Predecessor on the Ultra-Wide Word RAM

Philip Bille
DTU Compute, Technical University of Denmark, Lyngby, Denmark

Inge Li Gørtz
DTU Compute, Technical University of Denmark, Lyngby, Denmark

Tord Stordalen
DTU Compute, Technical University of Denmark, Lyngby, Denmark

Abstract

We consider the predecessor problem on the ultra-wide word RAM model of computation, which extends the word RAM model with *ultrawords* consisting of w^2 bits [TAMC, 2015]. The model supports arithmetic and boolean operations on ultrawords, in addition to *scattered* memory operations that access or modify w (potentially non-contiguous) memory addresses simultaneously. The ultra-wide word RAM model captures (and idealizes) modern vector processor architectures.

Our main result is a simple, linear space data structure that supports predecessor in constant time and updates in amortized, expected constant time. This improves the space of the previous constant time solution that uses space in the order of the size of the universe. Our result is based on a new implementation of the classic x-fast trie data structure of Willard [Inform. Process. Lett. 17(2), 1983] combined with a new dictionary data structure that supports fast parallel lookups.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases Ultra-wide word RAM model, predecessor, word-level parallelism

Digital Object Identifier 10.4230/LIPIcs.SWAT.2022.18

Related Version *Full Version*: `https://arxiv.org/abs/2201.11550`

Funding *Philip Bille*: Danish Research Council grant DFF-8021-002498.
Inge Li Gørtz: Danish Research Council grant DFF-8021-002498.

Acknowledgements We thank the anonymous reviewers for their comments, which improved the presentation of the article.

1 Introduction

Let S be a set of n w-bit integers. The *predecessor problem* is to maintain S under the following operations.

- predecessor(x): return the largest $y \in S$ such that $y \leq x$.
- insert(x): add x to S.
- delete(x): remove x from S.

The predecessor problem is a fundamental and well-studied data structure problem, both from the perspective of upper bounds [2,5,7,8,23,31,33,37,38,39] and lower bounds [1,5,28,29, 31,32,35]. The problem has many applications, for instance integer sorting [2,3,23,25], string sorting [4,9,20], and string searching [6,8,10,11,13]. See Navarro and Rojas-Ledesma [30] for a recent survey.

On the word RAM model of computation, the complexity of the problem is well-understood with the following tight upper and lower bound on the time for operations given by Pătraşcu and Thorup [33].

$$\Theta\left(\max\left[1, \min\left\{\log_w n, \frac{\log \frac{w}{\log w}}{\log\left(\log \frac{w}{\log w} / \log \frac{\log n}{\log w}\right)}, \log \frac{\log(2^w - n)}{\log w}\right\}\right]\right). \tag{1}$$

From the upper bound perspective, the first branch matches dynamic fusion trees [23], the second branch is based on an extension of the techniques from Beame and Fich [5], and the last branch is based on an extension of dynamic van Emde Boas trees [38]. Note that the lower bound implies that we cannot support operations in constant time for general n and w. Hence, a natural question is if practical models of computation capturing modern hardware can allow us to overcome the superconstant lower bound.

One such model is the *RAM with byte overlap* (RAMBO) by Brodnik et al. [14]. This model extends the word RAM model by adding a set of special words that share bits; flipping a bit in one word will also affect all the other words that share that bit. The precise model is determined by the layout of the shared bits. It is feasible to make hardware based on this model, and prototypes have been built [27]. In the RAMBO model, Brodnik et al. [14] gave a predecessor data structure using constant time per operation with $O(2^w/w)$ space (counting both regular words and shared words). They also gave a randomized version of the solution that uses constant time with high probability and reduces the regular space to $O(n)$ (but still needs $\Omega(2^w/w)$ space for the shared words). In both cases, the total space is near-linear in the size of the universe.

More recently, Farzan et al. [21] introduced the *ultra-wide word RAM model* (UWRAM). The UWRAM extends the word RAM model by adding special *ultrawords* of w^2 bits. The model supports standard boolean and arithmetic operations on ultrawords, as well as *scattered* memory operations that access w words in memory in parallel. The UWRAM model captures (and idealizes) modern vector processing architectures [15, 34, 36] (see Section 2 for details of the model). Farzan et al. [21] showed how to simulate algorithms for the RAMBO model on the UWRAM at the cost of increasing the space by a polylogarithmic factor. Simulating the above RAMBO solution for the predecessor problem, they gave a solution to the predecessor problem on the UWRAM using worst case constant time for all operations and $O(w2^w)$ space.

1.1 Our Results

We revisit the predecessor problem on the UWRAM and show the following main result.

▶ **Theorem 1.** *Given a set of n w-bit integers, we can construct an $O(n)$ space data structure on a UWRAM which supports* predecessor *in constant time and* insert *and* delete *in amortized expected constant time.*

Compared to the previous result of Farzan et al. [21], Theorem 1 significantly reduces the space from $O(w2^w)$ to linear while maintaining constant time for operations (note that query time is worst-case, while updates are amortized expected).

A key component in our solution is a new dictionary data structure of independent interest that supports fast parallel lookups on the UWRAM. We define the problem as follows. Recall that an ultraword X consists of w^2 bits. We view X as divided into w words of w consecutive bits each, numbered from right to left starting from 0. The ith word in X is denoted $X\langle i\rangle$ (we discuss the model in detail in Section 2). Given a set S of n w-bit integers, the *w-parallel dictionary problem* is to maintain S under the following operations.

- pMember(X): return an ultraword I where $I\langle i\rangle = 1$ if $X\langle i\rangle \in S$ and $I\langle i\rangle = 0$ otherwise.
- insert(x): Add x to S.
- delete(x): Remove x from S.

Thus, pMember takes an ultraword X of w integers and returns an ultraword encoding which of these integers are in S. To the best of our knowledge, the w-parallel dictionary problem has not been studied before. We show the following result.

▶ **Theorem 2.** *Given a set of n w-bit integers, we can construct an $O(n+w)$-space data structure on a UWRAM which supports* pMember *in worst case constant time and* insert *and* delete *in amortized expected constant time.*

Note that the queries are worst-case constant time, while the updates are amortized expected constant time. The time bounds of Theorem 2 thus match the well-known dynamic perfect hashing structure of Dietzfelbinger et al. [19] (which is also the basis of our solution), except that the queries are parallel. The space is linear except for the additive w term, which is needed even for storing the input to the pMember query.

1.2 Techniques

Our results are achieved by novel and efficient parallel implementations of well-known sequential data structures.

