Predecessor Search

GONZALO NAVARRO and JAVIEL ROJAS-LEDESMA, Millennium Institute for Foundational Research on Data (IMFD), Department of Computer Science, University of Chile, Chile

The *predecessor* problem is a key component of the fundamental sorting-and-searching core of algorithmic problems. While binary search is the optimal solution in the comparison model, more realistic machine models on integer sets open the door to a rich universe of data structures, algorithms, and lower bounds. In this article, we review the evolution of the solutions to the predecessor problem, focusing on the important algorithmic ideas, from the famous data structure of van Emde Boas to the optimal results of Patrascu and Thorup. We also consider lower bounds, variants, and special cases, as well as the remaining open questions.

CCS Concepts: • **Theory of computation → Predecessor queries**; **Sorting and searching**;

Additional Key Words and Phrases: Integer data structures, integer sorting, RAM model, cell-probe model

ACM Reference format:
Gonzalo Navarro and Javiel Rojas-Ledesma. 2020. Predecessor Search. *ACM Comput. Surv.* 53, 5, Article 105 (September 2020), 35 pages.
https://doi.org/10.1145/3409371

1 INTRODUCTION

Assume we have a set X of n keys from a universe U with a total order. In the *predecessor* problem, one is given a *query* element $q \in U$, and is asked to find the maximum $p \in X$ such that $p \leq q$ (the *predecessor* of q). This is an extension of the more basic *membership* problem, which only aims to find whether $q \in X$. Both are fundamental algorithmic problems that compose the "sorting and searching" core, which lies at the base of virtually every other area and application in Computer Science (e.g., see References [7, 22, 37, 57, 58, 64, 74]). Just consider the following very basic problems: "What was the last message received before this time instant?"; "Where does this element fit in my ordered set?"; "Up to which job in this list can be completed within a time slot?" These questions boil down to instances of the predecessor problem.

The general goal is to preprocess X so that predecessor queries can be answered efficiently. Two obvious solutions, both to the predecessor and the membership problems, are to maintain X sorted on an array (in the *static case*, where X does not change) or in a balanced search tree (to efficiently support updates on X, in the *dynamic case*). These solutions yield $O(\log n)$ search time, and can be shown to be optimal if we have to proceed by comparisons. In the (rather realistic) case where

This work was funded by the Millennium Institute for Foundational Research on Data (IMFD), Chile, and by Project CONICYT Fondecyt/Postdoctorado No. 3190550.
Authors' addresses: G. Navarro and J. Rojas-Ledesma, Millennium Institute for Foundational Research on Data (IMFD), Department of Computer Science, University of Chile, Beauchef 851, Santiago, Chile; emails: {gnavarro, jrojas}@dcc.uchile.cl.

0360-0300/2020/09-ART105 $15.00
https://doi.org/10.1145/3409371

other strategies are permitted, particularly if U is a range of integers, the problems exhibit a much richer structure and fundamental differences. For example, membership queries can be solved in $O(1)$ time via perfect hashing [50], whereas this is impossible in general for predecessor queries.

The history of the predecessor problem, from the first data structure of van Emde Boas van Emde Boas [99] to the optimal results of Patrascu and Thorup and the currently open questions, is full of elegant and inspiring ideas that are also valuable beyond this problem. The techniques and data structures introduced for predecessor search have had great impact in problems like, for instance, integer sorting [9, 10, 51, 57, 58, 63, 96], string searching [15, 19, 27, 29, 30, 49, 66] and sorting [13, 24, 48, 61], various geometric retrieval problems [1, 35–39, 74], and representations of bit-vectors and string sequences with rank/select support [22, 54, 78, 84, 90]. This article is a gentle introduction to those developments, striving for simplicity without giving up on formal correctness. We assume the reader is familiar with basic concepts used to characterize the performance of algorithms (such as worst-case, expected and amortized running times).

We start with a brief summary of the current results and the main algorithmic ideas in Section 2, for the impatient readers. We then review in Section 3 the fundamental data structures for the predecessor problem, tracing the evolution from the first data structure of van Emde Boas [99] to the optimal results of Patrascu and Thorup [81–83]. The most relevant ideas and results on lower bounds for the problem are surveyed in Section 4. Finally, we cover in Section 5 some work on other variants, special cases, of the predecessor problem, and discuss some of the questions that remain open. Only a medium background in algorithmics is assumed from the reader.

2 SUMMARY

In the predecessor problem, we are asked to preprocess a finite set $X \subseteq U$, so that later, given any $q \in U$, we can efficiently compute $pred(X, q) = \max\{p \in X,\ p \leq q\}$. We call $n = |X|$, $u = |U|$, and will assume $U = \{0, 1, \ldots, u - 1\}$ for simplicity. Even though this integer universe might seem a very specific case, all objects manipulated by a standard conventional computer are treated at the lowest level as bit patterns that can be interpreted as integers. Basic data types (like string characters, or floating-point numbers) are designed so that the order induced by the integers representing the elements is the same as the natural order of the original universe (e.g., see Reference [85, Section 3.5]).

2.1 Models of Computation

The complexity of the predecessor problem, both in the static and dynamic settings, is well understood under the assumption that elements are abstract objects with a total order that can only be compared. In this model, balanced search trees support predecessor queries in $O(\log n)$ time, which is optimal by basic information-theoretic arguments [67, Section 11.2]. However, given the restrictive nature of this comparison model, such optimality might be misleading: in many cases the universe U is discrete, in particular a range of the integers, and then a realistic computer can perform other operations apart from comparing the elements. Thus, the predecessor problem is mainly studied in three models: the *word-RAM* and *external memory* models for upper bounds, and the *cell-probe model* for lower bounds.

The word-RAM model [55] aims to reflect the power of standard computers. The memory is an array of addressable words of w bits that can be accessed in constant time, and basic logic and arithmetic operations on w-bit integers consume constant time. Since memory addresses are contained in words, it is assumed that $w \geq \log u \geq \log n$ (logarithms are to the base 2 by default). The word-RAM is actually a family of models differing in the repertoire of instructions assumed to be constant-time. Addition, subtraction, bitwise conjunction and disjunction, comparison, and shifts are usually included. This is the case in the AC^0-RAM model, where only operations that implement functions computable by unbounded fan-in circuits of constant depth and size polynomial in w are available. This set of operations is usually augmented, for instance to include multiplication and division, which are not constant-time in AC^0-RAM. Most of the upper bounds for predecessor search on integer inputs were introduced in a word-RAM that includes multiplication and division.

In the external memory model [2], together with the main memory, the machine has access to an unbounded *external* memory divided into blocks that fit B words each, and the main memory can store at most M blocks simultaneously. The cost of evaluating logic and arithmetic operations is assumed to be marginal with respect to the cost of transferring blocks from/to the external memory. Thus, the cost of algorithms is given only by the numbers of blocks transferred between the main and external memories.

Finally, in the cell-probe model, as in word-RAM, the memory is divided into words of w bits, but the cost of an algorithm is measured only by the number of memory words it accesses, and computations have zero cost. Its simplicity makes it a strong model for lower bounds on data structures [80], subsuming other important models, including word-RAM and external memory.

2.2 The Short Story

In the static setting, where X does not change along time, Patrascu and Thorup [81, 82] completely solved the predecessor problem in a remarkable set of papers. In the dynamic setting, where X may undergo updates, there is still room for improvement [83]. Patrascu and Thorup [81] showed that in the word-RAM, given a set of n integers of l bits (i.e., $l = \log u$), the optimal predecessor search time of any data structure using $2^a n$ bits of space, for any $a \geq \log l$ is, up to constant factors,

$$1 + \min \begin{cases} \log_w n \\ \log \frac{l - \log n}{a} \\ \dfrac{\log \frac{l}{a}}{\log\left(\frac{a}{\log n} \cdot \log \frac{l}{a}\right)} \\ \dfrac{\log \frac{l}{a}}{\log\left(\log \frac{l}{a} / \log \frac{\log n}{a}\right)} \end{cases} . \tag{1}$$

They introduce a matching deterministic lower bound in the cell-probe model for the static case, which holds even under randomized query schemes [82]. Thus, this bound is optimal under any selection of n, l, w, and a, and even if randomization is allowed.

In the dynamic setting, Patrascu and Thorup [83] described a data structure with optimal expected time (i.e., matching Equation (1)) for $l \leq w$ (considering the time as the maximum between updates and queries). The worst-case optimal running times of these operations are still open.

These optimality results also hold for the external memory model (replacing w by B in the first branch). Their static and dynamic lower bounds apply to the number of cell-probes that the query algorithm must make to the portion of memory where the data structure resides. By interpreting "cells probed" as "blocks transferred to main memory," the lower bounds apply to external memory. Moreover, any algorithm running in time $T(n)$ in a word-RAM can trivially be converted into an algorithm in external memory performing at most $T(n)$ I/Os. Such bounds are usually sub-optimal but, surprisingly, a simple modification of the optimal word-RAM data structures of Patrascu and Thorup yields an optimal data structure in the external memory model as well.

Some interesting simplified cases of Equation (1), using linear space (i.e., $O(n \log u)$ bits, implying $a = \log \log u + O(1)$) are, in each of the four branches of the formula: (1) constant if X is small compared to the machine word, $n = w^{O(1)}$; (2) $O(\log \log(u/n))$, decreasing as X becomes denser in U and reaching constant time when $n = \Theta(u)$; (3 and 4) $o(\log \log u)$ if X is very small compared to U. A simple function of u that holds as an upper and lower bound for any n is $\Theta(\log \log u)$, which is reached for example if $n = \sqrt{u}$. Note, however, that we can reach constant time by using $O(u)$ bits of space, if we set $a = \log(u/n)$. This is the classical solution for `rank` on bitvectors [41, 75].

2.3 Main Techniques and Data Structures

Two main techniques are used to support predecessor queries: *length reduction* and *cardinality reduction* [16]. Intuitively, in the first one the size of U is reduced recursively, while in the second one the size of X is reduced recursively. Data structures implementing length reduction are essentially based on tries (or digital trees) [64, Chapter 6.3] of height depending on u, while the ones implementing cardinality reduction are mainly based on B-Trees (or perfectly balanced multiary trees) [43, Chapter 18] of height depending on $|X|$. The two main representatives of these data structures are the van Emde Boas tree [101], and the fusion tree [51], respectively.

A van Emde Boas tree [101] is a trie of height $\log u$ in which the leaves are in a one-to-one correspondence with the elements of U. To store X, the leaves corresponding to elements in the set, and their ancestors, are bit-marked. Predecessor queries are supported by inspecting these marks via binary search on the levels. Since the height of the tree is $\lceil \log u \rceil$, this search takes $O(\log\log u)$ time. The main disadvantage of this data structure is that it uses $O(u)$ space (measured in words by default). Various improvements have been proposed in this direction. For instance, Willard presented the x-fast trie [102], a variant of van Emde Boas trees, which requires only $O(n \log u)$ space. They also introduced the y-fast trie, which combines an x-fast trie with balanced search trees to reduce the space to $O(n)$. The idea is to create an ordered partition of the set into $O(n/\log u)$ slots, chose one representative element from each slot (e.g., the minimum) and store them in an x-fast trie, and store each slot independently in a balanced search tree. Both of Willard's variants [102] of the van Emde Boas tree perform membership and predecessor queries in worst-case $O(\log\log u)$ time, and y-fast tries achieve amortized $O(\log\log u)$ update time in expectation (due to the use of hashing to store the levels of the x-fast trie). Combining Willard's variants [102] with table-lookup, Patrascu and Thorup [81] achieved the bound in the second branch of their optimal tradeoffs.

In an orthogonal direction, Fredman and Willard [51] introduced fusion trees, basically a B-Tree of degree depending on w. They showed how to pack w^{ε} keys into one single word such that predecessor queries among the keys are supported in constant time, for any $\varepsilon \leq 1/5$. This allows to simulate a B-Tree of degree w^{ε}, supporting predecessor queries in $O(\log_w n)$ time, while using only $O(n)$ space. Note that this query time is within $O(\log n/\log\log n)$, since $w = \Omega(\log n)$, and thus fusion trees asymptotically outperform binary search trees. Andersson [9] improved this bound by means of another data structure implementing cardinality reduction: the exponential search tree. This data structure is basically a multi-way search tree of height $O(\log\log n)$ in which the maximum degree of a node, instead of being fixed (as in fusion trees), decreases exponentially with the depth of the node. Combined with fusion trees, an exponential search tree supports predecessor queries in $O(\sqrt{\log n})$ time. More importantly, exponential search trees serve as a general technique to reduce the problem of searching predecessors in a dynamic set using linear space to the static version of the problem using polynomial space. This turned out to be a powerful tool for predecessor queries and integer sorting [12, 57].

Another key result was presented by Beame and Fich [14], who combined a fusion tree with a variant of the x-fast trie (thereby combining length and cardinality reduction). They replace the binary search in the levels of the x-fast trie by a multi-way search that, using the power of parallel hashing, can examine several different levels of the trie at once. Later, Patrascu and Thorup [81] refined and combined all these results to obtain their optimal bounds: the first branch resulting directly from fusion trees third and fourth branches of their optimal bounds.

3 DATA STRUCTURES

The two main techniques used in the design of data structures for the predecessor problem are length reduction and cardinality reduction [16]. None of the data structures based on exclusively one of these two techniques is optimal under all the regimes of the parameters. Achieving such an optimal data structure required an advanced combination of both techniques. We give an outlook of the main data structures for the predecessor problem based on length reduction and cardinality reduction in Sections 3.1 and 3.2, respectively. We then present in Sections 3.3 and 3.4 some of the data structures based on combining these two techniques, including the optimal data structure of Patrascu and Thorup [81]. All the data structures are described in the word-RAM model, unless otherwise specified. Some of the data structures make use of hash tables. In the static case, for those solutions we will assume perfect hash tables with deterministic constant-time lookup, which can be constructed in $O(n)$ expected time [50], and $O(n(\log\log n)^2)$ deterministic worst-case time [91]. Thus, for such data structures, query-time upper bounds will always be for the worst case.

3.1 Predecessor Search via Length Reduction

Data structures implementing length reduction are essentially based on tries of height depending on u. The first data structure implementing this concept for the predecessor problem was the van Emde Boas tree, originally introduced by van Emde Boas in 1977 [99] and studied today in undergraduate courses of algorithms [43, Chapter 20]. Van Emde Boas trees support predecessor queries and updates in $O(\log\log u)$ time, but their major drawback is that they use $\Theta(u)$ words, which may be too large. We give a brief overview of van Emde Boas trees in Section 3.1.1, and then present in Section 3.1.2 some of the data structures that improve their space usage while preserving their query and update time.

3.1.1 Van Emde Boas trees. The van Emde Boas tree was one of the first data structures (and algorithms in general) that exploited bounded precision to obtain faster running times [77], and they support predecessor queries and updates in $O(\log\log u)$ time. There are two main approaches to obtain this time: the cluster-galaxy approach and the trie-based approach.

Cluster-galaxy approach. The most popular one is a direct recursive approach introduced by Knuth [65] (as acknowledged by van Emde Boas [100]). In this approach, the universe is seen as a "galaxy" of $\sqrt{u}$ "clusters," each containing $\sqrt{u}$ elements. A van Emde Boas tree T over a universe U is a recursive tree that stores at each node:

- `T.min, T.max`: The minimum and maximum elements of U inserted in the tree, respectively. If the tree is empty, then `T.min` $= +\infty$, and `T.max` $= -\infty$. The value `T.max` is not stored recursively down the tree.
- `T.clusters`: An array with $\sqrt{u}$ children (one per cluster). The ith child is a van Emde Boas tree over a universe of size $\sqrt{u}$ representing the elements in the range $[i\sqrt{u}, (i+1)\sqrt{u}-1]$, for all $i \in [0, \sqrt{u}-1]$.
- `T.galaxy`: A van Emde Boas tree over a universe of size $\sqrt{u}$ storing which children (i.e., clusters) contain elements, and supporting predecessor queries on this information.

One can also think of clusters and galaxies in the following way: the galaxy is formed by the distinct values of the $\frac{\log u}{2}$ higher bits of the elements, and each such value c is associated with a cluster formed by the $\frac{\log u}{2}$ lower bits of the elements whose higher bits are c. The clusters are then divided recursively.

