981. doi:10.1145/322261.322274.