Our parallel dictionary structure of Theorem 2 is based on the dynamic perfect hashing structure of Dietzfelbinger et al. [19]. This is a two-level data structure similar to the classic static perfect hashing structure of Fredman et al. [22]. At the first level, a universal hash function partitions the input into smaller subsets, each of which is then resolved at the second level using another universal hash function mapping the elements into sufficiently large tables. The structure supports (sequential) membership queries in worst-case constant time by evaluating the hash functions and navigating the structure accordingly. Updates are supported in amortized expected constant time by carefully rebuilding and rehashing the structure during execution. At any point in time the structure never uses more than $O(n)$ space. We show how to parallelize the evaluation of a universal hash function (the simple and practically efficient *multiply-shift* hash function). Then, using the scattered memory access operations, we show how to access the corresponding entries in the structure in parallel. Our technique requires only small changes to the structure of Dietzfelbinger et al. [19] and we can directly apply their update operations to our solution. Thus, we are able to parallelize the worst-case constant time sequential membership query while maintaining the amortized expected constant update time bound of Dietzfelbinger et al. [19], leading to the bounds of Theorem 2.

Our predecessor data structure of Theorem 1 is based on the *x-fast trie* of Willard [39] combined with our parallel dictionary structure of Theorem 2. The x-fast trie consists of the trie T of the binary representation of the input set. Also, at each level i, the structure stores a dictionary containing the length-i prefixes of the input set. In total, this uses $O(nw)$ space. The x-fast trie supports predecessor queries in $O(\log w)$ time by binary searching the levels (with the help of the dictionaries) to find the longest common prefix of the query and the input set. Though not designed for it, we can implement updates on the x-fast trie in $O(w)$ time by directly updating each level of the dictionary accordingly. Our new predecessor structure, which we call the *xtra-fast trie*, instead stores the compact trie of the binary representation of the input set (i.e., the trie where paths of nodes with a single child are merged into a single edge). We store a dictionary representing the prefixes (similar to in the x-fast trie) using our parallel dictionary structure of Theorem 2, but now only for the branching nodes in the compact trie. This reduces the space to $O(n)$. To support predecessor queries for an integer x, we generate all w prefixes of x and apply a parallel membership query on these in the dictionary. We show how to identify the longest match in parallel which in turn allows us to identify the predecessor. In total this takes worst-case constant time for the predecessor query. To handle updates, we show how to modify the trie efficiently using scattered memory access operations and a constant number of dictionary updates, leading to the expected amortized constant time bound of Theorem 1.

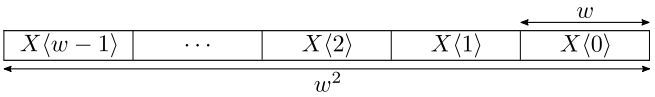

Figure 1 The layout of an ultraword X.

In our data structures we only need to store a constant number of ultrawords during the computation. This is important since modern vector processor architectures only have a limited number of ultraword registers.

1.3 Outline

In Section 2 we describe the UWRAM model of computation and some useful procedures. In Sections 3 and 4 we show how to do parallel hash function evaluation and w-parallel dictionaries, proving Theorem 2. Finally, in Section 5 we prove Theorem 1.

2 The Ultra-Wide Word RAM Model

The *word RAM* model of computation [24] consists of an unbounded memory of w-bit words and a standard instruction set including arithmetic, boolean, and bitwise operations (denoted "&", "|" and "∼" for *and*, *or* and *not*) and shifts (denoted "≫" and "≪") such as those available in standard programming languages (e.g., C). We make the standard assumption that we can store a pointer into the input in a single word and hence $w \geq \log n$, where n is the size of the input, and for simplicity we assume that w is even. We denote the address of x in memory as $\mathsf{addr}(x)$, and the address of an array is the address of its first index. The time complexity of a word RAM algorithm is the number of instructions and the space is the number of words stored by the algorithm.

The *ultra-wide word RAM* (UWRAM) model of computation [21] extends the word RAM model with special *ultrawords* of w^2 bits. As in [21], we distinguish between the *restricted UWRAM* that supports a minimal set of instructions on ultrawords consisting of addition, subtraction, shifts, and bitwise boolean operations, and the *multiplication UWRAM* that additionally supports multiplications. We extend the notation for bitwise operations and shifts to ultrawords. The UWRAM (both restricted and multiplication) also supports contiguous and scattered memory access operations, as described below. The time complexity is the number of instructions (on standard words or ultrawords) and the space complexity is the number of words used by the algorithms, where each ultraword is counted as w words. The UWRAM model captures (and idealizes) modern vector processing architectures [15, 34, 36]. See also Farzan et al. [21] for a detailed discussion of the applicability of the UWRAM model.

2.1 Instructions and Componentwise Operations

Recall that ultrawords consists of w^2 bits. We often view an ultraword X as divided into w words of w consecutive bits each, which we call the *components* of X. We number the components in X from right-to-left starting from 0 and use the notation $X\langle i\rangle$ to denote the ith word in X (see Figure 1). We will also use the notation $X = \langle x_{w-1}, \ldots, x_0\rangle$, denoting that $X\langle i\rangle = x_i$.

We define a number of useful componentwise operations on ultrawords that we will need for our algorithms in the following. Let X and Y be ultrawords. The *componentwise addition* of X and Y, denoted $X + Y$, is the ultraword Z such that $Z\langle i\rangle = X\langle i\rangle + Y\langle i\rangle \mod 2^w$.

We define *componentwise subtraction*, denoted $X - Y$, and *componentwise multiplication*, denoted XY, similarly. The *componentwise comparison* of X and Y is the ultraword Z such that $Z\langle i\rangle = 1$ if $X\langle i\rangle < Y\langle i\rangle$ and 0 otherwise. Given another ultraword I where each component is either 0 or 1, we define the *componentwise blend* of X, Y, and I to be the ultraword Z such that $Z\langle i\rangle = X\langle i\rangle$ if $I\langle i\rangle = 0$ and $Z\langle i\rangle = Y\langle i\rangle$ if $I\langle i\rangle = 1$.

Except for componentwise multiplication, all of the above componentwise operations can be implemented in constant time on the restricted UWRAM using standard word-level parallelism techniques [12, 24] (see the full version for details on blend). For our purposes, we will need componentwise multiplication as an instruction (for evaluating hash functions in parallel) and thus we include this in the instruction set of the UWRAM. This is the UWRAM model that we will use throughout the rest of the paper. Note that all of the componentwise operations are widely supported directly in modern vector processing architectures. For instance, a componentwise multiplication (e.g., the `vpmullw` operation) is defined in Intel's AVX2 vector extension [16].