Algorithm 1 shows how the predecessor is found with this structure. Each call is decomposed into one recursive call at the cluster or at the galaxy level, but not both. The time complexity

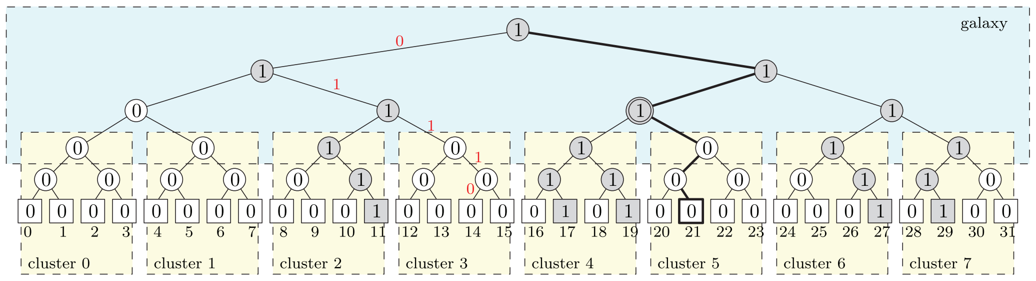


Fig. 1. A complete binary tree storing the set $X = \{11, 17, 19, 27, 29\}$ from the universe $U = \{0, 1, \ldots, 31\}$. The nodes containing elements from X, and their ancestors, are bit-marked with a 1. If the edges are labeled with 0 or 1, depending on whether they point to the left or the right children, respectively, then the concatenation of the labels in a root-to-leaf path yields the binary representation of the leaf (e.g., see the path down to 14 = 01110). Thus, the tree corresponds to a complete trie representing U. In the cluster-galaxy approach, the galaxy corresponds to the top half of the tree, while the clusters correspond to the trees rooted at the leaves of the galaxy. The root-to-leaf path corresponding to the element $q = 21$ (bit-marked as 111000) is highlighted with bold lines, as is the exit node of q with double lines (corresponding to the last 1 in the bit-marks of the path).

ALGORITHM 1: vEB_predecessor(T, q)

```
1: if $q \geq$ T.max then
2:     return T.max;
3: let cluster$_q \leftarrow \lfloor q/\sqrt{u} \rfloor$, low$_q \leftarrow (q \mod \sqrt{u})$, and high$_q \leftarrow \sqrt{u} \cdot$ cluster$_q$
4: if low$_q \geq$ T.clusters[cluster$_q$].min then
5:     return high$_q$+ vEB_predecessor(T.clusters[cluster$_q$], low$_q$)
6: return high$_q$ + T.clusters[vEB_predecessor(T.galaxy, cluster$_q$)].max
```

then satisfies the recurrence $T(u) = T(\sqrt{u}) + O(1)$, which solves to $O(\log \log u)$. Insertions and deletions are handled analogously.

This recursive approach requires address computations, though, which in turn require multiplications, and these were not taken as constant-time in the RAM models in use by the time of the original article [100]. To avoid multiplications, van Emde Boas described his solution based on tries. Today, instead, constant-time multiplications are regarded as perfectly acceptable [43, Section 2.2].

Trie-based approach. In the original approach [99–101], the elements of U are represented by a complete binary tree whose leaves in left-to-right order correspond to the elements of u in ascending order, respectively (see Figure 1). By assigning the label 0 (respectively, 1) to the edge connecting with the left (respectively, right) child of each node, the tree becomes a complete trie storing the binary representation of the elements of U. For each node of the tree (internal or leaf) a bit-mark is stored. A set X is represented by marking the leaves corresponding to its elements together with all its ancestors up to the root. Additionally, for every internal node v with bit-mark 1, two pointers are stored, pointing to the minimum and maximum marked leaves of the tree rooted at v. Finally, the leaves corresponding to consecutive elements in X are connected using a doubly linked list.

For any element $q \in U$, consider the sequence s_q of $h = \lceil \log u \rceil$ bit-marks in the path from the root to the leaf corresponding to q. There must be an index $j \in [0, h-1]$ such that $s_q[i] = 1$ for all $i \leq j$, and $s_q[k] = 0$ for all $k > j$ (i.e., s_q is of the form $1^j 0^{h-j}$). For such a j, the jth node in the path from the root of the tree to q is named the *exit node* of q. Note that if we can locate the exit node e of q, then the predecessor and successor of q can be computed in constant time using the pointers to the minimum and maximum leaves descending from e, and the doubly linked list connecting the leaves.

The idea of van Emde Boas to efficiently locate the exit node was to use binary search on the paths, a method inspired in the algorithm to find lowest common ancestors introduced by Aho et al. [3]. A simple way to perform this type of binary search on the levels is to store the levels in a two-dimensional array. Since the size of the paths is h, such a binary search can be implemented in $O(\log h)$ time, which is $O(\log \log u)$. However, this solution requires address computations, and therefore multiplication operations, which van Emde Boas was trying to avoid.

To achieve this running time without multiplications the solution was to decompose the tree into so-called *canonical subtrees*, a recursive subdivision of the tree into a top tree of height $h/2$ corresponding to the first $h/2$ levels, and $\sqrt{u}$ bottom trees of height $h/2$, whose roots were precisely the leaves of the top tree. The top tree represents, for each of the $\sqrt{u}$ different values of the leftmost $h/2$ bits of the elements of U, whether they appear in the set X or not. Similarly, for each of those different values of the leftmost $h/2$ bits, the respective bottom tree stores which of $\sqrt{u}$ different values of the rightmost $h/2$ bits are present in X. [1] The decomposition of the tree into canonical subtrees was also key to allow updates in $O(\log \log u)$ time, because marking all the bits of the affected path in the original tree would require $\Theta(\log u)$ time after each insertion or deletion. For the complete details on how these trees are stored and maintained, we refer the reader to van Emde Boas' original article [99].

Modern implementations of the van Emde Boas tree and its variants use hash tables to store the levels, to reduce the space required by the data structure while still supporting the binary searches on the levels efficiently (although the running time guarantees obtained are "with high probability" instead of worst-case). We explore some of these variants next.

3.1.2 *Reducing the Space of van Emde Boas Trees.*

X-fast tries: almost linear space, but with slow updates. In 1983, Willard [102] introduced a variant of van Emde Boas' data structure that uses $O(n)$ space while preserving the running times, under the name of "y-fast tries." As a first step towards his result, Willard [102] introduced a simpler data structure, the x-fast trie, in which the space used is almost linear, but updates are slow. Like van Emde Boas trees, an x-fast trie is a trie whose leaves correspond to the elements of U (present in X), and any root-to-leaf path yields the binary representation of the element at the leaf. The height of the x-fast trie is then $O(\log u)$ as well, but it has only $|X|$ leaves instead of u.

The first key idea to reduce the space was to maintain each level of the tree in a hash table. For each $l \in [1, \log u]$, a hash table H_l stores the prefixes of length l of every element in X, associated with the respective node in the trie at the lth level. By binary searching on these $\log u$ hash tables, one can find the exit node of any search key q in $O(\log \log u)$ time. By definition, the exit node of q cannot be a branching node. To navigate in constant time from the exit node to the predecessor or successor of q, each non-branching node with no left child (respectively, right child) points to the smallest leaf (respectively, largest leaf) in its subtree. As in the original van Emde Boas tree, the leaves are connected using a doubly linked list (see Figure 2). Given that each of the n elements of X appears in $O(\log u)$ hash tables, and since the trie has only $O(n \log u)$ nodes, the x-fast trie uses $O(n \log u)$ space in total.

Let $hi(q, l)$ be the l most significant bits of integer q. To find the predecessor in X of a query q, a binary search locates the exit node v of q, which corresponds to the largest l such that $hi(q, l) \in H_l$,

[1] The top and bottom trees correspond to the galaxy and clusters, respectively, in the variant described by Knuth [65].

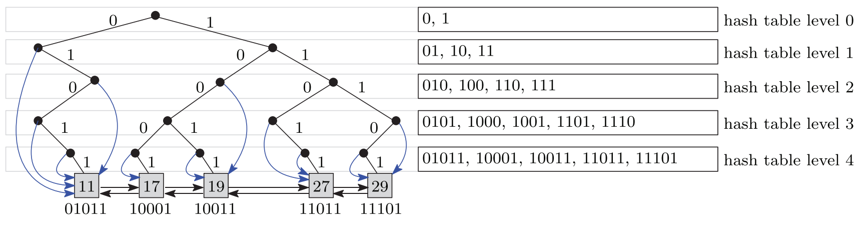


Fig. 2. An x-fast trie storing the set $X = \{11, 17, 19, 27, 29\}$ from the universe $U = \{0, 1, \ldots, 31\}$.

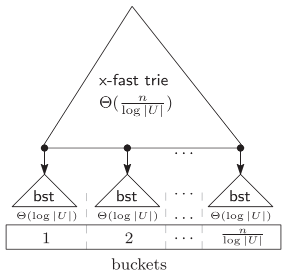


Fig. 3. An illustration of the bucketing technique in y-fast tries. The n elements of X are partitioned into $\Theta(n/\log u)$ equally sized buckets, which are stored using balanced binary search trees (bst). Only one (representative) element of each bucket is inserted in an x-fast trie.

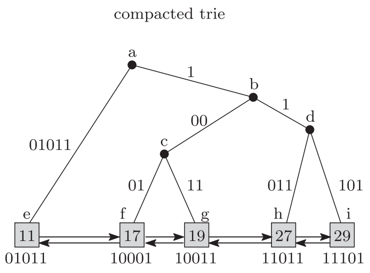
the hash table for level l. If v is a leaf, then the search is complete. Otherwise, v must be a non-branching node (otherwise, it could not be the deepest node prefixing q), In this case, v stores a pointer to the largest (or smallest) leaf in its subtree, which leads to either the predecessor or the successor of the query. Since the leaves are doubly linked, in either case the predecessor of q is found easily in constant time. Therefore, the total query time is within $O(\log \log u)$: the binary search among the hash tables takes $O(\log \log u)$ time, and the subsequent operations take just $O(1)$ additional time.

While x-fast tries drive the space from the $O(u)$ of van Emde Boas trees to $O(n \log u)$, they still do not reach linear space. Another drawback is that, during an insertion or deletion in an x-fast trie, the $O(\log u)$ hash tables, and the pointers to the largest/smallest leaves of the branching nodes in the affected path, must be updated. Thus, these operations take $O(\log u)$ expected time.

Y-fast tries: linear space and faster (amortized) updates. To overcome the space and update time inconveniences of the x-fast trie, Willard [102] used a (nowadays standard) bucketing trick. The n elements of X are separated into $\Theta(n/\log u)$ buckets of $\Theta(\log u)$ elements each. Each bucket is stored in a balanced search tree, and a representative element of each bucket (e.g., the minimum) is inserted in an x-fast trie (see Figure 3). This new data structure was called y-fast trie. Since the number of elements stored in the x-fast trie is $O(n/\log u)$, and each of the balanced search trees uses linear space, the total space of the y-fast trie is then within $O((n/\log u) \cdot \log u) = O(n)$.

To search for the predecessor of a key q, one first searches within the x-fast trie (in $O(\log \log u)$ time) to locate the bucket b to which the predecessor of q belongs. Since each of the bucket is represented as a balanced search with $O(\log u)$ elements, the predecessor of q in b can then be easily found in $O(\log \log u)$ additional time.

To analyze the running time of updates, note that insertions and deletions within the binary search trees take $O(\log \log u)$ time. The binary search trees are rebuilt when their sizes double (during insertions) or quarter (during deletions). These rebuilding operations require $O(\log u)$ time,



hash table

key	data
ϵ	a
1	b
10	c
11	d
0101	e
1000	f
1001	g
1101	h
1110	i

α	$l(\alpha)$	$p(\alpha)$	$[\lvert l(p(\alpha))\rvert + 1, \lvert l(\alpha)\rvert]$	key(α)
a	ϵ		[0,0]	ϵ
b	1	a	[1,1]	a
c	100	b	[2,3]	10
d	11	b	[2,2]	11
e	01011	a	[1,5]	0101
f	10001	c	[4,5]	1000
g	10011	c	[4,5]	1001
h	11011	d	[3,5]	1101
i	11101	d	[3,5]	1110

Fig. 4. An illustration of a z-fast trie storing the set $X = \{11, 17, 19, 27, 29\}$ from the universe $U = \{0, 1, \ldots, 31\}$. The pointers that allow efficiently finding the smallest and largest elements descending from an internal node have been omitted.

but because of the frequency with which they are performed (once in at least $\Theta(\log u)$ operations), their amortized cost is constant per operation. Similarly, insert and delete operations in the x-fast trie cost $O(\log u)$ expected time, but because they are carried out only when a new binary tree is built or an existing one is deleted, their amortized cost is also $O(1)$ in expectation. Thus, insertion and deletions in a y-fast trie require expected amortized time $O(\log \log u)$.

Mehlhorn and Näher: hashing on the cluster/galaxy approach. In 1990, Mehlhorn and Näher [69] showed that the same $O(\log \log u)$ query and amortized time of y-fast tries could be achieved in linear space, via a simple modification to the original van Emde Boas tree. Their solution was based on the cluster/galaxy approach, and on the power of perfect hashing. The idea was to store the $\sqrt{u}$ van Emde Boas trees that represent the clusters of a galaxy in a hash table, instead of storing them in an array so that no space is spent to store empty clusters. This simple idea reduces the space from $\Theta(u)$ to $\Theta(n \log \log u)$. To see why, consider what happens when an element is inserted in the tree. Since replacing the array of clusters by a hash table does not affect the number of nodes visited during an insertion, an insertion affects at most $O(\log \log u)$ nodes of the tree. Moreover, in each of these nodes at most one new entry is added to the hash table of clusters. Thus, after the insertion the total space of the data structure increases by at most $O(\log \log u)$ words. Clearly after inserting the n elements, the total space of the tree is bounded by $O(n \log \log u)$. While in the static setting queries can still be supported in $O(\log \log u)$ worst-case time, in the dynamic version queries and updates run in $O(\log \log u)$ expected time. Note that the space can be further improved to linear by using the same bucketing trick of y-fast tries; however, the running time of updates becomes $O(\log \log u)$ expected amortized instead of expected.

Z-fast tries: linear space, and fast updates (in expectation). In 2010, Belazzougui et al. [19] introduced the dynamic z-fast trie, a version of Willard's x-fast tries [102] that achieves linear space and $O(\log \log u)$ query and update time with high probability. The first version of this data structure was actually introduced one year earlier, by Belazzougui et al. [17], but it was static and could only find the longest common prefix between q and its predecessor. To improve upon the space and update times of the x-fast trie, Belazzougui et al. [19] made the following key changes (see Figure 4):

- In the z-fast trie the elements are stored in a compact trie, instead of a complete trie. The compact trie collapses unary paths, and as a result it has less than $2n$ nodes.
- Only one hash table is used, instead of one per level of the tree, for the binary searches. This, together with the compact trie, allows reducing the space to $O(n)$.

- The keys stored in the hash table are carefully chosen to allow the efficient location of the exit node using a variant of binary search called *fat binary search*. To illustrate the difference with traditional binary search, suppose that we search for a key x within the elements at positions in $[l, r]$ of a set S. In fat binary search, instead of comparing x to the element of S at position $\lfloor \frac{l+r}{2} \rfloor$, x is compared with the element $S[f]$ for the unique f in $[l, r]$ that is divisible by the largest possible power of 2.
- As in the x-fast trie, each internal node stores two pointers to support fast access to the nodes storing the minimum and maximum elements in the subtree. However, they do not point directly to these nodes, but to some other descendant in the path. Instead of accessing these elements in $O(1)$ time, they are reached in time $O(\log \log u)$. This approach (similar in essence to the canonical subtrees of van Emde Boas [99]), is key to allow faster updates.

The keys associated with each node in the hash table are chosen as follows. Let the label $l(\alpha)$ of a node α of the compact trie be the concatenation of the labels of the edges in the path from the root to α, and let $p(\alpha)$ be the parent of α (see Figure 4 again). The *2-fattest number* of a non-empty interval $[a, b]$ is the unique integer $f \in [a, b]$ such that f is divisible by 2^k, for some $k \geq 1$, and no number in $[a, b]$ is divisible by 2^{k+1}. The key associated with each node α is the prefix of length f of $l(\alpha)$, where f is the 2-fattest number in the interval $[|l(p(\alpha))| + 1, |l(\alpha)|]$. To understand why these keys allow efficiently searching for prefixes of a given query in the trie, note that when one is binary searching for a value i within an interval $[a, b]$, the first value within the interval that is visited by the binary search is precisely the 2-fattest number of $[a, b]$. A very similar idea for searching longest common prefixes in a trie was introduced independently by Ruzic [92], although there, the keys associated with a node are stored in different hash tables depending on their size instead of storing them all in the same hash, and the data structure is static.