We will need componentwise operations on components that are small constant multiples of w. In particular, we will need a $2w$*-bit componentwise multiplication* that multiplies $w/2$ components of w bits and returns the $w/2$ resulting components of $2w$ bits. Specifically, let $X = \langle 0, x_{w-2}, \ldots, 0, x_2, 0, x_0\rangle$ and $Y = \langle 0, y_{w-2}, \ldots, 0, y_2, 0, y_0\rangle$, i.e., X and Y store $w/2$ components aligned at the even positions. The $2w$-bit componentwise multiplication is the ultraword $Z = \langle z^+_{w-2}, z^-_{w-2}, \ldots, z^+_2, z^-_2, z^+_0, z^-_0\rangle$ where z^+_i and z^-_i is the leftmost and rightmost w bits, respectively, of the $2w$-bit product of x_i and y_i. We can implement $2w$-bit componentwise multiplication using standard techniques in constant time on the UWRAM. See the full version for details.

Finally, the UWRAM model supports the compress operation that, given X, returns the word that results from concatenating the rightmost bit of each component of X. We do not need the corresponding inverse spread operation, defined by Farzan et al. [21].

2.2 Memory Access

The UWRAM supports standard memory access operations that read or write a single word or a sequence of w contiguous words. More interestingly, the UWRAM also supports *scattered* access operations that access w memory locations (not necessarily contiguous) in parallel. Given an ultraword A containing w memory addresses, a *scattered read* loads the contents of the addresses into an ultraword X, such that $X\langle i\rangle$ contains the contents of memory location $A\langle i\rangle$. Given ultrawords X and A a *scattered write* sets the contents of memory location $A\langle i\rangle$ to be $X\langle i\rangle$. Scattered memory accesses captures the memory model used in IBM's *Cell* architecture [15]. They also appear (e.g., `vpgatherdd`) in Intel's AVX2 vector extension [16]. Scattered memory access operations were also proposed by Larsen and Pagh [26] in the context of the I/O model of computation. Note that while the addresses for scattered writes must be distinct, we can read simultaneously from the same address. We can use this to efficiently copy x into all w components of an ultraword X. To do so, create the ultraword $\langle 0, \ldots, 0\rangle$ by left-shifting any ultraword by w^2 bits, write x to address 0, and do a scattered read on $\langle 0, \ldots, 0\rangle$. We say that we *load* x into X.

3 Computing Multiply-Shift in Parallel

We show how to efficiently compute a universal hash function in parallel. The *multiply-shift* hashing scheme is a standard and practically efficient family of universal hash functions due to Dietzfelbinger et al. [18]. For some integer $1 \leq c \leq w$, define the class $H_c = \{h_a \mid 0 < a <$

2^w and a is odd$\}$ of hash functions where $h_a(x) = (ax \mod 2^w) \gg (w-c)$. Each function in H_c maps from w-bit to c-bit integers. The class H_c is *universal* in the sense that for any $x \neq y$ and for $h_a \in H_c$ selected uniformly at random, it holds that $P[h_a(x) = h_a(y)] \leq 2/2^c$.

We will show how to evaluate w such functions in constant time. Given $X\langle i\rangle = x_i$, $A\langle i\rangle = a_i$ and $C\langle i\rangle = 2^{c_i}$ where $h_i(x) = (a_i x \mod 2^w) \gg (w - c_i)$ the goal is to compute $H\langle i\rangle = h_i(x_i)$. To do so we first evaluate the functions in two rounds of $w/2$ functions each, and then combine the results.

Step 1: Evaluate the hash function on the even indices. We construct an ultraword H_{even} containing all the values of $h_i(x_i)$ at all even indices i. First construct the ultrawords

$$C' = \langle 0, 2^{c_{w-2}}, \ldots, 0, 2^{c_0}\rangle$$
$$T' = \langle 0, a_{w-2}x_{w-2} \mod 2^w, \ldots, 0, a_0 x_0 \mod 2^w\rangle.$$

To do so, we do componentwise multiplication of C with the constant $M = \langle 0, 1, \ldots, 0, 1\rangle$ and componentwise multiplications of A, X, and M. Then, we do a $2w$-bit multiplication of C' and T' and right shift the result by w. This produces the ultraword

$$H_{\text{even}} = \langle \star, (a_{w-2}x_{w-2} \mod 2^w) \gg (w - c_{w-2}), \ldots, \star, (a_0 x_0 \mod 2^w) \gg (w - c_0)\rangle$$

Thus, all even indices in H_{even} store the resulting hash values of the integers at the even indices in the input. We will not need the values in the odd indices (resulting from the $2w$-bit multiplication and the right shift) and therefore these are marked with a wildcard symbol $\star$.

Step 2: Evaluate the hash function on the odd indices. Symmetrically, we now construct the ultraword H_{odd} containing $h_i(x_i)$ at all odd indices i. To do so, repeat step 1 and modify the shifting to align the computation for the odd indices. More precisely, right shift X, C and A by w and repeat step 1, then left shift the result by w to align the results back to the odd positions. This produces the ultraword

$$H_{\text{odd}} = \langle (a_{w-1}x_{w-1} \mod 2^w) \ll c_{w-1}, \star, \ldots, (a_1 x_1 \mod 2^w) \ll c_1, \star\rangle$$

Step 3: Combine the results. Finally, we combine the results by blending H_{even} and H_{odd} using $I = \langle 1, \ldots, 1\rangle - M$, producing the ultraword H of the even indices of H_{even} and the odd indices of H_{odd}.

This takes constant time since componentwise multiplication, $2w$-bit multiplication, shifting, blending, loading 1 into $\langle 1, \ldots, 1\rangle$, and componentwise subtraction all run in constant time. Hence, we can evaluate each case of $w/2$ hash functions in constant time and combine the results in constant time. In summary, we have the following result.

▶ **Lemma 3.** *Given $X\langle i\rangle = x_i$, $A\langle i\rangle = a_i$, $C\langle i\rangle = 2^{c_i}$, and the constant $M = \langle 0, 1, \ldots, 0, 1\rangle$ we can evaluate each of the w multiply-shift hash functions $h_i(x) = (a_i x \mod 2^w) \gg (w - c_i)$ by computing the ultraword $H = \langle h_{w-1}(x_{w-1}), \ldots, h_0(x_0)\rangle$ in constant time on a UWRAM.*

4 The w-Parallel Dictionary

We now show how to construct the w-parallel dictionary of Theorem 2. To do so we use a dictionary by Dietzfelbinger et al. that implements a dynamic perfect hashing strategy [19]. Their dictionary already supports insert and delete in amortized expected constant time. Furthermore, it supports sequential member queries (i.e. "is $x \in S$") in worst case constant time. We will show that we can use scattered memory operations to run w member queries simultaneously, thus implementing pMember in constant time.

4.1 Dynamic Perfect Hashing

In this section we briefly describe the contents of the data structure of Dietzfelbinger et al. [19]. Note that we use the multiply-shift hashing scheme, while they use another class of universal hash functions. Multiply-shift satisfies all the necessary constraints and the analysis from [19] still works. It does however incur a multiplicative, constant space overhead for our arrays since the range of a multiply-shift function is a power of two.

The main idea of the data structure is as follows. Let S be a set of w-bit integers. Choose $h \in H_c$ and partition S into $2^c = \Theta(n)$ sets $S_0, \ldots, S_{2^c-1}$ where $S_i = \{x \mid x \in S \text{ and } h(x) = i\}$. Each set S_i is stored in a separate array using a hash function h_i. Dietzfelbinger et al. show how to implement the operations insert and delete such that they maintain that h_i has no collisions on S_i.

The data structure consists of the following.

- For each S_i, store an array T_i of size 2^{c_i}. Let $h_i(x) = (a_i x \mod 2^w) \gg (w - c_i)$. For each $x \in S_i$ let $T_i[h_i(x)] = x$, i.e. the position that x hashes to stores x. If there is no $x \in S_i$ that hashes to j, then $T_i[j] = 2^{w-1}$ if $j = 0$ and $T_i[j] = 0$ otherwise. We claim that $h_i(0)$ is always zero and $h_i(2^{w-1})$ is never zero, so it follows from this construction that $x \in S_i$ if and only if $T_i[h_i(x)] = x$. We have that $h_i(2^{w-1})$ is not zero because

$$h_i(2^{w-1}) \;=\; (a_i 2^{w-1} \mod 2^w) \gg (w - c_i) \;=\; 2^{w-1} \gg (w - c_i) \;\geq\; 1.$$

  The second step follows since a_i is odd; then $a_i 2^{w-1} = 2^{w-1} + (a_i - 1)2^{w-1}$, and the latter term is 0 modulo 2^w since $a_i - 1$ is even. The last step follows because $c_i \geq 1$.
- An array T of size 2^c. At index $T[i]$ we store the 5-tuple $(\mathsf{addr}(T_i), 2^{c_i}, a_i, \star, \star)$ where $\star$ are book-keeping values used by insert and delete. Note that 2^{c_i} and a_i encode h_i.
- The integers a and 2^c representing the top-level hash function $h(x) = (ax \mod 2^w) \gg (w - c)$, as well as $\mathsf{addr}(T)$.

It follows from this construction that $x \in S$ if and only if $T_i[h_i(x)] = x$ where $i = h(x)$. Dietzfelbinger et al. show that the data structure uses linear space, that member runs in worst-case constant time, and that insert and delete run in amortized expected constant time [19].

Extending the Data Structure. We extend this data structure by storing the constant $M = \langle 0, 1, \ldots, 0, 1, 0, 1\rangle$ from Section 3 used to evaluate multiply-shift functions in parallel. This increases the space of the data structure to $O(n + w)$. Note that linear space in w is needed even to store the input to a pMember query.

4.2 Parallel Queries

In this section, we begin by describing a single member query, before we show how to run w copies of the member query in parallel to support pMember. We compute $\mathsf{member}(x)$ as follows.

1. Using a and 2^c, compute $j = h(x)$.
2. Let $q = \mathsf{addr}(T) + 5j = \mathsf{addr}(T[j])$ (recall that each index in T stores five words). Read the values stored at q, $q + 1$ and $q + 2$ to get respectively $\mathsf{addr}(T_j)$, 2^{c_j} and a_j, the first three words stored at $T[j]$. Compute $k = h_j(x)$.
3. Check whether the value stored at $\mathsf{addr}(T_j) + k = \mathsf{addr}(T_j[k])$ is equal to x.

The parallel algorithm runs this algorithm for all w inputs simultaneously. Given $X = \langle x_{w-1}, \ldots, x_0\rangle$ we implement $\mathsf{pMember}(X)$ as follows. Each of the steps below executes the corresponding step above in parallel for each of the w inputs.

Step 1: Evaluate the top-level hash function. Load the two ultrawords $A = \langle a, \ldots, a\rangle$ and $C = \langle 2^c, \ldots, 2^c\rangle$. Compute the ultraword $J = \langle h(x_{w-1}), \ldots, h(x_0)\rangle$ using the multiply-shift algorithm of Lemma 3.

Step 2: Evaluate each of the second-level hash functions. Load $F = \langle 5, \ldots, 5\rangle$ and $P = \langle \mathsf{addr}(T), \ldots, \mathsf{addr}(T)\rangle$. Compute $Q = P + FJ$. Then $Q\langle i\rangle = \mathsf{addr}(T) + 5J\langle i\rangle = \mathsf{addr}(T[J\langle i\rangle])$. Do scattered reads of Q, $Q + \langle 1, \ldots, 1\rangle$, and $Q + \langle 2, \ldots, 2\rangle$ to produce the ultrawords P', C', and A'. We have that

$$P' = \langle \mathsf{addr}(T_{J\langle w-1\rangle}), \ldots, \mathsf{addr}(T_{J\langle 0\rangle})\rangle$$
$$C' = \langle 2^{c_{J\langle w-1\rangle}}, \ldots, 2^{c_{J\langle 0\rangle}}\rangle$$
$$A' = \langle a_{J\langle w-1\rangle}, \ldots, a_{J\langle 0\rangle}\rangle$$

Compute the ultraword $K = \langle h_{J\langle w-1\rangle}(x_{w-1}), \ldots, h_{J\langle 0\rangle}(x_0)\rangle$ using the multiply-shift algorithm of Lemma 3.

Step 3: Check whether the inputs are present in the dictionary. Do a scattered read of $P' + K$ and name the result R. Then $R\langle i\rangle = T_j[h_j(x_i)]$ where $j = h(x_i)$. Return the result I of componentwise equality between X and R. That is

$$I\langle i\rangle = \begin{cases} 1 & \text{if } X\langle i\rangle = R\langle i\rangle \\ 0 & \text{otherwise} \end{cases}$$

Evaluating the hash functions in steps 1 and 2 takes constant time according to Lemma 3. The remaining operations are scattered reads, loads and componentwise operations, all of which run in constant time. Since there is only a constant number of operations, pMember runs in constant time. This concludes the proof of Theorem 2.

4.3 Satellite Data

Suppose we associate some value $\mathsf{data}(x)$ with each $x \in S$. We extend the data structure to support the following operation, where $X = \langle x_{w-1}, \ldots, x_0\rangle$ as above.

- pRetrieve(X): returns a pair (I, D) where I is the result of pMember(X) and

$$D\langle i\rangle = \begin{cases} \mathsf{addr}(\mathsf{data}(x_i)) & \text{if } I\langle i\rangle = 1, \text{ i.e if } x_i \in S \\ \text{undefined} & \text{otherwise} \end{cases}$$

We return $\mathsf{addr}(\mathsf{data}(x))$ instead of $\mathsf{data}(x)$ since the data would not fit into an ultraword if $\mathsf{data}(x)$ requires more than one word to store.