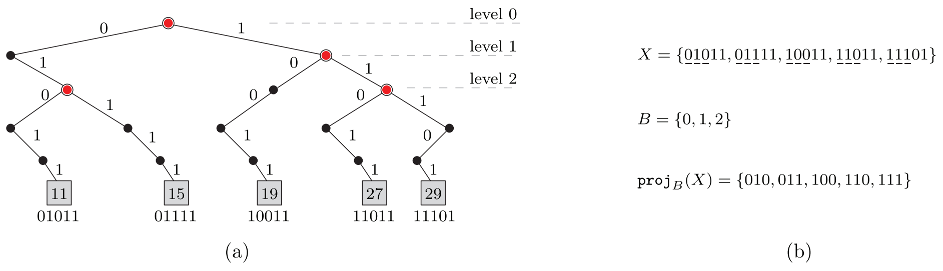
Belazzougui et al. [18] showed how to implement queries in $O(\log \log u)$ worst-case time, and updates in $O(\log \log u)$ expected time. As for other data structures, the only reason of having probabilities in the update bound is the use of hashing. Thus, improvements in dynamic hashing immediately translate into better time bounds for the z-fast trie.

All the solutions based on length reduction obtain times as a function of u, which drives the lengths of the keys. The times are independent of $|X|$, however. An orthogonal approach is to consider trees whose height depend on $|X|$, instead of on u. In the next section, we review the fusion tree, a data structure based on this approach.

3.2 Predecessor Search via Cardinality Reduction

Data structures implementing cardinality reduction are usually based on balanced search trees. The simplest of such data structures is a complete binary tree, which reduces the set of searched keys by one half at every level. This solution achieves predecessor search in $O(\log n)$ time independently of the universe size. Another basic idea is to use a B-Tree. Imagine that for any given set of b keys, one can implement predecessor queries in time $Q(b)$ using space $S(b)$. Then, using a B-Tree of degree b one could store any set of n keys ($n \gg b$) using space $O(S(b) \cdot n/b)$, and answer predecessor queries in time $O(Q(b) \cdot \log_b n)$. If one is able to store a set with $b = \omega(1)$ keys so that predecessor queries take constant time, then predecessor queries over a set of n keys can be answered in $o(\log n)$ time. In this section, we review data structures that implement this idea.

3.2.1 Fusion Trees. In 1993, Fredman and Willard [51] introduced the fusion trees. Basically, a fusion tree is B-Tree whose height depends on the size of the set X of keys, and whose degree depends on the word size w. They key component of this solution is the *fusion node*, a data structure that can support predecessor search among $\omega(1)$ keys in constant time using just $O(1)$ additional words. For this, Fredman and Willard designed an ingenious *sketching* technique that packs



$$X = \{\underline{010}11, \underline{011}11, \underline{100}11, \underline{110}11, \underline{111}01\}$$

$$B = \{0, 1, 2\}$$

$$\texttt{proj}_B(X) = \{010, 011, 100, 110, 111\}$$

Fig. 5. An illustration of the sketching in fusion trees for the set $X = \{11, 15, 19, 27, 29\}$. In (a), the branching nodes on the trie representing X have been highlighted; they occur only on the levels $B = \{0, 1, 2\}$. In (b), we illustrate the operation $\texttt{proj}_B$: the set X is represented at the top in binary, and the bits at positions in B have been underlined for each element of X. At the bottom, we show the set $\texttt{proj}_B(S)$ of skectches from X.

$b = \Theta(w^{1/5})$ keys into $O(1)$ words,[2] and they showed how to answer predecessor queries among the packed keys in constant time by means of word-level parallelism. Plugging the fusion node in the main idea described at the beginning of this section yields a b-ary search tree with query time $O(\log_w n)$. Next, we describe fusion nodes in detail, based on a simplified version of Fredman and Willard's work [51] presented by Patrascu in his PhD thesis [77].

Sketching. Let $S = \{x_1, \ldots, x_b\} \subseteq X$ be the values to sketch, and consider the binary trie representing these values as root-to-leaf paths. Note that there will be at most $b - 1$ branching nodes (i.e., nodes with more than one child) on these paths (see Figure 5(a)). Let B be the set of levels containing at least one of these branching nodes, and let $\texttt{proj}_B(x)$ be the result of projecting a value $v \in X$ on the bit positions in B. More precisely, $\texttt{proj}_B(v)$ is the integer of $|B|$ bits resulting from $\sum_{i=1}^{|B|} 2^i \cdot v[B[i]]$, where $B[i]$ denotes the ith element of B, and $v[j]$ the jth bit of v. The sketch of the set S is simply the set $\texttt{proj}_B(S) = \{\texttt{proj}_B(x_1), \ldots, \texttt{proj}_B(x_b)\}$ (see Figure 5(b) for an example). This takes $b\,|B| = O(b^2)$ bits, which fits in $O(1)$ words for $b = O(\sqrt{w})$.

Note that for any $y \in S$, if $x = pred(X, y)$ is the predecessor of y in X, the sketch $\texttt{proj}_B(x)$ is the predecessor of $\texttt{proj}_B(y)$ in $\texttt{proj}_B(S)$. For elements $y \notin S$ this might not be the case (in Figure 5(a) for instance, $\texttt{proj}_B(28) = \texttt{proj}_B(29) = 111$, thus 28 and 29 have the same predecessor in $\texttt{proj}_B(S)$, but not in S). This occurs because the exit node of y in the trie may be in a level that is not in B (because there are no branching nodes at that level), and the location of $\texttt{proj}_B(y)$ among the leaves of the trie for $\texttt{proj}_B(S)$ might be different to the location from y in the original trie. However, one can still find the predecessor of any query y using its neighbors among the sketches. Suppose that the sketch $\texttt{proj}_B(y)$ is between $\texttt{proj}_B(x_i)$ and $\texttt{proj}_B(x_{i+1})$, for some i. Let p be the longest common prefix of either y and x_i, or y and x_{i+1} (of those two, p is the longest); and let l_p denote the length of p. Note that p is necessarily the longest common prefix between y and not only x_i and x_{i+1}, but any element of S. Thus, in the trie for S, the node v in the l_pth level corresponding to p is precisely the exit node of y. Since only one of the children of v has keys from S, that child contains either the predecessor or the successor of y depending, respectively, on whether $p1$ or $p0$ is a prefix of y. If $p1$ is a prefix of y, then y's predecessor is the same as the predecessor of $e = p011\ldots1$, and if $p0$ is a prefix of y then y's successor is the same the successor of $e = p100\ldots0$. The predecessor (respectively, successor) of e can be safely determined by using only the sketches: all the bits of e and y at positions $b \in B$ such that $b \leq l_p$ are equal, and all the remaining bits in the suffix of e

[2]Originally Fredman and Willard [51] required b to be $O(w^{1/6})$; however $b = O(w^{1/5})$ is enough [79]. In terms of the overall performance of fusion trees the exact power is irrelevant, it only translates into a constant factor in the running time.

(specially those in positions of B) after the first l_p bits are the highest (respectively, the lowest) possible.

Implementation. To support predecessor queries on S one needs to perform several operations in constant time: first, the sketch corresponding to the query must be computed, then one must find its predecessor among the sketches in $\mathtt{proj}_B(S)$, and finally that predecessor must be translated into the real predecessor in S, by computing e as described above and finding its predecessor among the sketches. The key is an implementation of $\mathtt{proj}_B(S)$ that compresses $|B|$ scattered bits of any $x \in S$ into a space of $O(|B|^4)$ contiguous bits such that, when the b keys of S are compressed and concatenated into a word K, predecessor queries among the keys can be supported in constant time. Since K must fit in $O(1)$ words, one can sketch only $b = \Theta(w^{1/5})$ values, but this is still enough to obtain $O(\log_w n)$ query time. Fredman and Willard [51] showed how to compute K in $O(b^4)$ time. Solving predecessor queries on S involves some carefully chosen multiplication, masking, and *most significant bit* operations on K. They use word-level parallelism to compare the query with all the sketches at the same time, and give a constant-time implementation of the *most significant set bit* operation, which allows computing e in constant time. This approach relies heavily on constant-time multiplications. It is now known that multiplications are indeed required: Thorup [97] proved that constant-time multiplications are needed even to support just constant-time membership queries on sets of size $\Omega(\log n)$.

Updates. To analyze the running time of updates, note that whenever a fusion node is modified, the set of relevant bits might change, which would require to recompute all the $b = \Theta(w^{1/5})$ sketches. Thus, updating an internal node of the B-Tree (inserting or deleting a child, splitting the node, etc.) requires $O(b^4)$ time in the the worst case. This implies a total update time within $O(b^4 \cdot \log_b n)$ (the second term is the number of levels in the B-Tree). To reduce this to $O(\log_b n + \log b)$ amortized time, one can use a bucketing technique similar to y-fast tries: instead of storing all the elements in the B-Tree, its leaves point to balanced trees storing between $b^4/2$ and b^4 keys. Updates in the B-Tree are only done when the size of a balanced tree falls below $b^4/2$ after a deletion (triggering a merge), or exceeds b^4 after an insertion (triggering a split). Since an update within any of these balanced trees takes $O(\log b)$ time, and updates to the B-Tree are needed only every $O(b^4)$ update operations on X, the amortized running time of updates is within $O(\log_b n + \log b)$, for any $b \in O(w^{1/5})$.

3.2.2 Other Solutions for Small Sets. The fusion tree, and particularly the fusion node, was motivated by a work of Ajtai et al. [5], who introduced in 1984 a data structure for sets S of size $w/\log w$ with constant query and update time, in the cell-probe model. They showed how to implement all queries and updates using at most 4 cell probes. In this model, however, all the computations are free, which renders this solution impractical. Their main idea was to represent the keys in S in a compact trie T of w bits. The model allows them to define constant-time arithmetic operations to search, insert and delete a given key x in a compact trie T, as long as T fits in one word. Although unrealistic, the work of Ajtai et al. inspired other data structures for small sets, including the fusion nodes, the *q-nodes* (a key ingredient of the atomic heaps) introduced by Fredman and Willard [52], and the dynamic fusion nodes described by Patrascu and Thorup [83].

In 1994, Fredman and Willard [52] introduced the q-nodes, a variant of the fusion nodes that can store $(\log N)^{1/4}$ keys and perform predecessor queries and updates in constant worst-case time, provided that one has access to a common table of size $O(N)$. Combining the q-node with B-Trees yields a a data structure for the dynamic predecessor problem with search and update operations in $O(\log_w n)$ time, as long as $n \in \Theta(N)$, and $w \in \Theta(\log N)$. The main issue is that these guarantees hold only when the value of n is (approximately) known in advance, which is impossible in the fully dynamic version of the problem. However, such a data structure is useful when it is part of an algorithm for solving some other static problem. For instance, using q-nodes Fredman and Willard [52] introduced the *atomic heaps*, a data structure that allowed them to obtain the best algorithms at that time for the *minimum spanning tree* and the *shortest path* problems. In 2000, Willard [103] explored the impact of q-nodes on hashing, priority search trees, and various problems in computational geometry. The q-nodes are the key ingredient of the q^*-heap, a data-structure they introduced to obtain improved algorithms for the problems considered. The q^*-heap performs similar to the atomic heaps but running time bounds for the q^*-heap are in the worst-case, while the ones known for atomic heaps are amortized.

In 2014, Patrascu and Thorup [83] presented a simpler version of the fusion nodes, which improves their application under dynamic settings. Their data structure, the *dynamic fusion node*, stores up to $O(w^{1/4})$ keys while supporting predecessor queries and updates in constant worst-case time. Their solution combines the techniques of Ajtai et al. [5], and of Fredman and Willard [51]: they simulate the compact trie representation of Ajtai et al. [5] by introducing "don't care" characters in the sketches of Fredman and Willard [51]. By using the dynamic fusion node, one can obtain a simpler implementation of fusion trees: Since updates are now done in constant time within the fusion node, there is no need to use a different data structure at the bottom of the B-Tree (i.e., there is no need for bucketing) to obtain efficient updates. Besides, the update time now becomes $O(\log_b n)$ worst-case instead of amortized $O(\log_b n + \log b)$.

None of the data structures based only on length reductions (i.e., van Emde Boas trees and its variants) is faster than those based only on cardinality reduction (i.e., fusion tree-based solutions) for all configurations of n, u, and w, and the same holds in the other direction. A natural approach in the hope of finding optimal solutions is combining both techniques. We describe next some results based on such combination of techniques. We warn the reader that the descriptions become necessarily more technical from now on, but they mostly build on combining previous ideas.

3.3 Combining Length and Cardinality Reductions

A simple combination of the y-fast trie [102] and the fusion tree [51] improves the running time of the operations to $O(\sqrt{\log n})$, which is better than each data structure by itself in the worst case. Since fusion nodes actually allow implementing B-Trees with any branching factor in $O((\log u)^{1/5})$, the time bounds of the fusion tree can be improved to $O(\sqrt{\log n})$ for $n \le (\log u)^{(\log\log u)/25}$, while retaining $O(n)$ space: simply use a branching factor of $\Theta(2^{\sqrt{\log n}})$ in the B-Tree,[3] and store $2^{\Theta(\sqrt{\log n})}$ elements in each of the binary search trees at the leaves. For the case $n > (\log u)^{(\log\log u)/25}$, Willard's y-fast tries [102] have query time and expected update time within $O(\log\log u) \subseteq O(\sqrt{\log n})$.[4] Better results can be obtained with more sophisticated combinations of cardinality and length reduction. We review in this section three fundamental ones: a data structure presented by Andersson [8] achieving sublogarithmic query times (as fusion trees) without multiplications, the exponential search trees, by Andersson [9], and a data structure introduced by Beame and Fich [14].

3.3.1 *Sublogarithmic Searching without Multiplications.*

Fusion trees make extensive use of constant-time multiplications, however sublogarithmic search times can be achieved without this operation, as shown by Andersson [8]. Combining ideas from the y-fast tries and the fusion trees,

[3] $n \le (\log u)^{(\log\log u)/25} \Rightarrow \log n \le (\log\log u)^2/25 \Rightarrow 2^{\sqrt{\log n}} \le (\log u)^{1/5}$.
[4] $n > (\log u)^{(\log\log u)/25} \Rightarrow \log n > (\log\log u)^2/25 \Rightarrow \sqrt{\log n} > (\log\log u)/5$.

Andersson [8] presented a data structure that uses only AC^0-RAM operations, supports predecessor queries in $O(\sqrt{\log n})$ time, with expected $O(\sqrt{\log n})$ update time, and uses linear space.

The idea is to reduce the problem of supporting predecessor queries among long keys, via length reduction, into that of maintaining short keys that can be packed into a small number of words, and be queried and updated efficiently. The data structure is basically a tree in which the top levels correspond to a y-fast trie, and each leaf of this y-fast trie points to a packed B-Tree (similar to the fusion tree). As in the fusion tree, only $\Theta(n/2^{\sqrt{\log n}})$ elements are stored in the main data structure; the rest are in balanced search trees of height $\Theta(\sqrt{\log n})$. The structure stores $\sqrt{\log n}$ levels of the y-fast trie, which halves the length of the keys at each level, for a total reduction factor of $2^{\sqrt{\log n}}$. Because of this reduction, at this point at least $2^{\sqrt{\log n}}$ keys fit in one word. Hence, each leaf of the y-fast trie points to a packed B-Tree with branching factor $2^{\sqrt{\log n}}$ and height $O(\sqrt{\log n})$. The searches among the keys of each B-Tree node are performed in constant time via a lookup table.

Brodal [33] constructs a data structure that is similar to Andersson's [8], which also avoids multiplications and achieves sublogarithmic search times. It uses buffers to delay updates to the packed B-Tree. In the worst case, it uses $O(f(n))$ time to perform insertions and deletions, and $O((\log n)/f(n))$ time for predecessor queries, for any function f such that $\log\log n \le f(n) \le \sqrt{\log n}$. Yet, it uses $O(nu^{\varepsilon})$ space, for some constant $\varepsilon > 0$.

3.3.2 Exponential Search Trees. The exponential search trees were introduced by Andersson [9] in 1996. They give a general method for transforming any data structure D_P for the static predecessor problem supporting queries in time $Q(n)$, into a linear-space dynamic data structure with query and amortized update time $T(n)$, where $T(n) \le O(Q(n)) + T(n^{k/(k+1)})$. The only two conditions that D_P must meet for this are that it can be constructed in $O(n^k)$ time, and that it uses $O(n^k)$ space, for some constant $k \ge 1$. Combining their technique with the fusion tree and the y-fast trie, exponential search trees yield a data structure for the dynamic predecessor problem with worst-case query time and amortized update time of the order of

$$\min \begin{cases} \sqrt{\log n} \\ \log\log u \cdot \log\log n \, . \\ \log_w n + \log\log n \end{cases} \tag{2}$$

An exponential search tree is a multiway search tree in which the keys are stored at the leaves, the root has degree $\Theta(n^{1/(k+1)})$, and the degree of the other nodes decrease geometrically with the depth. Besides the children, each internal node stores a set of *splitters* for navigation (as in B-Trees): when searching for a key at a node, one can determine which child the key belongs to by using local search among the splitters. More precisely, let $b = n^{1/(k+1)}$. At the root of the tree, the n keys from X are partitioned into b blocks, each of size $n/b = n^{k/(k+1)}$. Like in B-Trees, the set of splitters of the node consist of the minimum element of the blocks $2, \ldots, b$, and this set is stored in an instance of the data structure D_P. An exponential search tree is then built recursively for each of the b blocks, which become the children of the root. The main difference with B-Trees is that the degree of the nodes changes with the depth: the nodes at depth i have a degree of $n^{(k/(k+1))^i}$. Thus, after $\log_{(k+1)/k} \log n \in O(\log\log n)$ levels, the nodes store a constant number of keys.