We extend the data structure as follows to support pRetrieve. Store two words for each index in T_i. For each $x \in S_i$, the first word in $T_i[h_i(x)]$ stores x and the second stores $\mathsf{addr}(\mathsf{data}(x))$. The remaining entries store either 0 or 2^{w-1}, as above.

To do the retrieval, first compute $I = \mathsf{pMember}(X)$. However, in step 3, multiply K by $\langle 2, \ldots, 2\rangle$ before the scattered read since each index in T_i now stores two words. Also, add $\langle 1, \ldots, 1\rangle$ to $P' + \langle 2, \ldots 2\rangle K$ and do a scattered read to compute the ultraword D. The space of the data structure remains $O(n + w)$ (assuming that $\mathsf{data}(x)$ uses constant space), and pRetrieve runs in constant time.

5 The $x$tra-fast Trie

In this section we describe our data structure, the $x$tra-fast trie, which supports predecessor in worst case constant time and insert and delete in amortized expected constant time.

Below we assume that we have keys of $w-1$ bits each and give a solution that uses $O(n+w)$ space. At the end of this section we will reduce the space to $O(n)$ and extend the solution to w-bit keys, proving Theorem 1.

5.1 Data Structure

Consider the compacted trie T over the binary representation of the elements in S. For each node $v \in T$ define $\mathsf{str}(v)$ to be the bitstring encoded by the path from the root to v in T. Also let $\mathsf{min}(v)$ and $\mathsf{max}(v)$ be the smallest and largest leaves in the subtree of v, respectively. By $\mathsf{min}(v)$ and $\mathsf{max}(v)$ we refer both to a leaf and to the value the leaf represents.

For each edge $(u,v) \in T$, let $\mathsf{label}(u,v)$ be $\mathsf{str}(u)$ followed by the first bit on the edge (u,v). Define $\mathsf{key}(u,v)$ to be $\mathsf{label}(u,v)$ followed by a single 1-bit and $w - |\mathsf{label}(u,v)| - 1$ zeroes. Note that $|\mathsf{key}(u,v)| = w$ and that the keys of two distinct edges in T always differ. See Figure 2 for an example.

We define the *exit edge* for an integer x to be the edge in T where the match of x ends. In other words, it is the edge $(u,v) \in T$ such that $\mathsf{label}(u,v)$ is a prefix of x and $|\mathsf{label}(u,v)|$ is maximum. See Figure 2 for an example. It is possible that x has no exit edge if the root has fewer than two children.

Our data structure consists of the following:

- A sorted, doubly linked list L of the leaves of T, i.e., the elements of S.
- A dictionary D supporting parallel queries using Theorem 2. For each edge $(u,v) \in T$ we store an entry in D with the key $\mathsf{key}(u,v)$ and $\mathsf{data}(u,v) = (\mathsf{addr}(\mathsf{min}(v)), \mathsf{addr}(\mathsf{max}(v)))$. Here, $\mathsf{addr}(\mathsf{min}(v))$ and $\mathsf{addr}(\mathsf{max}(v))$ are the addresses to the corresponding elements in L, and we denote the addresses to $\mathsf{min}(v)$ and $\mathsf{max}(v)$ as the *min-* and *max-pointer* of (u,v).
- The two ultraword constants M' and H described in the next section.

Storing L and the ultraword constants takes $O(n+w)$ space combined. Since T is compacted there are $O(n)$ entries in D, so by Theorem 2 the dictionary also uses $O(n+w)$ space.

5.2 Predecessor Queries

The main idea of the predecessor query for x is to first find the exit edge of x by simultaneously searching for all prefixes of x in D. Then we use the min- and max-pointer of the exit edge to find the predecessor of x. If x has no exit edge, then the root does not have an outgoing edge matching the leftmost bit of x. If the leftmost bit of x is 1, the predecessor of x is the largest leaf in the left subtree of the root, and otherwise x has no predecessor. Assuming that x has an exit edge, the procedure has three steps.

Step 1: Compute all prefixes of x. Let $b_{w-2}b_{w-3}\cdots b_0$ be the binary representation of x of length $w-1$. We compute the ultraword

$$\overline{X} = \langle b_{w-2}b_{w-3}\cdots b_0 1 \ , \ b_{w-2}b_{w-3}\cdots b_1 10 \ , \ \ldots \ , \ 10\cdots 0\rangle.$$

That is, $\overline{X}\langle i\rangle$ contains the prefix of x of length i followed by a 1-bit and $w-i-1$ zeroes. Thus, for any edge $(u,v) \in T$ such that $\mathsf{label}(u,v)$ is the length-i prefix of x, we have $\overline{X}\langle i\rangle = \mathsf{key}(u,v)$. We compute $\overline{X}$ as follows.

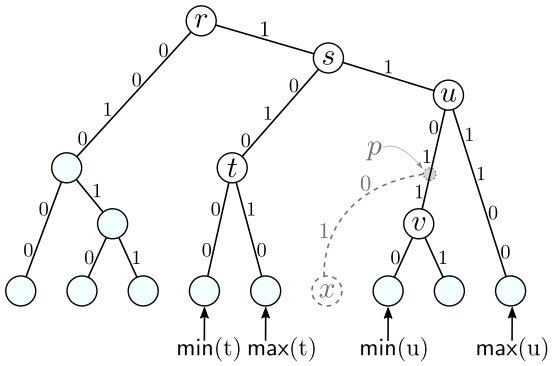

Figure 2 An $x$tra-fast trie for $S = \{001000,001010,001011,101000,101010,110110,110111,111100\}$. The dashed edge and nodes illustrate how the trie would change if $x = 110101$ were inserted. The exit edge for x is (u, v) since we match the bitstring 1101 but do not match the next 1 on (u, v). Similarly, the exit edge for 100100 is (s, t). We have that $\mathsf{key}(u, v) = \mathsf{label}(u, v)\underline{1000} = 1101\underline{1000}$ where the underlined part is what we append to the labels to disambiguate the keys. Similarly, $\mathsf{key}(r, s) = 1\underline{100000}$ and $\mathsf{key}(s, t) = 10\underline{10000}$. The dictionary entry of (s, u) has $\mathsf{key}(s, u) = 11\underline{10000}$, and the min- and max-pointer of (s, u) are $\mathsf{addr}(\mathsf{min}(u))$ and $\mathsf{addr}(\mathsf{max}(u))$. Similarly, the min-pointer of (r, s) is to $\mathsf{min}(s) = \mathsf{min}(t)$ and the max-pointer is to $\mathsf{max}(s) = \mathsf{max}(u)$. Note that if we insert x we would have to update the min-pointer of (s, u), since $x < \mathsf{min}(v)$. However, the min-pointer of (r, s) remains unchanged since $\mathsf{min}(t) < x$.