To answer a predecessor search, the $O(\log\log n)$ levels of the tree are traversed in a root-to-leaf path. At each node in the path, the data structure D_P is queried to determine the child that contains the answer. It follows that searches in the exponential search tree are supported in time $T(n) = O(Q(n^{1/(k+1)})) + T(n^{k/(k+1)})$.

Unfortunately, updating this data structure requires rebuilding it partially or even globally, which only allows for amortized update times. Note that for large enough word sizes, the last branch in the bound of Equation (2) is better than the update time of fusion trees in the dynamic case, which could only achieve amortized time $O(\log_b n + \log b)$, for any $b \in O(w^{1/5})$. Andersson and Thorup [12] de-amortized the bounds for updates of the exponential search trees by using eager partial rebuilding and showing how to insert or delete an element in constant worst case time, once the element or its predecessor has been found in the tree.

3.3.3 Beame and Fich Solution for Polynomial Space. Beame and Fich [14] introduced a variant of the x-fast tries that, if $\log\log u < \sqrt{\log n \log\log n}$, yields a solution with query time in $O(\frac{\log\log u}{\log\log\log u})$, using $O(n^2 \log n / \log\log n)$ space. Combining this with a fusion tree if $\log\log u \geq \sqrt{\log n \log\log n}$ improves the time of static predecessor queries to $O(\min\{\frac{\log\log u}{\log\log\log u}, \sqrt{\frac{\log n}{\log\log n}}\})$. This result shows that, if one is willing to spend $n^{O(1)}$ space, then the query time of the van Emde Boas tree can be improved by a factor of $\log\log\log u$. For some time, it was widely conjectured [55] that this was impossible.

Inspired in the parallel comparison technique introduced by Fredman and Willard [51] to obtain constant-time queries in fusion nodes, Beame and Fich [14] introduce the idea of *parallel hashing*, key for their solution. They show that one can take advantage of a large word size w to answer membership queries in several dictionaries at once, in constant time. More precisely, they prove that given k sets of elements from a universe of size 2^u, if $w \in \Omega(uk^2)$, then k independent parallel membership queries, one per set, can be supported in constant time. Their data structure uses $O(uk2^{(r+1)k})$ bits, where 2^r is an upper bound to the size of the sets.

The relevance of parallel hashing is that it allows replacing the binary searches performed on the levels of the x-fast trie (when answering a query) by a parallel search over a multiway tree. This can be interpreted as examining several levels of the x-fast trie at once. Parallel searches allow one to implement a recursive data structure in which, after each such search, either the length of the relevant portion of the keys, or the number of keys under consideration, are reduced significantly: the number n of keys and their length l become n' and l', respectively, and either $l' = l$ but $n' \leq n^{1-1/v}$, or $l' = l/v$ for some v such that $n \geq v^v \geq \log u$ (for values of n such that there is no possible v meeting this condition they use fusion trees).

Beame and Fich [14] described their data structure only for the static predecessor problem. However, combining their solution with the exponential search tree [12] yields a dynamic data structure that uses linear space, and with worst-case query time and amortized update time within $O(\min\{\frac{\log\log u}{\log\log\log u}\log\log n, \sqrt{\frac{\log n}{\log\log n}}\})$ (i.e., paying an extra $\log\log n$ factor in the time of queries to support updates).

Finally, Beame and Fich [14] showed that their solution for the static predecessor problem is optimal in the following sense: there are values of n and w such that one cannot obtain a data structure with space polynomial in n that answers predecessor queries in time $o(\frac{\log\log u}{\log\log\log u})$, and there are values of $\log u$ and w such that, using polynomial space, predecessor queries cannot be answered in time $o(\sqrt{\frac{\log n}{\log\log n}})$. The existence of a data structure that is optimal with respect to the entire spectrum of possibilities of word size, universe size, set size, and space usage, remained open until the remarkable work of Patrascu and Thorup [81, 82], which we review next.

3.4 The Optimal Upper Bounds

Patrascu and Thorup [81, 82] provided tight tradeoffs between query time and space usage for the static predecessor problem. Their data structure is an advanced combination of a variety of the techniques and data structures we have reviewed. These results were originally introduced in [81], but one year later [82] they showed that their lower bound holds also under randomized settings, proving that their data structure is optimal even when randomization is allowed. In 2014, they extended their results to the dynamic version of the problem [83].

3.4.1 Static Predecessor. Patrascu and Thorup [81] showed that in a RAM with word size w, given a set of n integers of l bits each (i.e., $u = 2^l$), there is a data structure using $S = O(2^a n)$ bits of space, for any $a \geq \log l$, that answers predecessor queries in the order of the times given in Equation (1).

To illustrate how the branches in this upper bound cover the whole spectrum of possibilities, consider the case where $a = \Theta(\lg l)$ (i.e., linear space data structures) and $l = w$:

- For n such that $\log n \in [1, \frac{\log^2 w}{\log\log w}]$ the minimum occurs in the first branch, which increases from $\Theta(1)$ to $\Theta(\frac{\log w}{\log\log w})$;
- For n such that $\log n \in [\frac{\log^2 w}{\log\log w}, \sqrt{w}]$ the minimum occurs in the third branch, increasing from $\Theta(\frac{\log w}{\log\log w})$ to $\Theta(\log w)$;
- For n such that $\log n \in [\sqrt{w}, w]$ the minimum occurs in the second branch, decreasing with n from $\Theta(\log w)$ back to $\Theta(1)$.

Note that in this example the fourth branch never yields the minimum query time. This is because this branch is relevant when the universe is super-polynomial in n (i.e., $l = \omega(\log n)$), and the space is sub-linear in n (i.e., $a = o(\log n)$). Consider, for instance, a case in which $a = \sqrt{\log n}$, and $w = l = \log^c n$, for some constant $c > 2$. Under these settings, the first branch yields a bound of $\frac{\log n}{c \log\log n}$. This is worse than at least the second branch, which is asymptotically within $O(\log\log n)$. More precisely, the second branch yields a value of $\log \frac{l}{a}$ (up to an additive constant factor), which is the same as the numerator in the third and fourth branch. However, while under these settings the denominator of the third branch becomes $o(1)$, the denominator of the fourth one becomes c. Thus, this branch is the optimal choice for large enough c.

The upper bound of Equation (1) is achieved by a data structure on RAM whose query algorithm is deterministic, and thus the bound holds for its worst-case complexity. This data structure results from a clever combination and improvement of different results preceding the work of Patrascu and Thorup [81].

Fusion trees, external memory, and the first branch. The first branch is the only one depending on w in the word-RAM model and on B in the external memory model.

In a word-RAM machine, this bound is achieved using fusion trees [51]. Moreover, fusion trees allow increasing the available space per key for the data structures corresponding to the other three branches of Equation (1). Given that the total space available is $O(2^a n)$ bits, the bits available per key is on average $O(2^a)$. However, using a simple bucketing trick, the w bits available per key for the other three branches can be increased up to $O(2^a w)$. To do this, divide the n keys into n/w buckets of size w, and create a set X' of size n/w by choosing one representative element from each bucket (e.g., the minimum). The data structures corresponding to the other branches will be initialized over X' instead of the original X. This increases the available bits per key for those data structures up to $O(2^a w)$. To find the predecessor within each bucket, a fusion tree is initialized for each of the n/w buckets, using $O(n)$ space in total. Thus, once the bucket in which the predecessor of a query q has been found, the precise predecessor within the bucket is found using the respective fusion tree in constant time.

In external memory, the bound of the first branch, and the gain in space per key, can be achieved by considering $B = w$ and replacing fusion trees with the simpler B-Trees [43, Chapter 18].

Van Emde Boas trees, and the second branch. The second branch is relevant for polynomial universes (i.e., $l = O(\log n)$). The bound of this branch is achieved by van Emde Boas trees [101] modulo some simple improvements. As described in Section 3.1.1, this data structure reduces the key length from l to $l/2$ at each recursive step in constant time. This yields an upper bound of $O(\log l)$, which can be improved using two simple ideas:

- Stop the recursion when $l \leq a$, instead of when l is constant. This new base case can be solved in constant time using lookups on a shared table of $2^a \cdot l$ bits. This improves the query time to $O(\log \frac{l}{a})$.
- Partition the universe into n slots based on the first $\log n$ bits of the key, and store each slot in a van Emde Boas tree with keys of $w - \log n$ bits. Using a table of $2^{\log n} \log \leq nl$ bits one can determine in constant time in which of the n slots to look for the predecessor of any query q. Combining the first idea with this one yields the complexity of $O(\log \frac{w-\log n}{a})$.

Beame and Fich's data structure, and the last branches. The third and fourth branches are relevant for when the universe is super-polynomial with respect to n (i.e., $l = \omega(\log n)$): the third one is asymptotically better when $a = \omega(\log n)$ (i.e., for super polynomial space, like in the data structure of Beame and Fich [14]), while the last branch is asymptotically better when $a = o(\log n)$ (i.e., for small-space data structures). The upper bound of the third branch is obtained by a careful combination of cardinality and length reductions, inspired by the solution of Beame and Fich [14]. As seen, this structure can improve upon van Emde Boas', but it needs a lot of space. Interestingly, the same techniques can be useful for small-space data structures. For the last branch they use the same combination of length and cardinality reduction, but with a special selection of how cardinality is reduced tailored for the case of small space.

3.4.2 Dynamic Predecessor. In the dynamic setting, Patrascu and Thorup [83] showed that if randomization is allowed, then there is a data structure achieving (in expectation) the optimal bounds. The optimal expected operation time (maximum between queries and updates) for dynamic predecessor is asymptotically

$$1 + \min \begin{cases} \log_w n \\ \log \frac{\log (2^l - n)}{\log w} \\ \dfrac{\log \frac{l}{\log w}}{\log\left(\log \frac{l}{\log w} / \log \frac{\log n}{\log w}\right)} \end{cases}. \qquad (3)$$

The first obvious difference with the static bound of Equation (1) is that there is no direct reference to space used by the data structure (i.e., a does not appear in this bound). The data structure achieving this bound uses linear space, and no asymptotic improvements can be obtained by using more space. Intuitively, the larger the space the harder it is to maintain it updated. The first branch is achieved by a dynamic fusion tree [83] implemented using the dynamic fusion node described in Section 3.2.1. For the third branch they give a dynamic version of the data structure for the fourth branch of the optimal static upper bound, based on Beame and Fich's combination of length and cardinality reductions [14]. In terms of just the bound, the third branch it is the same as the fourth branch of the static bound, but considering $a = \log w$. Since the first and third branches of the dynamic bound are the same as the first and fourth branches of the static bound, respectively, they are trivially optimal: any lower bound for the static problem applies to the dynamic version as well. The data structure for the second branch is a dynamic variant of the van Emde Boas tree similar to that for the second branch of the optimal static bound. The main difference is that the partition of the universe into n slots needs to be maintained in a dynamic data structure instead of in a table, which can be achieved for instance by using bit vectors [41, 75]. Moreover, for the base case of the recursion, instead of using complete tabulation, a dynamic fusion node is used when $l \leq w$. The upper bound of this variant degrades with respect to the static one: the term $l - \log n$ of the static bound is replaced by $\log(2^l - n)$. However, Patrascu and Thorup were able to prove a matching lower bound, showing that the upper bound of this branch is also optimal.

4 LOWER BOUNDS

The first super-constant lower bound for the predecessor problem was proven by Ajtai [4]. He showed that, for word size within $O(\log n)$, there is no data structure in the cell-probe model supporting predecessor queries in constant time while using space polynomial in n. Several improvements to this bound followed [14, 71, 72, 93], until in 2006 Patrascu and Thorup [81, 82] proved an optimal lower bound for the static problem, even when allowing randomized query schemes. We review some of these results in this section.

4.1 Communication Complexity Lower Bounds

Miltersen [71] generalized Ajtai's proof [4], and obtained a lower bound of $\Omega(\sqrt{\log\log u})$ for the static predecessor problem when $w \leq 2^{(\log n)^{1-\varepsilon}}$ for any $\varepsilon > 0$. To prove this bound, he introduced a general technique for translating time-complexity lower bounds for static data structures into lower bounds for dynamic data structures, and showed that if the time of updates is in $O(2^{(\log u)^{1-\varepsilon}})$, then predecessor queries take $\Omega(\sqrt{\log\log u})$ time, for any constant $\varepsilon > 0$. Apart from the lower bound, Miltersen [71] introduced two key ideas: the lower bound arguments were based on the communication complexity of the problem; and the bounds held even for a simpler version of the problem in which each element of X is associated with one of two different colors (e.g., red or blue), and the goal is to determine the color of the predecessor of a given query.

Miltersen [71] observed that a static data structure problem in the cell-probe model can be interpreted as a communication game between two players, Alice and Bob, in which Alice (the query algorithm) holds the query, Bob (the data structure) holds the table of S cells storing the data structure, and they must communicate to find the answer to the query. The communication between Alice and Bob is structured in strictly alternating rounds: Alice first requests the content of a cell by sending a block of $\log S$ bits with the cell name, and then Bob sends a message with w bits containing the content of that cell. The complexity of this communication protocol is given by the number of rounds of communication T that occur between Alice and Bob to find the answer to the query. A lower bound on T yields a lower bound for the algorithm represented by Alice in the cell-probe model.

Using this technique, Miltersen et al. [72] extended the lower bounds of Ajtai [4] and Miltersen [71] to randomized settings, and showed that for certain universe sizes they also yield an $\Omega((\log n)^{1/3})$ lower bound on query time. More importantly, to obtain their proofs they introduced a *round elimination lemma*, which became a general tool to prove lower bounds for data structures based on communication complexity [14, 81, 93], and which inspired the optimal lower bounds of Patrascu and Thorup [81, 82].

Round elimination. Intuitively, to prove a lower bound for some problem using round elimination, suppose that one has a communication protocol with T rounds for the problem. The idea is to eliminate all rounds of communication and reach a state that implies a contradiction. For this, one shows that the initial message of the protocol contains a small amount of information about the sender's input, under some probability distribution on the inputs of Alice and Bob [93]. Thus, eliminating the first message yields a protocol with $T-1$ rounds of communication where the other player starts, and with only slightly higher average error probability. Repeating this elimination T times yields a protocol with zero rounds of communication, and thus, the average error probability of this protocol must be at least $1/2$. Hence, from the total increase in error probability after the T rounds one can obtain lower bounds on T. Miltersen's round elimination lemma [71] provides bounds on the increase in error probability after eliminating one round of communication.

Building on the work of Miltersen [71], Miltersen et al. [72], and Beame and Fich [14] showed that their solution to the predecessor problem was optimal in the following sense:

- For all sufficiently large universe U, there is a constant c and a value of n such that any deterministic cell-probe data structure that uses $n^{O(1)}$ cells, each of word size $2^{(\log u)^{1-\Omega(1)}}$, must probe at least $\frac{c \log\log u}{\log\log\log u}$ cells to find the predecessor of a given query; and
- For all sufficiently large n, there is a constant c and a value of u such that any deterministic cell-probe data structure for predecessor that uses $n^{O(1)}$ cells, each of word size $(\log u)^{O(1)}$, must probe at least $\sqrt{\frac{c \log n}{\log\log n}}$ cells while processing a query.

Later, Sen and Venkatesh [93] proved a stronger version of Miltersen et al.'s round elimination lemma [72], and showed that Beame and Fich's lower bounds [14] hold not only for deterministic data structures but also for randomized query schemes. The main deficiency of the lower bounds introduced up to the work of Sen and Venkatesh [93] was that they yield only lower bounds in the form of functions depending either on the universe size or on the size of the input set, but not on both values together [93].

Patrascu and Thorup [81] pointed out two other drawbacks of lower bounds based on communication complexity. On the one hand, the relation between cell-probe data structures and communication complexity was not expected to be tight. Intuitively, in a communication protocol, Bob can remember the messages of past rounds of communication, and answer requests from Alice with messages based on it. However, in the communication game described by Miltersen [71] Bob is just a table of cells storing the data structure, and cannot remember anything. Thus, Bob's responses must depend only on Alice's last message. On the other hand, lower bounds for data structures based on the communication complexity of a protocol cannot be used to distinguish data structures that use polynomial space (like in the one of Beame and Fich [14]) from those using only linear space. To see why, consider two data structures using space S and S', respectively, and suppose that $S' = S^c$ for some constant $c > 1$. Note that the differences in space between these two data structures only change the size of Alice's messages by a constant factor of c, because $\log S' = c \log S$. Thus, the difference in space can increase the number of rounds in the communication protocol only by constant factors: Alice can break her messages of $\log S'$ bits into c separate messages of $\log S$ bits.