Let M' be the constant such that $M'\langle i\rangle$ consists of i consecutive 1-bits followed by $w - i$ consecutive 0-bits. Let H be the constant where the $(i+1)$th leftmost bit in $H\langle i\rangle$ is 1 and the remaining bits are zeroes. First load x into X such that $X = \langle x, x, \ldots, x\rangle$. Then compute $\overline{X} = (X \mathbin{\&} M') \mid H$.

Step 2: Find the exit edge (u, v) of x. First do $(I, P) = \mathsf{pRetrieve}(\overline{X})$ on D. Then compute $c = \mathsf{compress}(I)$ such that the ith rightmost bit in c is 1 if $I\langle i\rangle = 1$ and zero otherwise. Note that x has no exit edge if $c = 0$. Find the index k of the leftmost bit in c that is 1 (see [23]). Then $\overline{X}\langle k\rangle = \mathsf{key}(u, v)$ where (u, v) is the exit edge of x. Furthermore, the values stored at the addresses $P\langle k\rangle$ and $P\langle k\rangle + 1$ are the min- and max-pointers of (u, v), respectively.

Step 3: Find the predecessor of x. Use the min- and max-pointer of (u, v) found in step 2 to retrieve $\mathsf{min}(v)$ and $\mathsf{max}(v)$. If $x \geq \mathsf{max}(v)$ then return $\mathsf{max}(v)$, otherwise return the element immediately left of $\mathsf{min}(v)$ in L. Note that there might not be an element immediately left of $\mathsf{min}(v)$ if x is smaller than than everything in S, in which case x has no predecessor.

Since we search for all prefixes of x and take the edge corresponding to the longest prefix found, we find the exit edge (u, v) of x. If $x \in S$, then $x = v = \mathsf{max}(v)$ and we correctly return that x is the predecessor of itself. If $x \notin S$ then the path to where x would have been located if it were in T branches off (u, v) either to the left (if $x < \mathsf{min}(v)$) or right (if $x > \mathsf{max}(v)$). In the first case, $\mathsf{predecessor}(x)$ is the element located immediately left of $\mathsf{min}(v)$ in T, and in the second case $\mathsf{predecessor}(x)$ is $\mathsf{max}(v)$.

By Theorem 2 the parallel dictionary query in step 2 takes worst case constant time. Finding the leftmost bit that is 1 takes constant time on the word RAM [23]. The remaining operations are standard operations available in the model, so the procedure runs in constant time.

5.3 Insertions

The main idea of the insertion procedure is as follows. Since T is compacted, inserting a new leaf x will cause only a constant number of edges to be inserted and removed, so we can make these changes sequentially. Furthermore, some of the at most $w-1$ edges on the path from the root to x might have their min- or max-pointers changed, and we will update these edges in parallel.

Consider inserting $x = 110101$ in the trie in Figure 2. When x is inserted we add a new leaf for x, as well as a new node p at the location where the path to x branches off the exit edge (u, v) of x. This removes the edge (u, v), but adds the three new edges (u, p), (p, x) and (p, v). Furthermore, we must update the min-pointer of (s, u), because $\mathsf{min}(v)$ was replaced by x as the smallest leaf under u. On the other hand, we do not update the min-pointer of (r, s) because $\mathsf{min}(t)$ is smaller than x. Note that we do not explicitly store internal nodes and therefore do not add p anywhere in the data structure.

We now describe the insertion procedure. First we note that if x does not have an exit edge it is because the root does not have an outgoing edge which shares the same leftmost bit as x. This case is easily solved by adding an edge from the root to the new leaf x and adding x to either the start or end of L. We will now assume that x has an exit edge, and also that x branches off its exit edge to the left; the other case is symmetric.

Step 1: Find the predecessor of x. Do a predecessor query as described in Section 5.2, which determines

- The predecessor of x in L.
- The exit edge (u, v) of x, $\mathsf{label}(u, v)$ and $\mathsf{data}(u, v) = (\mathsf{addr}(\mathsf{min}(v)), \mathsf{addr}(\mathsf{max}(v)))$.
- The result (I, P) of $\mathsf{pRetrieve}(\overline{X})$ on D.

Step 2: Insert x in L. Insert x immediately to the right of its predecessor in L.

Step 3: Update edges. We insert (u, p), (p, x) and (p, v) and remove (u, v) from D. We find the labels of the three edges to insert as follows. We have that $\mathsf{label}(u, p) = \mathsf{label}(u, v)$ since (u, p) is the edge (u, v) shortened by adding the node p and since only the first character of the edge affects the label. By definition, $\mathsf{label}(p, x)$ and $\mathsf{label}(p, v)$ consist of $\mathsf{str}(p)$ with a zero and a one appended, respectively. We compute $\mathsf{str}(p)$ by finding the longest common prefix $\hat{p}$ of x and $\mathsf{min}(v)$. To do so, do bitwise XOR between x and $\mathsf{min}(v)$ and find the index k of the leftmost bit that is 1 in the result (see [23]). Now k indicates the leftmost bit where x and $\mathsf{min}(v)$ differ. To extract the longest common prefix compute $\hat{p} = x \;\&\; {\sim}((1 \ll (k+1)) - 1)$. Given the labels we can easily construct the keys for the edges.

We now construct the satellite data for the edges. Both the min- and max-pointer for (p, x) are $\mathsf{addr}(x)$ since x is a leaf. For (p, v) they are $\mathsf{addr}(\mathsf{min}(v))$ and $\mathsf{addr}(\mathsf{max}(v))$, which were determined during the predecessor query. Finally, the min-pointer for (u, p) is $\mathsf{addr}(x)$ and the max-pointer is $\mathsf{addr}(\mathsf{max}(v))$.

Step 4: Update min-pointers. We update the min-pointers for the edges on the path from the root to u that are incorrect after inserting x. Note that inserting x cannot invalidate any max-pointers since we assumed that x branched off its exit edge to the left. The edges that must be updated are exactly those that have a min-pointer to $\mathsf{min}(v)$, since x has replaced $\mathsf{min}(v)$ as the smallest leaf under u.

Consider the result (I, P) from the pRetrieve query. We begin by setting $I\langle k' \rangle = 0$ for the index k' corresponding to the exit edge (u, v) of x (we know k' from the predecessor query). The indices in I that now contain 1 indicate the edges on the path from the root to u.