To overcome these deficiencies, Patrascu and Thorup [81] developed a new lower-bound technique specifically tuned for the cell-probe model: the *cell-probe elimination lemma.*

4.2 Pure Cell-probe Model Lower Bounds

Patrascu and Thorup [81, 82] obtained a lower bound for data structures supporting predecessor queries in the form of a function that depends simultaneously on the size of the universe and the input set, the word-length, and the number of bits used by data structure (see Equation (1)). Since they also described a data structure in RAM whose upper bound is asymptotically the same function, their lower and upper bounds are optimal both in the cell-probe and RAM models.

4.2.1 Static Lower Bounds. Patrascu and Thorup showed that given a set of n integers of l bits each, the predecessor search time of any data structure using $2^a n$ bits of space, for any $a \geq \log l$, is lower bounded by the formula in Equation (1) multiplied by some constant. This is the same formula for their upper bounds, and hence the optimality of Patrascu and Thorup's data structure [81, 82] for the static predecessor problem.

In terms of the lower bound, their main result was to prove the tight bound for $a = \log l + o(\log n)$, in particular, the second and fourth branches of the tradeoff. As described in Section 3.4, the upper bound of branch two is achieved via a slight variation of van Emde Boas' data structure [101]; while the upper bound of branch four is achieved using a tuned version of the data structure of Beame and Fich [14]. Beame and Fich [14] improved the running time of van Emde Boas' data structure [101] by using space polynomial in n. Branches two and four show that such an improvement required indeed polynomial space. These branches also imply that for near-linear space (i.e., $a = \log l + o(\log n)$) and polynomial universes (i.e., $w = l = \gamma \log n$, constant $\gamma > 1$) van Emde Boas' original data structure [101] was optimal. As mentioned, lower bound techniques based on the communication model were useless, since one could not even distinguish $a = \log l + 2$ from $a = \log l + \log n$. The third branch yields improved bounds for the case where $a = \log l + \Omega(\log n)$ and $a \leq \log l + w^{1-\varepsilon}$. The best previous lower bound was $\Omega(\min\{\frac{\log n}{\log w}, \frac{\log w}{\log a}\})$. The third branch implies a bound of $\Omega(\min\{\frac{\log n}{\log w}, \frac{\log w}{\log\log w + (\log a/\log n)}\})$.

To prove the third branch, Patrascu and Thorup [81] combined the round-elimination lemma of Sen and Venkatesh [93] with the *message compression* technique introduced by Chakrabarti and Regev [34]. For the bounds of branches two and four, they introduced the *cell-probe elimination lemma*. In terms of techniques, this is their most important contribution.

Cell-probe elimination. Proofs based on cell-probe elimination are in essence similar to those based on round-elimination: the goal is to iteratively eliminate all the cell probes, and reach a state that implies a contradiction. A key new idea of Patrascu and Thorup [81] was to augment the cell-probe model with the concept of *published bits*. Apart from the traditional memory of the cell-probe model (which must be accessed through cell-probes that impact the cost of algorithms), the published bits are special memory of bounded size that algorithms can access for free. The published bits are initialized with the data structure at construction time, and the size of this special memory is a function of the input. Observe that in this version of the model, if the input of a problem has n bits, and these n bits have been published, then the problem can be solved trivially with no cell probes to the regular memory. To eliminate cell probes, a small number of cells accessed frequently (i.e., by at least a constant fraction of the probes) are published. Obviously, as the number of bits that have been published increases, the cell-probe complexity of the algorithm decreases. If after T cell-probe eliminations one arrives at a complexity of zero cell probes, and less than n have been published bits, then one has a contradiction: there is part of the input that is unknown to the algorithm, and thus the query cannot be answered. Hence, from the total increase in published bits one can obtain lower bounds on T. Patrascu and Thorup's cell-probe elimination lemma [81] provides bounds on the increase in published bits required to eliminate one cell probe.

Another key idea to beat communication complexity lower bounds was to use a *direct sum* approach: Patrascu and Thorup [81] showed that an optimal data structure representing k independent instances of the predecessor problem (with the same universe and input set sizes) in space $k \cdot s$ cannot support queries over an instance in time better than an optimal data structure representing only one instance in space s. Intuitively this means that, in the worst case, sharing space between several instances of the predecessor problem does not yield a data structure with improved query time (when compared to using a separate data structure for each instance).

To illustrate the strength of this direct sum approach, consider the case of near-linear universes (i.e., $l = \log n + \delta$, with $\delta = o(\log n)$). The direct sum approach allows to transfer lower bounds for larger universes to near-linear universes. For instance, if one knows that for polynomial universes the optimal time is $\Omega(\log \frac{\delta}{a})$, one can prove the same bound for near-linear universe as follows: Consider $n/2^{\delta}$ independent instances of the predecessor problem, where the input of each instance is a set with 2^{δ} integers of 2δ bits each. A predecessor data structure for n l-bit integers can be used to represent these independent instances: Add a prefix to each integer with the number of the instance to which it corresponds (which takes $\log(n/2^{\delta}) = \log n - \delta$ bits), and store all the integers combined in the data structure. To answer queries within an instance, prefix the query with the number of the instance, and query the data structure. So, according to the direct sum approach, the complexity of this representation cannot be asymptotically better than the complexity of an optimal data structure for one of the instances. Since the size u' of the universe of each instance ($u' = 2^{2\delta}$) is polynomial in the size n' of the input set ($n' = 2^{\delta} = \sqrt{u'}$), the bound of $\Omega(\log \frac{\delta}{a})$ for polynomial universes holds for each instance, and thus it holds also for the near-linear universe.

Finally, another key idea to prove the bounds was to allow the query algorithm to reject queries in the following way: when the algorithm receives a query, it first inspects the published bits and decides whether it can or cannot answer the query. Only when the algorithm decides to answer the query (and after taking this decision) it can make cell probes, and in this case it must end with a correct answer. This model was crucial for the deterministic lower bounds, but it posed a challenge for randomized settings: in the randomized error case it could be possible that the algorithm only accepts queries leading to errors. This is why the static bounds were introduced first in 2006 [81], and extended to randomized settings one year later [82].

4.2.2 Dynamic Lower Bounds. The static lower bound holds trivially in the dynamic case as well. One would expect, however, the dynamic version to be harder. Patrascu and Thorup [83] showed that with key length $l \le w$, the optimal expected operation time (i.e., the maximum time between queries and updates) for dynamic predecessor is that of Equation (3), up to a constant factor.

Note that there is a close resemblance between the bound in Equation (3) for the dynamic predecessor problem, and the bound in Equation (1) for the static problem. An obvious difference is that the dynamic bound does not include any restriction on space usage. This is because dynamic lower bounds hold regardless of the available space, as long as updates are reasonably fast. For instance, if insertion of an element into a dynamic data structure $\mathrm{D}_{\mathrm{dyn}}$ takes amortized time t_u per element, then one can obtain a static data structure $\mathrm{D}_{\mathrm{stat}}$, which uses $O(n \cdot t_u)$ space, by simulating n insertions in $\mathrm{D}_{\mathrm{dyn}}$ and storing the cells that were modified ($O(n \cdot t_u)$) during these insertions in a hash table. Any processing done before inserting the integers is considered a universal constant, which is not counted in the cell-probe model, and Reference [83]. Now, the cost of a predecessor query on $\mathrm{D}_{\mathrm{dyn}}$ can be matched by $\mathrm{D}_{\mathrm{stat}}$, thus lower bounds for $\mathrm{D}_{\mathrm{stat}}$ also apply to $\mathrm{D}_{\mathrm{dyn}}$.

Because of the discussion above, the first and third branches of Equation (3) correspond to the near-linear space versions of the first and fourth branches of the static bound in Equation (1), respectively. As mentioned in Section 3.4, the third branch of Equation (1) is relevant when the universe is super-polynomial (i.e., $l = \omega(\log n)$), and the space is within $n \cdot 2^{\Omega(\log n)}$. Thus, this branch is not relevant for dynamic bounds, and in consequence it does not appear in Equation (3).

Finally, the second branch of Equation (3) is similar to the second branch of Equation (1), but in this case the term $l - \log n$ is improved up to $2^l - n$. For near-linear space, the second branch of the static bound of Equation (1) is relevant when $\log n \ge l/2$. To obtain the improvement for the dynamic bound, Patrascu and Thorup [83] proceeded as follows. Let S' be the set of $n' = \sqrt{2^l - n} < n$ elements of length $l' = \lceil \log(2^l - n) \rceil$ from the universe $[1, 2^{l'}]$. In this case, again it holds that

$\log n' \geq l'/2$, and the static lower bound states that queries on S' require time $\Omega(\log \frac{l' - \log n'}{\log w}) = \Omega(\log \frac{\log (2^l - n)}{\log w})$. As before, a dynamic data structure $\mathrm{D}_{\mathrm{dyn}}$ for sets of n keys of length l with update time t_u can be translated into a static near-linear space data structure $\mathrm{D}_{\mathrm{stat}}$ for S', and hence the static lower bound of $\Omega(\log \frac{\log (2^l - n)}{\log w})$ for $\mathrm{D}_{\mathrm{stat}}$ applies also for $\mathrm{D}_{\mathrm{dyn}}$.

Note that the dynamic data structure of Patrascu and Thorup [83] (described in Section 3.4.2) requires randomization in both queries and updates to match the running time lower bound described here. However, this is not the case for the static problem, where the query times are deterministic and worst-case, and the lower bound holds even under randomized settings. Thus, it is open which is the optimal deterministic running time in the dynamic case. We discuss this and other open questions at the end of the survey.

5 VARIANTS AND SPECIAL CASES OF THE PREDECESSOR PROBLEM

Various special cases and variants of the predecessor problem have also been considered [16, 20, 31, 32, 42], especially since the optimal results of Patrascu and Thorup [81] settled the static problem. Since the lower bounds (both static and dynamic) for the general problem will not directly apply to these special cases, many issues about them remain open. We review below some of the fundamental variants considered in the literature.

5.1 Distance-sensitive Predecessor Searches

Bose et al. [31] introduced the *Distance-sensitive Predecessor* problem, to unify data structures for the predecessor and the *Membership* problems.

Consider a set X of elements from a universe U, and let $p = pred(X, q)$ denote the predecessor of q in X, for $q \in U$. Bose et al. [31] showed that the predecessor p of q can be computed in time $O(\log \log \Delta)$, where $\Delta = q - p$. Note that if the query belongs to X, then $\Delta = 0$ and thus $O(\log \log \Delta) = O(1)$, which is the running time achievable by hash tables for membership queries. Furthermore, since $\Delta < u$, the bound is never worse than $O(\log \log u)$. Their data structure is essentially an x-fast trie, and their main result is actually a new algorithm to find predecessors within the trie. The key idea is to replace the binary search on the levels of the x-fast trie by a doubly exponential search (i.e., searching on the levels 2^{2^i}, for $i \in [1, \log \log \log u]$) starting from the leaves. In this way, if the query is present in X, the search takes constant time.

More precisely, the search algorithm described by Bose et al. [31] works as follows. Starting from the leaves (i.e., the level at depth $\log u$), search in a hash table whether the prefix q' of the query q corresponding to the current level is present. If q' is found, then the algorithm proceeds with the usual binary search, starting from the x-fast trie node corresponding to q'. This will take $O(\log h_{q'})$ time, where $h_{q'}$ is the height of the trie node. If q' is not found, then the algorithm queries whether $q' - 1$ is present at the same level. If this node is found, then the predecessor of q can be found in constant time using the pointer from the node for $q' - 1$ to the largest leaf descending from it. If $q' - 1$ is not present, then the doubly exponential search for prefixes of q in the tree continues. Checking for the presence of $q' - 1$ guarantees that, if the search continues to a higher level of the trie, it is because the answer is far away. Thus, the first time a prefix q' is found, it holds that $h_{q'} \in O(\log \Delta)$, and thus the running time of the algorithm is bounded by $O(\log \log \Delta)$.

To achieve updates in the same $O(\log \log \Delta)$ time (expected amortized), Bose et al. [31] described a solution based on y-fast tries that combines bucketing with skip lists [88]. Their data structure uses $O(n \log \log \log u)$ expected words of space. Belazzougui et al. [20] showed that queries can be supported in time $O(\log \log \Delta)$ using only $O(n)$ space, by implementing a search similar to that of

Bose et al. [31], but using a z-fast trie instead of an x-fast trie. In addition, their approach also supports queries in $O(\log\log \frac{u}{s-p})$ time, where s and p are the successor and predecessor of the query, respectively.[5] The running time of updates in their solution was not distance-sensitive, however. Ehrhardt and Mulzer [45] remedied this by presenting another variant of the z-fast trie, which, using $O(n)$ space, supports queries in $O(\log\log\Delta)$ time, and updates in $O(\log\log\Delta)$ expected worst-case time.

5.2 Biased Predecessor Search

Bose et al. [32] considered the problem of performing predecessor queries in time that depends on the distribution of the queried elements (namely, the *Biased Predecessor* problem). In this case, each element i of the universe has some probability p_i of being queried, and the goal is to support predecessor queries in time depending on their inverse probability. In the comparison model, for example, *biased search trees* [26] support predecessor and membership queries in $O(\log 1/p_i)$ time. The expected query time of the biased search tree is linear in the entropy H of the distribution of the queries ($H = \sum_{i=0}^{u-1} p_i \log 1/p_i$), and this is optimal in the comparison model. However, one would expect to perform faster in the RAM model, given that this is possible in for the classical version of the problem.

Recently, Ferragina and Vinciguerra [104] introduced a static data structure for this variant based on a geometric approach. The idea is to transform the input into the set of points $(x, \mathtt{rank}(x))$, and approximate this set by m segments so that the approximation error is at most a given ε. Their data structure, known as a PGM-index, uses $O(m)$ space and answers queries in OH average time.

Bose et al. [32] presented various data structures for the static biased predecessor problem, with different space and query time tradeoffs. Given a probability distribution over the possible queries in a universe U with entropy H they show, for instance, that there is a data structure that supports predecessor queries in $O(\log(H/\varepsilon))$ expected time, using $O(n + u^\varepsilon)$ space, for any $\varepsilon > 0$. The idea behind this result is simple: Place all the elements $i \in U$ with probability $p_i \geq (1/u)^\varepsilon$ into a hash table T, together with a pointer to their predecessor in X. Separately, store all the elements of X into a y-fast trie. Given that there are at most u^ε elements with probability of being queried greater than $(1/u)^\varepsilon$, the hash table requires $O(u^\varepsilon)$ space, and the total space used by the data structure is $O(n + u^\varepsilon)$. To answer a query, first the hash table is checked to see if the query (and thus the answer) is stored there. If the query is not present in the hash table, then one simply searches the y-fast trie for the answer. The expected running time of this approach is $O(\sum_{i\in T} p_i + \sum_{j\in U\setminus T} p_j \cdot \log\log u)$, which they show to be bounded by $O(\log(H/\varepsilon))$. Selecting for T the elements with probability at least $(1/2)^{\log^\varepsilon u}$, instead of $(1/u)^\varepsilon$, yields a data structure that requires $O(n + 2^{\log^\varepsilon u})$ space and has expected query time $O((1/\varepsilon)\log H)$. To obtain a linear-space data structure, they present a variant of the solution of Beame and Fich [14] (which uses polynomial space) combined with exponential search trees (to reduce the space down to linear). This approach yields a data structure supporting queries in $O(\sqrt{H})$ time and using linear space.

In a similar direction, Belazzougui et al. [21] studied the predecessor problem under two biased scenarios. In the first one, the queries are distributed uniformly in U, while the distribution of the input set X is arbitrary. For this case, they introduce a data structure supporting queries in constant time while using linear space. The idea is to divide U into $n\log^c n$ equally sized buckets, and use

[5]Bille et al. [30] presented an homologous data structure for distance-sensitive predecessor search in the pointer-machine model, supporting predecessor queries in $O(\log \frac{u}{s-p})$. Moreover, they mentioned that it was possible to achieve $O(\log\log \frac{u}{s-p})$ query time in the word-RAM model, citing a personal communication with Mihai Patrascu. This was confirmed by Belazzougui et al. [20], who presented the first formal proof for this result, based on z-fast tries.

a bitmap to store which buckets are full/empty. This bitmap can be encoded using $O(n)$ space so that queries take $O(1)$ time [78] (because of the universe size $n \log^c n$). Queries within the non-empty buckets can be answered using the optimal data structure for the general problem [81]. Note that a query uniformly distributed at random on U falls in an empty bucket with high probability $(1 - \frac{n}{n \log^c n}) = 1 - o(1)$, and thus such a query can be answered in constant time.