Next we identify the edges that needs to be updated by creating I' where $I'\langle i \rangle = 1$ if and only if both $I\langle i \rangle = 1$ and what is stored at address $P\langle i \rangle$ is the address of $\mathsf{min}(v)$. To do so, first do a scattered read of P and store the result in M. Now M contains $\mathsf{addr}(\mathsf{min}(b))$ for each edge (a, b) on the path to u.[1] Note the value of $P\langle i \rangle$ is arbitrary if $I\langle i \rangle = 0$, i.e. if no edge has the length-i prefix of x as its label. Load $\mathsf{addr}(\mathsf{min}(v))$ into the ultraword V. Let E be the result of componentwise equality between M and V. Then $E\langle i \rangle = 1$ if and only if what is stored at address $P\langle i \rangle$ is $\mathsf{addr}(\mathsf{min}(v))$. Finally compute $I' = I \,\&\, E$.

Now we use P and I' to update the incorrect min-pointers. First, load the address of the node for x into U. Then compute B by blending M (the result of the scattered read of P) and U conditioned on I' such that

$$B\langle i \rangle = \begin{cases} M\langle i \rangle & \text{if } I'\langle i \rangle = 0 \quad \text{(i.e. the value already at the address } P\langle i \rangle\text{)} \\ U\langle i \rangle & \text{if } I'\langle i \rangle = 1 \quad \text{(i.e. the address of } x\text{)} \end{cases}$$

Finally, do a scattered write of B to the addresses in P. Hence, what is stored at the address $P\langle i \rangle$ remains the same if $I'\langle i \rangle = 0$ and is replaced by the address of x otherwise.

The predecessor query in step 1 takes constant time. The operations in step 2 and step 4 are all standard RAM or UWRAM operations, except for finding the leftmost 1-bit which takes constant time [23]. The dictionary updates in step 3 run in amortized expected constant time by Theorem 2. Since the rest of step 3 consists of standard operations, the running time for insertions is amortized expected constant.

5.4 Deletions

The deletion procedure is essentially the inverse of the insertion procedure. We assume that x is the left child of its parent p; the other case is symmetric.

Step 1: Find x. Do a predecessor query for x. Since $x \in S$, the predecessor of x is itself. This determines

- The position of x in L.
- The exit edge (p, x) for x, along with $\mathsf{label}(p, x)$. Since $x \in S$, this edge must end in the leaf for x.
- The result (I, P) of $\mathsf{pRetrieve}(\overline{X})$ on D.

Step 2: Update min-pointers. If p is the root (i.e. if $|\mathsf{label}(p, x)| = 1$) we remove the edge (p, x) from D and remove x from L which completes the deletion of x. Otherwise p is an internal node and must have another child which we denote by v. Consider the edges on the path to p. Any min-pointer to x should be replaced by the address of $\mathsf{min}(v)$, since $\mathsf{min}(v)$ is the successor of x and also in the subtree of all of these edges. We find $\mathsf{min}(v)$ in the node immediately right of x in L. As we did for insertions, replace any min-pointer that is an address of x by the address of $\mathsf{min}(v)$ in parallel using I and P.

[1] If x branched off to the right of its exit edge, we would do a scattered read of $P + \langle 1, \ldots, 1 \rangle$ to load the max-pointers instead of min-pointers.

Step 3: Delete edges. We delete (p, x) and (p, v) from D. Determine $\mathsf{label}(p, v)$ by flipping the last bit in $\mathsf{label}(p, x)$. Using the labels we easily find the keys. Note that we do not explicitly delete the edge (u, p) or insert the edge (u, v). These two edges share the same key, and the min-pointer of (u, p) was changed to the address of $\mathsf{min}(v)$ in step 2.

Step 4: Update L. Remove x from L.

Steps 1, 2 and 4 all take constant time (see Sections 5.2 and 5.3). The two deletions in step 3 take amortized constant time according to Theorem 2. The remainder of step 3 takes constant time, so deletions run in amortized expected constant time.

5.5 Reducing to Linear Space and Supporting w-bit Keys

Here, we reduce the space to $O(n)$ and show how to support w-bit keys, concluding the proof of Theorem 1.

The $O(w)$ term in the space bound above is due to the w-parallel dictionary D and $O(1)$ ultraword constants. To avoid this when $n = o(w)$, we will initially support predecessor, insert and delete using the *dynamic fusion tree* by Pătraşcu and Thorup [33] (based on the fusion tree by Fredman and Willard [23]), which uses linear space and supports all three operations in constant time for sets of size $w^{O(1)}$. Simultaneously, we build the ultraword constants we need over the course of $\Theta(w)$ insertions, maintaining linear space. When $n \geq w$, the constants have been built and we move all elements into the trie. If at any point $n \leq w/2$, we move all elements from the trie into a fusion tree and remove the trie and the ultraword constants, leaving us with linear space and $\Theta(w)$ insert operations in which to rebuild the constants. Updates still run in amortized expected constant time since we always do $\Omega(w)$ updates before we move $O(w)$ elements.

To extend the solution to work with w-bit keys, we partition the input set S into S_0 and S_1 where $S_i = \{s \mid s \in S \text{ and the leftmost bit of } s \text{ is } i\}$, and store an $x$tra-fast trie for each set. Suppose the leftmost bit of an integer x is i. An insert, delete or predecessor operation on x is performed on the data structure for S_i. Additionally, if $i = 1$ and the predecessor query on S_1 returns that x has no predecessor, we return the largest element in S_0, or report that x has no predecessor if S_0 is empty.

6 Conclusion and Open Problems

We have studied the predecessor problem on the UWRAM model of computation. We have given a linear space data structure that supports predecessor queries in worst case constant time and updates in amortized expected constant time.

Furthermore, we have shown how to implement a w-parallel dictionary on the UWRAM. The dictionary supports w simultaneous membership queries in worst case constant time and individual updates in amortized expected constant time.

We wonder if it is possible to achieve constant time with high probability for all operations in the predecessor problem. The limiting factor for our solution is the time for updates in the w-parallel dictionary. There are dictionaries that achieve constant time with high probability for all operations in the word RAM model, e.g. [17]. However, such dictionaries seem to require hash functions that are difficult to evaluate in parallel on the UWRAM. For instance, [17] uses the modulo operator, for which we cannot see an obvious way to make a component-wise version.

References

1 Miklós Ajtai. A lower bound for finding predecessors in Yao's cell probe model. *Comb.*, 8(3):235–247, 1988. doi:10.1007/BF02126797.

2 Arne Andersson. Faster deterministic sorting and searching in linear space. In *Proc. 37th FOCS*, pages 135–141, 1996. doi:10.1109/SFCS.1996.548472.

3 Arne Andersson, Torben Hagerup, Stefan Nilsson, and Rajeev Raman. Sorting in linear time? *J. Comput. Syst. Sci.*, 57(1):74–93, 1998. doi:10.1006/jcss.1998.1580.

4 Lars Arge, Paolo Ferragina, Roberto Grossi, and Jeffrey Scott Vitter. On sorting strings in external memory (extended abstract). In *Proc. 29th STOC*, pages 540–548, 1997. doi:10.1145/258533.258647.