In their second scenario, the distribution of the queries is arbitrary but the input keys are drawn form a *smooth-distribution* [11, 70] on U. They provide a data structure with following time-space tradeoffs: (i) constant-time queries with high probability (w.h.p.) using $O(n^{1+\delta})$ space, for $\delta > 0$; (ii) $O(\log \log \log n)$-time queries w.h.p. using $O(n^{1+o(1)})$ space; and (iii) $O(\log \log n)$-time queries w.h.p. using $O(n)$ space. The data structure is again based on partitioning the universe into buckets, each one of $O(\log n)$-size w.h.p. The representative elements of the buckets are maintained using Patrascu and Thorup's variant of the van Emde Boas tree [81], while each bucket is represented using a q^*-heap [103]. The tradeoffs are obtained by considering different space-time parameters in the second branch of the optimal bounds [81]. As mentioned in Section 3.2.2, the use of q^*-heaps requires access to a large precomputed table of size depending on n. The solution of Belazzougui et al. [21] requires a table that occupies only $o(n)$ bits, but this still limits the result to scenarios where n (or some approximation) is known in advance. This limitation can be removed by replacing the use of q^*-heaps by the dynamic fusion tree of Patrascu and Thorup [83].

5.3 Improvements on the Redundant Space

Consider a set X of n integers from a universe U of size $u = |U|$. Suppose that we first initialize a data structure D to answer predecessor queries on X, and then we discard X. Note that using only D one can recover X by simply asking for the predecessor of every element in U. Thus, D must necessarily encode X and, in consequence, the minimum amount of information D must store is $B(n, u) = \lceil \log \binom{u}{n} \rceil$ bits. Supporting predecessor queries in $O(t)$ time, for some parameter t, seems to require space additional to $B(n, u)$ that depends also on t. This extra space, denoted $R(n, u, t)$, is known as the redundancy of the data structure [53, 54]. The total space occupied by a data structure supporting predecessor queries in $O(t)$ time can then be expressed as $B(n, u) + R(n, u, t)$. While the optimal bounds of Patrascu and Thorup [81] can provide bounds for $R(n, u, t)$ for certain cases, the complete tradeoff between the values of n, u, t, and the value of $R(n, u, t)$ remains open.

Grossi et al. [54] presented lower and upper bounds for $R(n, u, t)$. As they noted, from the lower bounds of Patrascu and Thorup [81] one can already infer, for instance, that $R(n, u, 1)$ can be $o(n)$ only when $n = \text{polylog}\, u$ (the degenerate case of small sets solvable in $O(1)$ time using the fusion node) or $u \in O(n\, \text{polylog}\, n)$. For $u = n^{O(1)}$, the lower bound for $B(n, u) + R(n, u, 1)$ is $\Omega(n^{1+\delta})$ for any fixed $\delta > 0$ (because the value of a in Equation (1) must be $\Omega(\text{polylog}\, n)$ for constant query time). In this case, the redundancy is considerably higher than $B(n, u)$, since $B(n, u) = O(n \log u) = o(R(n, u, 1))$. In terms of upper bounds, Grossi et al. [54] showed that the variant of van Emde Boas tree introduced by Patrascu and Thorup [81] (for the second branch of the optimal bounds) can answer predecessor queries in $O(\log \delta)$ time using $O(B(n, u))$ bits of space when $n \geq u / \log^\delta(u)$, for $\delta > 1$. For the general case, they introduce a data structure running in time $t = O(\frac{\log n}{\log \log u})$ and with redundancy $R(n, u, t) \in O(n \log \log u)$ bits, provided that one has access to a precomputed table of u^γ bits, for some constant $\gamma < 1$. Their structure is essentially a B-Tree with branching factor $b = O(\sqrt{\log u})$ represented succinctly, and the precomputed table allows to support constant-time queries over the sets of b keys within each node.

The structure of Grossi et al. [54] is in fact an example of an *index* for the predecessor problem. In this case, we distinguish not only between the space in the data structure occupied by the input keys and the additional redundant space but also distinguish the cost of making probes to the redundant part of the data structure (called the index), and making probes to the input keys. The main motivation for this distinction is that, if the index is small enough so that it can fit in cache, then probes to the index would run considerably faster than those to the input keys. The structure of Grossi et al. [54] makes only 1 access to the input data.

Inspired in this index, Cohen et al. [42] introduced the γ-node, a data structure based on fusion nodes that stores a set of $w/\log w$ keys using an index with $O(1)$ words, and supports queries in time $O(\log w)$ while making only $O(1)$ probes to the input. For general sets, they first divide the keys into consecutive slots of $w/\log w$ keys. They index each slot using a γ-node, and index a representative key of each slot (e.g., the smaller key) using another linear-space data structure (e.g., a fusion tree or a y-fast trie). Combining the approach above with the optimal data structure of Patrascu and Thorup [81], for instance, yields an index that answers queries making only $O(1)$ probes to the input, an optimal number of probes to the index, and with running time $O(\#\text{probes} + \log w)$. This approach works only for the static version of the problem, however; the existence of an efficient dynamic index was left open by Cohen et al. [42].

Indeed, the more basic goal of obtaining an optimal-time index using space $O(B(n,u))$ is not hard to achieve. Using simple transformations, Patrascu and Thorup's optimal data structure [81] can be turned into a data structure in which $R(n,t,u) = O(n)$, and with optimal query time for that space. The universe is partitioned into n slots of size u/n. The ith slot stores n_i elements from a universe of size $u' = u/n$. Using $a = \log u' = \log(\log u - \log n)$ in the optimal tradeoffs, yields a data structure with $n_i 2^a = n_i \log(u/n)$ bits of space, and with optimal query time. Summing the space over all the slots yields a total space of $n \log(u/n)$ bits. Note that for $n \leq u/2$, $B(n,u) \sim n\log(u/n) + 1.44n$ by using Stirling's approximation, and thus the total space is less than $B(n,u)$. The mapping of elements to the respective buckets can be achieved by means of a bit vector of $2n$ bits, with support for constant-time `rank` and `select`, which requires $o(n)$ additional bits [41, 75]: $\texttt{rank}_b(i)$ is the number of bits $b \in \{0,1\}$ up to position i in the bit vector; $\texttt{select}_b(j)$ is the position of the jth occurrence of b in the bit vector. The total space of the structure is, therefore, $B(n,u) + O(n)$ bits.

This idea of splitting the universe into u/n buckets actually works for any linear-space data structure: it improves the space from $O(n \log u)$ bits to down to $O(n \log(u/n))$ bits while preserving the query time. The idea was first hinted by Patrascu and Thorup [81], and made explicit by Belazzougui and Navarro [22]. They described another variant of the van Emde Boas tree that uses $O(n \log(u/n))$ bits of space with query time $t = O(\log \frac{\log u - \log n}{\log w})$ (replacing the $\log(\log u - \log n)$ term in the denominator in the second branch of Equation (1) by $\log w$). The main difference with Patrascu and Thorup's variant of van Emde Boas trees is that, instead of stopping the recursion when the key length $l \leq a$ and using tabulation for that base case, they stop the recursion when $l < (\log w)/2$ and switch to a more efficient data structure. In this base case, the size of universe is at most $\sqrt{w}/2$, and thus the size t of the sets is also bounded by $\sqrt{w}/2$. They introduce a data structure for such sets supporting constant-time queries while using only $O(1)$ words.

Patrascu [78] presented a data structure in which $R(n,u,t) \leq \frac{u}{((\log u)/t)^t} + O(u^{3/4} \operatorname{polylog} u)$ bits, obtaining an exponential dependence between the running time of queries and the redundancy of the data structure. Their solution combines bucketing with a succinct representation of B-Trees. Let $b \geq 2$ such that $b \log b = \frac{\varepsilon \log u}{t}$, and let $r = b^t = (\frac{\log u}{t})^{\Theta(t)}$. They partition the universe into u/r buckets of size r and represent each bucket using their variant of B-Trees, which solve predecessor queries in time $O(\log_b r) = O(t)$. Queries over the u/r representative elements of the buckets can be supported in $O(\log t)$ time using using $\frac{u}{r} r^{\Omega(1/t)} \log u \leq \frac{u}{r} b \log u = u/b^{\Theta(t)}$ bits of space. For this sake they use the variant of van Emde Boas trees for the second branch of the optimal tradeoffs of Patrascu and Thorup [81]. However, this data structure requires a large precomputed table of size $O(u^{\varepsilon})$ to support constant-time queries in the B-Trees, for some constant $\varepsilon > 3/4$. Patrascu and Viola [84] proved that for values of $u \leq n \operatorname{polylog} n$, the bound for $R(n,u,t)$ by Patrascu [78] is the best possible for the more general problem of answering `rank` queries in a bit vector with n 1s and $u - n$ 0s.

For the special case of polynomial universes (i.e., $u = n^{\alpha}$, for $\alpha = \Theta(1)$), and word size $w = \Theta(\log u)$, Pibiri and Venturini [87] introduced a data structure that supports predecessor queries in optimal time over an ordered set X of n integers using $n\lceil \log \frac{u}{n} \rceil + 2n + o(n)$ bits of space. Their variant stores the integers using the Elias-Fano [46, 47] representation of the set, and supports predecessor queries in optimal $O(\min\{1 + \log \frac{u}{n}, \log\log n\})$ time. When $1 \leq \alpha \leq 1 + \frac{\log\log n}{\log n}$, the minimum in the time bound occurs in the term $1 + \log \frac{u}{n}$, which they achieve by means an of auxiliary `rank`/`select` data structures on top of the Elias-Fano representation of X. For values of $\alpha > 1 + \frac{\log\log n}{\log n}$ the minimum occurs in the $\log\log n$-term of the time bound. In this case, the $o(n)$-bits redundancy is made possible by splitting X into $\lceil \frac{n}{\log^2 u} \rceil$ buckets of size $\log^2 u$. Within the buckets, queries are answered by binary searching directly over the Elias-Fano representation of X in $O(\log\log^2 u) = O(\log\log n)$ time. To locate the buckets in $O(\log\log n)$ time, the representative elements of the buckets are stored in a y-fast trie, using $O(\frac{n \log u}{\log^2 u}) = o(n)$ bits.

5.4 Batched Queries and Integer Sorting

In the *Batched Predecessor* problem, along with the set X of n keys, a set $Q = \{q_1, q_2, \ldots, q_m\}$ of m queries is given. The goal is then to compute the set $P = \{p_1, p_2, \ldots, p_m\}$ such that each p_i is the predecessor of q_i in X. In the comparison model, this problem can be solved in optimal time $O(m \log(n/m) + m)$ by combining merging and binary search [23].

Karpinski and Nekrich [62] presented a static data structure that uses $O(n^m)$ space and answers the m queries in $O(\sqrt{\log n})$ total time, for any $m \in O(\sqrt{\log n})$. Their solution is based on a variant of the van Emde Boas tree [99] that uses the parallel hashing technique of Beame and Fich [14] to reduce the length of the m query keys in parallel. Their approach yields, for instance, a data structure that answers $\sqrt{\log n}$ queries in constant amortized time per query using space within $O(n^{\varepsilon\sqrt{\log n}})$. For bounded universes (e.g., $\log\log u \in o(\sqrt{\log n})$), their approach also leads to an $O(n^m)$-space static data structure that answers m queries in $O(\log\log u)$ total time, for $m \in O(\log\log u)$. For example, this yields a data structure for the case when $u \in n^{\log^{O(1)} n}$ that, using space in $O(n^{\varepsilon \log\log n})$, answers $O(\log\log u)$ queries in constant amortized time each.

This variant of predecessor search has also been studied in external memory [2]. Bender et al. [23] considered batched predecessor queries in external memory for the case when $m < n^c$, for some constant $c < 1$, and $m = \Omega(B)$. They provided different tradeoffs between preprocessing and query times. They show that a batch of predecessor queries cannot be answered asymptotically faster than handling them one by one if the preprocessing time is polynomial with respect to n, but that it can be answered in $O(\frac{\log n}{B})$ I/Os per query if we allow exponential preprocessing time. Bender et al. [25] studied a variant of this problem in which the length of the keys in Q is different from that the keys in X, and provide lower and upper bounds sensitive not only to n and m but also to the lengths of the keys, and to how the elements from Q and X interleave.

Integer sorting. A particularly interesting special case of batched predecessor is when $m = \Theta(n)$, which is closely related to *Integer Sorting* [9, 10, 51, 57, 58, 63, 96]. On the one hand, if we can sort n integers from a universe U of size u in time $T(n,w,u)$, then the m queries can be computed using $O(\frac{T(n,w,u)}{n})$ time per query by simply sorting X and Q, merging the results, and reporting the output. On the other hand, if we can preprocess X and compute the predecessors of the elements in Q in total time $T(n,w,u)$, then we can sort a set X of n integers in $O(T(n,w,u) + n)$ time by solving an instance of the batched predecessor problem with $X = Q = S$, and then building the sorted output with the help of a dictionary mapping each element to its predecessor. The developments and techniques that lead to faster integer sorting algorithms and predecessor data structures are closely related. In fact, some of the data structures presented here (like fusion trees [51] and exponential search trees [9]) were originally introduced as intermediate tools for the ultimate goal of sorting integers, or presented as priority queues (like van Emde Boas trees [101]), which are known to be equivalent to sorting [98].

After some initial results by Paul and Simon [86] and Kirkpatrick and Reisch [63] (based on the exponential length reduction technique introduced for van Emde Boas trees [99]), Fredman and Willard [51] introduced the fusion tree and presented the first deterministic linear-space algorithm for sorting integers in $o(n \log n)$ time (for all possible values of n, w and u). This is achieved by inserting all the elements in a fusion tree at an amortized $O(\frac{\log n}{\log \log n})$ time per element, and then traversing the tree to obtain the sorted output. Moreover, they showed that, using randomization, fusion trees could be combined with y-fast tries to obtain an $O(n\sqrt{\log n})$ expected time sorting algorithm. While these results were later improved, techniques based on word-level parallelism like the ones used in the fusion node remained at the center of every integer sorting algorithm that followed [55].

One of such improvements was presented by Andersson [9], whose exponential search trees could achieve updates in amortized $O(\sqrt{\log n})$ time, and with worst-case $O(\sqrt{\log n})$-time queries. Inserting the elements to sort in an exponential search tree, and then iteratively querying for predecessors starting from the maximum element, yields a deterministic worst-case sorting algorithm running in $O(n\sqrt{\log n})$time. Since then, exponential search trees became a key tool for faster integer sorting algorithms [55, 56, 57, 96].

Thorup [96] built upon Andersson's solution [9] and introduced a deterministic algorithm sorting n integers in $O(n(\log \log n)^2)$ time. The key component of this algorithm is a data structure that answers n predecessor queries over keys of w/q-bits in time $O(\frac{n}{q} \log \log n)$ after a preprocessing step consuming $O(n^{2+\varepsilon})$ time. The idea is to use a trie of height $\log n \log \log n$ over an alphabet of size $u^{1/\log n \log \log n}$. For each query, binary search is used to find the deepest node of the trie that corresponds to a prefix of the query, as in van Emde Boas trees [99], in time $O(\log(\log n \log \log n)) = O(\log \log n)$. After this, the original problem is reduced to a set of subproblems of combined size n over a universe of size $u^{1/\log n \log \log n}$. In this smaller universe, linear-time sorting can be used to solve all the sub-problems in $O(n)$ total time. With the help of this batched predecessors data structure, insertions in an exponential search trees can be implemented in $O((\log \log n)^2)$ amortized time by performing insertions in batches. For this, the insertions at a node of the tree are buffered until the size d of the buffer given by the number of children of the node. Once the buffer reaches its maximum size, the d inserted integers are passed down the tree to be inserted at the children.

Currently, the fastest deterministic linear-space integer sorting algorithm is the one introduced by Han [57], which runs in $O(n \log \log n)$ time in the worst case. If randomization is allowed, then it is possible to sort n integers in $O(n\sqrt{\log \log n})$ expected time using linear space [59]. Han's deterministic algorithm [57], as Thorup's solution [96], is also based on performing batched insertions in an exponential search tree, although this time number of insertions that are buffered at a node of degree d is d^2, instead of d. Moreover, once the buffer is filled, the inserted integers are passed down only one level instead of all the way down the tree, until all the elements to sort are passed down one level. Note that if the degree of the root of the exponential search tree is $n^{\frac{1}{k}}$, then after all the integers in the set to sort have been passed down to the next level, the trees rooted at nodes in the second level of the tree induce an ordered partition of the set into $n^{\frac{1}{k}}$ subsets, of size

$n^{\frac{k-1}{k}}$. Thus, Han [57] interprets the passing down steps as an ordered partition problem, and shows how to solve it adapting the *signature sort* algorithm [10] to this setting by using a novel parallel hashing technique.

Integer sorting algorithms (and predecessor data structures) can be used to sort floating-point numbers as well. In fact, the IEEE 754 floating-point standard was designed so that if each element in a set of floating-point numbers is mapped to the integer represented by the same bit string, the relative order of the elements is preserved [85, Section 3.5]. Thus, floating-point numbers can be sorted by means of integer-sorting algorithms. Until recently, it was widely believed that for real numbers this was not the case, and that sets of real numbers had to be sorted by using comparison-based algorithms. In 2020, Han [58] showed that a set of real numbers can be mapped to a set of integers in $O(n\sqrt{\log n})$ time so that the respective order of the mapped elements is preserved, and thereafter be sorted with any integer sorting algorithm. The real-to-integer mapping procedure relies on Patrascu and Thorup's dynamic data structure [83] to achieve the claimed running time using only linear space.

5.5 Related Problems

Techniques and data structures introduced for predecessor search have been key not only for integer sorting but also for several other problems, even when used as "black boxes." For instance, the introduction of fusion trees immediately implied an improvement for all algorithms with integer inputs and running times dominated by integer sorting algorithms; Willard [103] explored several of them. For data structures in which only the relative order of the elements in the universe matters to support queries and updates (and not their exact values), predecessor data structures became a powerful tool for reducing any arbitrary universe to the simpler rank space (i.e., $\{1, \ldots, n\}$). This have been extensively used, for instance, in dominance and containment problems in computational geometry [36–39]. Moreover, several problems in diverse fields of computer science can be interpreted as generalizations of predecessor search [7, 14, 35, 72, 82]. We briefly overview some of the most obvious of these generalizations, and mention slight variations of predecessor data structures that have impacted their state of the art.

Range Searching and Interval Stabbing. In the *Interval Stabbing* problem one is given a set I of (non-necessarily disjoint) intervals, and the goal is to preprocess I so that given a query value (or point) q, one can efficiently report the intervals in I containing p. *Range Searching* is the dual problem in which the elements represented by the data structure are points, the queries are intervals (or hyper-rectangles in general), and the goal is to report all the elements of the set within a given query interval. Predecessor search over a set X can clearly be solved with interval stabbing data structures: simply map each element p of X to an interval $[p, s-1]$ starting at p and ending just before the successor s of p, and store them in the data structure; the predecessor of a value q is then the left boundary of the interval stabbed by q in the data structure. Predecessor queries can also be supported using data structures for range searching, because interval stabbing trivially reduces to two-dimensional range searching by mapping each interval $[a, b]$ to the two-dimensional point (a, b), and each query value q to the rectangle $[-\infty, q] \times [q, \infty]$.

While range searching in two dimensions generalizes predecessor search, in one dimension the inverse is true: range searching can easily be solved using data structures for predecessor search. For instance, predecessor data structures based on tries (like van Emde Boas trees, and the ones after it) with query time $Q(n)$ can be adapted to report all the elements within a given query range in $O(Q(n) + k)$ time, where k is the size of the output. However, surprisingly, one-dimensional range queries can be supported in time exponentially faster than predecessor queries. For example, Alstrup et al. [6] presented a static linear-space data structure that can retrieve an arbitrary element within the query interval (or report there are no elements) in constant time, and report all the k elements in $O(k)$ time. For dynamic sets, Mortensen et al. [74] introduced a data structure that finds an arbitrary element within a query interval in $O(\log\log\log u)$ time, and supports updates in time $O(\log\log u)$. Their data structure is inspired by van Emde Boas trees, and use y-fast tries as an auxiliary data structure.

Predecessor data structures and techniques are also key for range searching in higher dimensions, because they allow reducing arbitrary universes to rank space. The current best results for the static two-dimensional version were presented by Chan et al. [36], and one of the elements in their solution is the predecessor data structure of Grossi et al. [54]. In the dynamic version, Mortensen [73] presented the first data structure with sub-logarithmic query and update times. He introduced a data structure for a variant of the *colored predecessor* problem in which the elements of the set are associated with a color, and the goal is to report the predecessor of a given query element with a given color. The data structure was a variant of van Emde Boas trees supporting queries in $O(\log^2 \log n)$ time in the worst case, and in $O(\log\log n)$ expected time when randomization is allowed. Current best results for the dynamic version were presented by Chan and Tsakalidis [39], and their solution depends also on the data structure for colored predecessor search described by Mortensen [73]. The relevance of the colored predecessor problem for various geometric retrieval problems comes from the fact data structures for colored predecessor can act as a direct replacement in situations where the classical fractional cascading technique [40, 94] is used, especially in dynamic settings, for which Mortensen's data structure [73] offers better tradeoffs.

Orthogonal Point Location. In this problem, one is given a subdivision of the space into disjoint orthogonal cells (i.e., the edges are vertical or horizontal), and the goal is to build a data structure so that the cell containing a given query point can be quickly identified. This problem generalizes the *Persistent Predecessor Search* problem [35], which is in turn a generalization of predecessor search. Chan and Patrascu [38] considered two-dimensional point location queries. They introduced a simplified version of the fusion tree combining both cardinality and length reductions, and extended it to support a special type of point location queries in $O(\frac{\log n}{\log\log n})$ time. Then, by combining this data structure with a a slight variation of the exponential search trees they obtain a data structure for the general *Point Location* problem in two dimensions. Later, Chan [35] improved these results by introducing an optimal linear-space data structure. For this, he introduced a partially persistent[6] version of van Emde Boas trees, supporting updates in $O(\log\log u)$ expected amortized time, and queries over any of the previous versions of the data structure in $O(\log\log u)$ time. In three dimensions, the current best bounds for both problems were introduced by Chan et al. [37], and their solution is based on a recursive partition of the space inspired by the cluster-galaxy approach of van Emde Boas trees.

Prefix Search. Another generalization of predecessor search is prefix search in strings. In this problem one is given a set of strings S and a pattern p, and the goal is to return the longest prefix of p that is also a prefix of some string in S. The z-fast trie [19], for instance, was introduced in this context of prefix and predecessor search for strings. Given a set S of prefix-free strings (i.e., no string of S is a prefix of another) from a binary alphabet and of variable length within $O(w)$, z-fast tries can answer prefix queries in $O(\log\max(|p|, |p^+|, |p^-|))$ time ($|p^\pm|$ is the successor/predecessor of p in S). For strings from a general alphabet of size σ, Fischer and Gawrychowski [49] presented a weighted variant of exponential search trees in which querying or updating an element of weight

[6]A data structure is partially persistent if it supports updates and allows querying past versions of the data structure.

w in a set of total weight W costs $O(\frac{\log\log u}{\log\log\log u}\log\frac{\log W}{\log w})$ time, replacing the $\log\log n$ term in the bounds of dynamic exponential search trees[7] by a $\log\frac{\log W}{\log w}$ term. Combining this variant with other ideas they introduce a dynamic linear-size data structure answering prefix queries for a pattern p in time $O(|p| + \log\log\sigma)$. Other variants of predecessor search data structures have been a valuable component of solutions to problems on indexing texts and searching patterns on strings [15, 27, 29, 30, 66].

Fully Indexable Dictionaries. Predecessor queries over a set X of integers can easily be implemented by means of a bit-vector with support for rank and select (namely, a fully indexable dictionary [90]); recall Section 5.3. For this, simply consider a bit vector b with u bits, and with a 1 in the ith position if and only if $i \in S$. The predecessor of q in X is then given by $\texttt{select}_1(\texttt{rank}_1(x))$. Thus, the lower bound from Patrascu and Thorup [81, 82] for predecessor search provides a trade-off between space ocupied by the dictionary and the minimum of the running times of rank and select. For instance, from Patrascu and Thorup's bounds, we know that constant running times for both rank and select are only possible when the universe size is close to n, in particular, for $u \in O(n \operatorname{polylog} n)$. Grossi et al. [54] showed that both lower and upper bounds for the predecessor problem can be transformed into lower and upper bound for the fully indexable dictionary problem, and vice versa.

Finally, since the problems above are generalizations of predecessor search, Patrascu and Thorup's lower bounds [81, 82] hold for them as well. Moreover, the lower bounds have inspired similar results (either by means of reductions, or by slight variations) to a variety of other problems, including two-dimensional dominance queries [82], marked ancestor problems [7], longest common extensions in trees [28], range minimum queries [68], and representing sequences of elements from an arbitrary alphabet, with support for random access, rank, and select [22]. This list will certainly grow with time.

5.6 Practical Solutions

Many of the data structures we have presented, including the optimal results of Patrascu and Thorup [81–83], are not good candidates for a verbatim implementation, because of the large constants involved. More basic ideas, like those of van Emde Boas, are more promising in practice, but still may offer little improvement over binary search. Experience shows that good ideas behind theoretical results are always valuable, but they must be identified and combined with a good deal of algorithm engineering. This is especially relevant in the predecessor problem, where all the complexities are sublogarithmic, and therefore the impact of the constants is high. In this section, we collect some of those algorithm engineering efforts.

In 2004, Dementiev et al. [44] described a tuned version of van Emde Boas trees and compared it experimentally with various comparison-based data structures (like Red-Black trees [43, Chapter 13]), demonstrating that their variant achieves a significantly better performance. In their solution, highly specialized for 32-bit keys, they made three changes with respect to the original data structure. First, in the structure of the nodes, the galaxy is maintained using a bit vector and an array of pointers instead of a recursive van Emde Boas tree. Thus, the root of the tree has a bit vector of size 2^{16}, which represents the 16 highest bits of each key. Second, the recursion is stopped when the length of the keys is 8; thus, the tree has only three levels. Finally, the clusters at the second level are maintained using hash tables, and in the third level hash tables are used to store the direct answer. The size of these hash tables increase and decrease with updates as needed, ranging between 4 and $2^8 = 256$. While at the time of Dementiev et al.'s work [44] the most

[7]This bound is achieved when combining exponential search trees with Beame and Fich's solution; see Section 3.3.

common word-size in computers was 32 bits, this has changed today to 64 bits. Although their solution is highly efficient in time, extending it to 64-bit keys seem unfeasible. In this case, the size of the root bit vector and cluster array would increase up to 2^{32} bits and words, respectively, which is (to date) unacceptable.

In 2010, Nash and Gregg [76] experimentally compared various dynamic data structures for searching predecessors over 32 and 64 bit integer keys, including Dementiev et al.'s variant of van Emde Boas trees [44]. They introduced a data structure that combines a trie variant known as burst trie [60] with bucketing. According to their experiments, this data structure performed better in time than the other data structures they considered in the comparison. Their work was inspired by a similar article presented by Rahman et al. [89] almost ten years earlier.

For fusion trees, exponential search trees, and similar structures, we know of no practical implementation other than simple B-Trees tuned for predecessor search [89]. In the case of fusion trees, for instance, note that the most common word size in today's computers is $w = 64$ bits, and thus the fusion node would be in charge of maintaining sets of just $\lfloor w^{1/4} \rfloor = \lfloor 2\sqrt{2} \rfloor = 2$ keys. Clearly, the overhead of these complex structures does not pay off.

6 CONCLUSIONS

There has been a good deal of research on the Predecessor problem, from the first data structure introduced by van Emde Boas [99] in 1977, to the optimal results of Patrascu and Thorup [81, 82] in 2006–2007, which completely settled the static version of the problem, and their results in 2014 [83], which did the same for the dynamic version of the problem when randomization is allowed. However, several issues remain open for the dynamic case of the problem, and for the special cases and variants described in the previous section. We mention some of the questions on predecessor search that remain open.

Dynamic predecessor bounds. The most fundamental of those unsolved questions is the deterministic complexity of the dynamic predecessor problem: the optimal dynamic data structure of Patrascu and Thorup [83] heavily relies on randomization. It is even open whether the running time of lower bounds of Patrascu and Thorup [83] can be matched by data structures with upper bounds that apply with high probability, instead of in expectation. The bounds of Patrascu and Thorup [83] apply for the maximum between update and query time. The optimal bound for the static case allow to argue that, even if no updates are allowed, one cannot improve significantly the query time (when compared to the dynamic version). However, it is unknown whether there is any data structure in which updates can be supported considerably faster than queries, provided that the query time of the data structure is already close to the optimal bound.

Distance-sensitive queries based on cardinality reduction. There are no lower bounds for any of the special cases reviewed here. The optimal complexity of this case is then open, even in the static setting. For the general predecessor problem, obtaining optimal results required a wise combination of data structures based on cardinality and length reduction. Current data structures supporting distance-sensitive queries are essentially versions of the van Emde Boas tree, and thus based on length reduction. Whether there are distance-sensitive data structures based on cardinality reduction is open. It is not even clear how the running time of those data structures, if they existed, should look. While the bounds depending on u translate naturally to bounds depending on Δ (e.g., $O(\log\log u)$ to $O(\log\log\Delta)$) this is not the case for bounds depending exclusively on n and w, such as the $O(\log_w n)$ bound of the fusion trees.

Dynamic data structures for biased queries. The data structures introduced by Bose et al. [32] for the Biased Predecessor problem also work only under static settings. In this variant, finding an efficient dynamic data structure might be harder than in the general case, especially when considering that the distribution of the queries might change upon insertions and deletions of elements. If the distribution is considered fixed, and applies to both queries and updates, then using dynamic dictionaries might yield a simple solution. In the comparison model, the splay-trees introduced by Sleator and Tarjan [95] improve, when compared to traditional balanced trees, the running time of predecessor queries for heavily biased distributions in which a small set of elements is accessed frequently. This remains true even if the working set changes over time. It is unknown whether there is an analogous dynamic data structure for integer sets in the RAM model in which, like in splay-trees, queries to frequently or recently accessed elements are supported considerably faster than the general case.

Dynamic indexes and succinct data structures. In general, there is very little development on dynamic versions of the structures aimed at lowering the redundancy over the raw data. The indexes introduced by Grossi et al. [54] and Cohen et al. [42] work only for the static predecessor problem, and the existence of efficient dynamic indexes remains open. The succinct structure of Belazzougui and Navarro [22] is also static.

REFERENCES

[1] Peyman Afshani, Cheng Sheng, Yufei Tao, and Bryan T. Wilkinson. 2014. Concurrent range reporting in two-dimensional space. In *Proceedings of the 25th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'14)*, Chandra Chekuri (Ed.). SIAM, 983–994.

[2] Alok Aggarwal and Jeffrey Scott Vitter. 1988. The input/output complexity of sorting and related problems. *Commun. ACM* 31, 9 (1988), 1116–1127.

[3] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. 1976. On finding lowest common ancestors in trees. *SIAM J. Comput.* 5, 1 (1976), 115–132.

[4] Miklós Ajtai. 1988. A lower bound for finding predecessors in Yao's cell probe model. *Combinatorica* 8, 3 (1988), 235–247.

[5] Miklós Ajtai, Michael L. Fredman, and János Komlós. 1984. Hash functions for priority queues. *Info. Control* 63, 3 (1984), 217–225.

[6] Stephen Alstrup, Gerth Stølting Brodal, and Theis Rauhe. 2001. Optimal static range reporting in one dimension. In *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing (STOC'01)*. 476–482.

[7] Stephen Alstrup, Thore Husfeldt, and Theis Rauhe. 1998. Marked ancestor problems. In *Proceedings of the 39th Annual Symposium on Foundations of Computer Science (FOCS'98)*. 534–544.

[8] Arne Andersson. 1995. Sublogarithmic searching without multiplications. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science (FOCS'95)*. 655–663.

[9] Arne Andersson. 1996. Faster deterministic sorting and searching in linear space. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science (FOCS'96)*. 135–141.

[10] Arne Andersson, Torben Hagerup, Stefan Nilsson, and Rajeev Raman. 1998. Sorting in linear time? *J. Comput. Syst. Sci.* 57, 1 (1998), 74–93.

[11] Arne Andersson and Christer Mattsson. 1993. Dynamic interpolation search in $o(\log\log n)$ time. In *Proceedings of the 20th International Colloquium on Automata, Languages and Programming (ICALP'93)*. 15–27.

[12] Arne Andersson and Mikkel Thorup. 2007. Dynamic ordered sets with exponential search trees. *Journal of the ACM* 54, 3 (2007), 13.

[13] Lars Arge, Paolo Ferragina, Roberto Grossi, and Jeffrey Scott Vitter. 1997. On sorting strings in external memory. In *Proceedings of the 29th Annual ACM Symposium on the Theory of Computing (STOC'97)*, Frank Thomson Leighton and Peter W. Shor (Eds.). ACM, 540–548.

[14] Paul Beame and Faith E. Fich. 2002. Optimal bounds for the predecessor problem and related problems. *J. Comput. Syst. Sci.* 65, 1 (2002), 38–72.

[15] Djamal Belazzougui. 2012. Worst-case efficient single and multiple string matching on packed texts in the word-RAM model. *J. Discrete Algor.* 14 (2012), 91–106.