5 Paul Beame and Faith E. Fich. Optimal bounds for the predecessor problem and related problems. *J. Comput. Syst. Sci.*, 65(1):38–72, 2002. doi:10.1006/jcss.2002.1822.

6 Djamal Belazzougui. Worst-case efficient single and multiple string matching on packed texts in the word-RAM model. *J. Discrete Algorithms*, 14:91–106, 2012. doi:10.1016/j.jda.2011.12.011.

7 Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. Monotone minimal perfect hashing: searching a sorted table with $O(1)$ accesses. In *Proc. 20th SODA*, pages 785–794, 2009. URL: http://dl.acm.org/citation.cfm?id=1496770.1496856.

8 Djamal Belazzougui, Paolo Boldi, and Sebastiano Vigna. Dynamic z-fast tries. In *Proc. 17th SPIRE*, pages 159–172, 2010. doi:10.1007/978-3-642-16321-0_15.

9 Michael A. Bender, Martin Farach-Colton, and Bradley C. Kuszmaul. Cache-oblivious string B-trees. In *Proc. 25th PODS*, pages 233–242, 2006. doi:10.1145/1142351.1142385.

10 Philip Bille, Mikko Berggren Ettienne, Inge Li Gørtz, and Hjalte Wedel Vildhøj. Time-space trade-offs for Lempel-Ziv compressed indexing. *Theor. Comput. Sci.*, 713:66–77, 2018. doi:10.1016/j.tcs.2017.12.021.

11 Philip Bille, Inge Li Gørtz, and Frederik Rye Skjoldjensen. Deterministic indexing for packed strings. In *Proc. 28th CPM*, pages 6:1–6:11, 2017. doi:10.4230/LIPIcs.CPM.2017.6.

12 Philip Bille, Inge Li Gørtz, and Frederik Rye Skjoldjensen. Partial sums on the ultra-wide word RAM. *Theor. Comput. Sci.*, 905:99–105, 2022. Announced at TAMC 2020. doi:10.1016/j.tcs.2022.01.002.

13 Philip Bille, Gad M. Landau, Rajeev Raman, Kunihiko Sadakane, Srinivasa Rao Satti, and Oren Weimann. Random access to grammar-compressed strings and trees. *SIAM J. Comput.*, 44(3):513–539, 2015. doi:10.1137/130936889.

14 Andrej Brodnik, Svante Carlsson, Michael L. Fredman, Johan Karlsson, and J. Ian Munro. Worst case constant time priority queue. *J. Syst. Softw.*, 78(3):249–256, 2005. doi:10.1016/j.jss.2004.09.002.

15 Thomas Chen, Ram Raghavan, Jason N. Dale, and Eiji Iwata. Cell Broadband engine architecture and its first implementation - A performance view. *IBM J. Res. Dev.*, 51(5):559–572, 2007. doi:10.1147/rd.515.0559.

16 Intel Corporation. Intel® advanced vector extensions programming reference. *Intel Corporation*, 2011.

17 Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In *Proc. 17th ICALP*, pages 6–19, 1990. doi:10.1007/BFb0032018.

18 Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. A reliable randomized algorithm for the closest-pair problem. *J. Algorithms*, 25(1):19–51, 1997. doi:10.1006/jagm.1997.0873.

19 Martin Dietzfelbinger, Anna R. Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert Endre Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. Comput.*, 23(4):738–761, 1994. doi:10.1137/S0097539791194094.

20 Martin Farach. Optimal suffix tree construction with large alphabets. In *Proc. 38th FOCS*, pages 137–143, 1997. doi:10.1109/SFCS.1997.646102.

21 Arash Farzan, Alejandro López-Ortiz, Patrick K. Nicholson, and Alejandro Salinger. Algorithms in the ultra-wide word model. In *Proc. 12th TAMC*, pages 335–346, 2015. doi:10.1007/978-3-319-17142-5_29.

22 Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. ACM*, 31(3):538–544, 1984. doi:10.1145/828.1884.

23 Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.*, 47(3):424–436, 1993. doi:10.1016/0022-0000(93)90040-4.

24 Torben Hagerup. Sorting and searching on the word RAM. In *Proc. 15th STACS*, pages 366–398, 1998. doi:10.1007/BFb0028575.

25 Yijie Han. Deterministic sorting in $O(n \log\log n)$ time and linear space. *J. Algorithms*, 50(1):96–105, 2004. doi:10.1016/j.jalgor.2003.09.001.

26 Kasper Green Larsen and Rasmus Pagh. I/O-efficient data structures for colored range and prefix reporting. In *Proc. 23rd SODA*, pages 583–592, 2012. doi:10.1137/1.9781611973099.49.

27 R. Leben, M. Miletic, M. Špegel, A. Torst, A. Brodnik, and K. Karlsson. Design of high performance memory module on PC100. In *Proc. Electrotechnical and Computer Science Conference (ERK)*, pages 75–78, 1999.

28 Peter Bro Miltersen. Lower bounds for union-split-find related problems on random access machines. In *Proc. 26th STOC*, pages 625–634, 1994. doi:10.1145/195058.195415.

29 Peter Bro Miltersen, Noam Nisan, Shmuel Safra, and Avi Wigderson. On data structures and asymmetric communication complexity. *J. Comput. Syst. Sci.*, 57(1):37–49, 1998. doi:10.1006/jcss.1998.1577.

30 Gonzalo Navarro and Javiel Rojas-Ledesma. Predecessor search. *ACM Comput. Surv.*, 53(5):105:1–105:35, 2020. doi:10.1145/3409371.

31 Mihai Pătraşcu and Mikkel Thorup. Time-space trade-offs for predecessor search. In *Proc. 38th STOC*, pages 232–240, 2006. doi:10.1145/1132516.1132551.

32 Mihai Pătraşcu and Mikkel Thorup. Randomization does not help searching predecessors. In *Proc. 18th SODA*, pages 555–564, 2007. URL: http://dl.acm.org/citation.cfm?id=1283383.1283443.

33 Mihai Pătraşcu and Mikkel Thorup. Dynamic integer sets with optimal rank, select, and predecessor search. In *Proc. 55th FOCS*, pages 166–175, 2014. doi:10.1109/FOCS.2014.26.

34 James Reinders. Intel® AVX-512 instructions. Intel® Corporation, 2013.

35 Pranab Sen and Srinivasan Venkatesh. Lower bounds for predecessor searching in the cell probe model. *J. Comput. Syst. Sci.*, 74(3):364–385, 2008. doi:10.1016/j.jcss.2007.06.016.

36 Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanaël Prémillieu, Alastair Reid, Alejandro Rico, and Paul Walker. The ARM scalable vector extension. *IEEE Micro*, 37(2):26–39, 2017. doi:10.1109/MM.2017.35.

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

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

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