[16] Djamal Belazzougui. 2016. Predecessor search, string algorithms and data structures. In *Encyclopedia of Algorithms*. 1605–1611.

[17] Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. 2009. Monotone minimal perfect hashing: Searching a sorted table with $O(1)$ accesses. In *Proceedings of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'09)*. 785–794.

[18] Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. 2010. Fast prefix search in little space, with applications. In *Proceedings of the 18th Annual European Symposium on Algorithms (ESA'10)*. 427–438.

[19] Djamal Belazzougui, Paolo Boldi, and Sebastiano Vigna. 2010. Dynamic z-fast tries. In *Proceedings of the 17th International Symposium on String Processing and Information Retrieval (SPIRE'10)*. 159–172.

[20] Djamal Belazzougui, Paolo Boldi, and Sebastiano Vigna. 2012. Predecessor search with distance-sensitive query time. *CoRR* abs/1209.5441.

[21] Djamal Belazzougui, Alexis C. Kaporis, and Paul G. Spirakis. 2018. Random input helps searching predecessors. In *Proceedings of the 11th International Conference on Random and Exhaustive Generation of Combinatorial Structures (GASCom'18)*. 106–115.

[22] Djamal Belazzougui and Gonzalo Navarro. 2015. Optimal lower and upper bounds for representing sequences. *ACM Trans. Algor.* 11, 4 (2015), 31:1–31:21.

[23] Michael A. Bender, Martin Farach-Colton, Mayank Goswami, Dzejla Medjedovic, Pablo Montes, and Meng-Tsung Tsai. 2014. The batched predecessor problem in external memory. In *Proceedings of the 22th Annual European Symposium on Algorithms (ESA'14)*. 112–124.

[24] Michael A. Bender, Martin Farach-Colton, and Bradley C. Kuszmaul. 2006. Cache-oblivious string B-trees. In *Proceedings of the 25th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'06)*, Stijn Vansummeren (Ed.). ACM, 233–242.

[25] Michael A. Bender, Mayank Goswami, Dzejla Medjedovic, Pablo Montes, and Kostas Tsichlas. 2020. Batched predecessor and sorting with size-priced information in external memory. *CoRR* abs/2004.13197. arxiv:2004.13197.

[26] Samuel W. Bent, Daniel Dominic Sleator, and Robert Endre Tarjan. 1985. Biased search trees. *SIAM J. Comput.* 14, 3 (1985), 545–568.

[27] Philip Bille, Mikko Berggren Ettienne, Inge Li Gørtz, and Hjalte Wedel Vildhøj. 2017. Time-space trade-offs for Lempel-Ziv compressed indexing. In *Proceedings of the 28th Annual Symposium on Combinatorial Pattern Matching (CPM'17)*. 16:1–16:17.

[28] Philip Bille, Pawel Gawrychowski, Inge Li Gørtz, Gad M. Landau, and Oren Weimann. 2016. Longest common extensions in trees. *Theor. Comput. Sci.* 638 (2016), 98–107.

[29] Philip Bille, Inge Li Gørtz, and Frederik Rye Skjoldjensen. 2017. Deterministic indexing for packed strings. In *Proceedings of the 28th Annual Symposium on Combinatorial Pattern Matching (CPM'17)*. 6:1–6:11.

[30] Philip Bille, Gad M. Landau, Rajeev Raman, Kunihiko Sadakane, Srinivasa Rao Satti, and Oren Weimann. 2015. Random access to grammar-compressed strings and trees. *SIAM J. Comput.* 44, 3 (2015), 513–539.

[31] Prosenjit Bose, Karim Douïeb, Vida Dujmovic, John Howat, and Pat Morin. 2013. Fast local searches and updates in bounded universes. *Comput. Geom.* 46, 2 (2013), 181–189.

[32] Prosenjit Bose, Rolf Fagerberg, John Howat, and Pat Morin. 2016. Biased predecessor search. *Algorithmica* 76, 4 (2016), 1097–1105.

[33] Gerth Stølting Brodal. 1997. Predecessor queries in dynamic integer sets. In *Proceedings of the 14th Annual Symposium on Theoretical Aspects of Computer Science (STACS'97)*. 21–32.

[34] Amit Chakrabarti and Oded Regev. 2004. An optimal randomised cell probe lower bound for approximate nearest neighbour searching. In *Proceedings of the 45th Symposium on Foundations of Computer Science (FOCS'04)*. 473–482.

[35] Timothy M. Chan. 2013. Persistent predecessor search and orthogonal point location on the word RAM. *ACM Trans. Algorithms* 9, 3 (2013), 22:1–22:22.

[36] Timothy M. Chan, Kasper Green Larsen, and Mihai Patrascu. 2011. Orthogonal range searching on the RAM, revisited. In *Proceedings of the 27th ACM Symposium on Computational Geometry (SoCG'11)*. 1–10.

[37] Timothy M. Chan, Yakov Nekrich, Saladi Rahul, and Konstantinos Tsakalidis. 2018. Orthogonal point location and rectangle stabbing queries in 3D. In *Proceedings of the 45th International Colloquium on Automata, Languages, and Programming (ICALP'18)*. 31:1–31:14.

[38] Timothy M. Chan and Mihai Patrascu. 2009. Transdichotomous results in computational geometry, I: Point location in sublogarithmic time. *SIAM J. Comput.* 39, 2 (2009), 703–729.

[39] Timothy M. Chan and Konstantinos Tsakalidis. 2017. Dynamic orthogonal range searching on the RAM, revisited. In *Proceedings of the 33rd International Symposium on Computational Geometry (SoCG'17)*. 28:1–28:13.

[40] Bernard Chazelle and Leonidas J. Guibas. 1986. Fractional cascading: I. A data structuring technique. *Algorithmica* 1, 2 (1986), 133–162.

[41] David R. Clark. 1996. *Compact PAT Trees.* Ph.D. Dissertation. University of Waterloo, Canada.

[42] Sarel Cohen, Amos Fiat, Moshik Hershcovitch, and Haim Kaplan. 2015. Minimal indices for predecessor search. *Info. Comput.* 240 (2015), 12–30.

[43] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms (3rd ed.).* MIT Press.

[44] Roman Dementiev, Lutz Kettner, Jens Mehnert, and Peter Sanders. 2004. Engineering a sorted list data structure for 32 bit key. In *Proceedings of the 6th Workshop on Algorithm Engineering and Experiments (ALENEX'04)*. 142–151.

[45] Marcel Ehrhardt and Wolfgang Mulzer. 2017. Delta-fast tries: Local searches in bounded universes with linear space. In *Proceedings of the 15th International Symposium on Algorithms and Data Structures (WADS'17)*. 361–372.

[46] Peter Elias. 1974. Efficient storage and retrieval by content and address of static files. *J. ACM* 21, 2 (1974), 246–260.

[47] Robert Mario Fano. 1971. *On the Number of Bits Required to Implement an Associative Memory*. Massachusetts Institute of Technology, Project MAC.

[48] Martin Farach. 1997. Optimal suffix tree construction with large alphabets. In *Proceedings of the 38th Symposium on Foundations of Computer Science (FOCS'97)*. IEEE Computer Society, 137–143.

[49] Johannes Fischer and Pawel Gawrychowski. 2015. Alphabet-dependent string searching with wexponential search trees. In *Proceedings of the 26th Annual Symposium on Combinatorial Pattern Matching (CPM'15)*. 160–171.

[50] M. L. Fredman, J. Komlós, and E. Szemerédi. 1984. Storing a sparse table with $O(1)$ worst case access time. *J. ACM* 31, 3 (1984), 538–544.

[51] Michael L. Fredman and Dan E. Willard. 1993. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.* 47, 3 (1993), 424–436.

[52] Michael L. Fredman and Dan E. Willard. 1994. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comput. Syst. Sci.* 48, 3 (1994), 533–551.

[53] Anna Gál and Peter Bro Miltersen. 2007. The cell probe complexity of succinct data structures. *Theor. Comput. Sci.* 379, 3 (2007), 405–417.

[54] Roberto Grossi, Alessio Orlandi, Rajeev Raman, and S. Srinivasa Rao. 2009. More haste, less waste: Lowering the redundancy in fully indexable dictionaries. In *Proceedings of the 26th International Symposium on Theoretical Aspects of Computer Science (STACS'09)*. 517–528.

[55] Torben Hagerup. 1998. Sorting and searching on the word RAM. In *Proceedings of the 15th Annual Symposium on Theoretical Aspects of Computer Science (STACS'98)*. 366–398.

[56] Yijie Han. 2001. Improved fast integer sorting in linear space. *Info. Comput.* 170, 1 (2001), 81–94.

[57] Yijie Han. 2004. Deterministic sorting in $O(n \log\log n)$ time and linear space. *J. Algor.* 50, 1 (2004), 96–105.

[58] Yijie Han. 2020. Sorting real numbers in $O(n\sqrt{\log n})$ time and linear space. *Algorithmica* 82, 4 (2020), 966–978.

[59] Yijie Han and Mikkel Thorup. 2002. Integer sorting in $O(n\sqrt{\log\log n})$ expected time and linear space. In *Proceedings of the 43rd Symposium on Foundations of Computer Science (FOCS'02)*. 135–144.

[60] Steffen Heinz, Justin Zobel, and Hugh E. Williams. 2002. Burst tries: A fast, efficient data structure for string keys. *ACM Trans. Info. Syst.* 20, 2 (2002), 192–223.

[61] Wing-Kai Hon, Tak Wah Lam, Rahul Shah, Siu-Lung Tam, and Jeffrey Scott Vitter. 2011. Cache-oblivious index for approximate string matching. *Theor. Comput. Sci.* 412, 29 (2011), 3579–3588.

[62] Marek Karpinski and Yakov Nekrich. 2005. Predecessor queries in constant time? In *Proceedings of the 13th Annual European Symposium on Algorithms (ESA'05)*. 238–248.

[63] David G. Kirkpatrick and Stefan Reisch. 1984. Upper bounds for sorting integers on random access machines. *Theor. Comput. Sci.* 28 (1984), 263–276.

[64] Donald E. Knuth. 1973. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley.

[65] Donald E. Knuth. 1977. *Notes on the van Emde Boas Construction of Priority Deques: An Instructive Use of Recursion*. Classroom notes. Stanford University.

[66] Tsvi Kopelowitz. 2012. On-line indexing for general alphabets via predecessor queries on subsets of an ordered list. In *Proceedings of the 53rd Annual IEEE Symposium on Foundations of Computer Science (FOCS'12)*. 283–292.

[67] A. Levitin. 2007. *Introduction to the Design and Analysis of Algorithms* (2nd ed.). Addison-Wesley.

[68] Mingmou Liu and Huacheng Yu. 2020. Lower bound for succinct range minimum query. *CoRR* abs/2004.05738.

[69] Kurt Mehlhorn and Stefan Näher. 1990. Bounded ordered dictionaries in $O(\log\log N)$ time and $O(n)$ space. *Info. Process. Lett.* 35, 4 (1990), 183–189.

[70] Kurt Mehlhorn and Athanasios K. Tsakalidis. 1993. Dynamic interpolation search. *J. ACM* 40, 3 (1993), 621–634.

[71] Peter Bro Miltersen. 1994. Lower bounds for union-split-find related problems on random access machines. In *Proceedings of the 26th Annual ACM Symposium on Theory of Computing (STOC'94)*. 625–634.

[72] Peter Bro Miltersen, Noam Nisan, Shmuel Safra, and Avi Wigderson. 1998. On data structures and asymmetric communication complexity. *J. Comput. Syst. Sci.* 57, 1 (1998), 37–49.

[73] Christian Worm Mortensen. 2006. Fully dynamic orthogonal range reporting on RAM. *SIAM J. Comput.* 35, 6 (2006), 1494–1525.

[74] Christian Worm Mortensen, Rasmus Pagh, and Mihai Patrascu. 2005. On dynamic range reporting in one dimension. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing (STOC'05)*. 104–111.

[75] J. Ian Munro. 1996. Tables. In *Proceedings of the 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'96)*. 37–42.

[76] Nicholas Nash and David Gregg. 2010. Comparing integer data structures for 32- and 64-bit keys. *ACM J. Exper. Algor.* 15 (2010).
[77] Mihai Patrascu. 2008. *Lower Bound Techniques for Data Structures.* Ph.D. Dissertation. Massachusetts Institute of Technology, USA.
[78] Mihai Patrascu. 2008. Succincter. In *Proceedings of the 49th Annual IEEE Symposium on Foundations of Computer Science (FOCS'08).* 305–313.
[79] Mihai Patrascu and Erik D. Demaine. 2004. Tight bounds for the partial-sums problem. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'04)*, J. Ian Munro (Ed.). SIAM, 20–29.
[80] Mihai Patrascu and Erik D. Demaine. 2006. Logarithmic lower bounds in the cell-probe model. *SIAM J. Comput.* 35, 4 (2006), 932–963.
[81] Mihai Patrascu and Mikkel Thorup. 2006. Time-space trade-offs for predecessor search. In *Proceedings of the 38th Annual ACM Symposium on Theory of Computing (STOC'06).* 232–240.
[82] Mihai Patrascu and Mikkel Thorup. 2007. Randomization does not help searching predecessors. In *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'07).* 555–564.
[83] Mihai Patrascu and Mikkel Thorup. 2014. Dynamic integer sets with optimal rank, select, and predecessor search. In *Proceedings of the 55th IEEE Annual Symposium on Foundations of Computer Science (FOCS'14).* 166–175.
[84] Mihai Patrascu and Emanuele Viola. 2010. Cell-probe lower bounds for succinct partial sums. In *Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'10).* 117–122.
[85] David A. Patterson and John L. Hennessy. 2012. *Computer Organization and Design—The Hardware/Software Interface* (5th ed.). Academic Press.
[86] W. Paul and Janos Simon. 1980. Decision trees and random access machines. *Logic Algor.* 30 (1980), 331–340.
[87] Giulio Ermanno Pibiri and Rossano Venturini. 2017. Dynamic Elias-Fano representation. In *Proceedings of the 28th Annual Symposium on Combinatorial Pattern Matching (CPM'17).* 30:1–30:14.
[88] William Pugh. 1990. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM* 33, 6 (1990), 668–676.
[89] Naila Rahman, Richard Cole, and Rajeev Raman. 2001. Optimised predecessor data structures for internal memory. In *Proceedings of the 5th International Workshop on Algorithm Engineering (WAE'01).* 67–78.
[90] Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. 2007. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Algor.* 3, 4 (2007), 43.
[91] Milan Ruzic. 2008. Constructing efficient dictionaries in close to sorting time. In *Proceedings of the 35th International Colloquium on Automata, Languages and Programming (ICALP'08).* 84–95.
[92] Milan Ruzic. 2009. Making deterministic signatures quickly. *ACM Trans. Algor.* 5, 3 (2009), 26:1–26:26.
[93] Pranab Sen and Srinivasan Venkatesh. 2008. Lower bounds for predecessor searching in the cell probe model. *J. Comput. Syst. Sci.* 74, 3 (2008), 364–385.
[94] Qingmin Shi and Joseph JaJa. 2003. *Fast Fractional Cascading and Its Applications.* Technical Report. University of Maryland, College Park, MD.
[95] Daniel Dominic Sleator and Robert Endre Tarjan. 1983. Self-adjusting binary trees. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing (STOC'83).* 235–245.
[96] Mikkel Thorup. 1998. Faster deterministic sorting and priority queues in linear space. In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'98).* 550–555.
[97] Mikkel Thorup. 2003. On AC^0 implementations of fusion trees and atomic heaps. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'03).* 699–707.
[98] Mikkel Thorup. 2007. Equivalence between priority queues and sorting. *J. ACM* 54, 6 (2007), 28.
[99] Peter van Emde Boas. 1977. Preserving order in a forest in less than logarithmic time and linear space. *Info. Process. Lett.* 6, 3 (1977), 80–82.
[100] Peter van Emde Boas. 2013. Thirty nine years of stratified trees. In *Proceedings of the 2nd International Symposium on Computing in Informatics and Mathematics (ICSIM'13).* 1–14.
[101] Peter van Emde Boas, R. Kaas, and E. Zijlstra. 1977. Design and implementation of an efficient priority queue. *Math. Syst. Theory* 10 (1977), 99–127.
[102] Dan E. Willard. 1983. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Info. Process. Lett.* 17, 2 (1983), 81–84.
[103] Dan E. Willard. 2000. Examining computational geometry, van Emde Boas trees, and hashing from the perspective of the fusion tree. *SIAM J. Comput.* 29, 3 (2000), 1030–1049.
[104] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-index: A fully-dynamic compressed learned index with provable worst-case bounds. *Proc. of the VLDB Endowment* 13, 8 (2020), 1162–1175.

Received February 2019; revised May 2020; accepted June 2020