Dynamic Integer Sets with Optimal Rank, Select, and Predecessor Search

Mihai Pătraşcu
Passed away 2012

Mikkel Thorup
Department of Computer Science
University of Copenhagen
Copenhagen, Denmark
Email: mikkel2thorup@gmail.com

***Abstract*—We present a data structure representing a dynamic set S of w-bit integers on a w-bit word RAM. With $|S| = n$ and $w \geq \log n$ and space $O(n)$, we support the following standard operations in $O(\log n/ \log w)$ time:**

- **insert**(x) **sets** $S = S \cup \{x\}$.
- **delete**(x) **sets** $S = S \setminus \{x\}$.
- **predecessor**(x) **returns** $\max\{y \in S \mid y < x\}$.
- **successor**(x) **returns** $\min\{y \in S \mid y \geq x\}$.
- **rank**(x) **returns** $\#\{y \in S \mid y < x\}$.
- **select**(i) **returns** $y \in S$ **with** $\text{rank}(y) = i$**, if any.**

Our $O(\log n/ \log w)$ bound is optimal for dynamic rank and select, matching a lower bound of Fredman and Saks [STOC'89]. When the word length is large, our time bound is also optimal for dynamic predecessor, matching a static lower bound of Beame and Fich [STOC'99] whenever $\log n/ \log w = O(\log w/ \log \log w)$.

Technically, the most interesting aspect of our data structure is that it supports all the above operations in constant time for sets of size $n = w^{O(1)}$. This resolves a main open problem of Ajtai, Komlos, and Fredman [FOCS'83]. Ajtai et al. presented such a data structure in Yao's abstract cell-probe model with w-bit cells/words, but pointed out that the functions used could not be implemented. As a partial solution to the problem, Fredman and Willard [STOC'90] introduced a *fusion node* that could handle queries in constant time, but used polynomial time on the updates. We call our small set data structure a *dynamic fusion node* as it does both queries and updates in constant time.

***Keywords*-dynamic data structures; integer data structures;**

I. Introduction

We consider the problem of representing a dynamic set S of integer keys so that we can efficiently support the following standard operations:

- insert(x) sets $S = S \cup \{x\}$.
- delete(x) sets $S = S \setminus \{x\}$.
- member(x) returns $[x \in S]$.
- predecessor(x) returns $\max\{y \in S \mid y < x\}$.
- successor(x) returns $\min\{y \in S \mid y \geq x\}$. We could also have demanded $y > x$, but then the connection to rank and select below would be less elegant.
- rank(x) returns $\#\{y \in S \mid y < x\}$.
- select(i) returns $y \in S$ with rank$(y) = i$, if any. Then predecessor$(x) =$ select(rank$(x) - 1$) and successor$(x) =$ select(rank(x)).

Our main result is a deterministic linear space data structure supporting all the above operations in $O(\log n/ \log w)$ time where $n = |S|$ is the set size and w is the word length. This is on the word RAM which models what can be implemented in a programming language such as C [1] which has been used for fast portable code since 1978. Word operations take constant time. The word size w, measured in bits, is a unifying parameter of the model. All integers considered are assumed to fit in a word, and with $|S| = n$, we assume $w \geq \log n$ so that we can at least index the elements in S. The random access memory of the word RAM implies that we can allocate tables or arrays of words, accessing entries in constant time using indices that may be computed from keys. This feature is used in many classic algorithms, e.g., radix sort [2] and hash tables [3].

A unifying word size was not part of the original RAM model [4]. However, real CPUs are tuned to work on words and registers of a certain size w (sometimes two sizes, e.g., 32 and 64 bit registers). Accessing smaller units, e.g., bits, is more costly as they have to be extracted from words. On the other hand, with unlimited word size, we can use word operations to encode a massively parallel vector operations unless we make some artificial restrictions on the available operations [5]. The limited word size is equivalent to a "polynomial universe" restriction where all integers used are bounded by a polynomial in the problem size and sum of the integers in the input. Restrictions of this type are present in Yao's cell-probe model [6] and in Kirkpatrick and Reisch's integer sorting [7]. Fredman and Willard [8] made the word length explicit in their seminal $O(n \log n/ \log \log n) = o(n \log n)$ sorting algorithm.

Our $O(\log n/ \log w)$ bound is optimal for dynamic rank and select, matching a lower bound of Fredman and Saks [9][1]. The lower bound is in Yao's cell-probe model [6] where we only pay for probing cells/words in memory. The lower bound is for the query time and holds even with polylogarithmic update times. Conceivably, one could get the same query time with constant time updates. Our optimality is for the maximal operation time including both queries and updates.

When the word length is large, our time bound is also optimal for dynamic predecessor, matching a cell-probe lower bound of Beame and Fich [10] whenever $\log n/ \log w = O(\log w/ \log \log w)$. This lower bound is for the query time in the static case with any polynomial space representation of the set S. Contrasting rank and select, predecessor admits faster solutions when the word length is small, e.g., van Emde Boas' [11] $O(\log w)$ time bound. The predecessor bounds for smaller word lengths are essentially understood if

[1]For rank, see Theorem 3 and remark after Theorem 4 in [9]. The lower bound for select follows by another simple reduction to be presented here.

we allow randomization, and then our new bound completes the picture for dynamic predecessor search. With key length $\ell \leq w$, we will get that the optimal expected maximal operation time for dynamic predecessor (no rank or select) is within a constant factor of

$$\max\{1, \min \begin{cases} \frac{\log n}{\log w} \\ \frac{\log \frac{\ell}{\log w}}{\log\left(\log \frac{\ell}{\log w} \middle/ \log \frac{\log n}{\log w}\right)} \\ \log \frac{\log(2^\ell - n)}{\log w} \end{cases}$$

Our clean $O(\log n / \log w)$ bound for dynamic predecessor search should be compared with Fredman and Willard's [8] bound of $O(\log n / \log w + \sqrt{\log n})$[2]. For larger words, it was improved to $O(\log n / \log w + \log \log n)$ by Andersson and Thorup [13], but the $\log \log n$ term still prevents constant time when $\log n = O(\log w)$.

Technically, the most interesting aspect of our data structure is that it operates in constant time for sets of size $n = w^{O(1)}$. This resolves a main open problem of Ajtai, Komlos, and Fredman [14]. Ajtai et al. presented such a data structure in Yao's abstract cell-probe model with w-bit cells/words, but pointed out that the functions used could not be implemented. As a partial solution to the problem, Fredman and Willard [8] introduced a *fusion node* that could handle queries in constant time, but used polynomial time on the updates. Despite inefficient updates, this lead them to the first sub-logarithmic bounds for dynamic predecessor searching mentioned above. We call our data structure for small sets a *dynamic fusion node* as it does both queries and updates in constant time.

Fredman and Willard [15] later introduced atomic heaps that using tables of size s can handle all operations in constant time for sets of size $(\log s)^{O(1)}$. Our focus is on understanding the asymptotic impact of a large word length. We only use space linear in the number of stored keys, but our capacity for sets of size $w^{O(1)}$ with constant operation time is the largest possible regardless of the available space. Such a large set size is new even in the simple case of a deterministic dynamic dictionary with just membership and updates in constant time.

Like Fredman and Willard [8], [15], we do use multiplication. Thorup [16] has proved that this is necessary, even if we just want to support membership in constant time for sets of size $\Omega(\log n)$ using standard instructions. However, as in [17] for the classic static fusion nodes, if we allow self-defined non-standard operations, then we can implement our algorithms using AC^0 operations only. The original cell-probe solution from [14] does not seem to have an AC^0 implementation even if we allow non-standard operations.

We emphasize that our new dynamic fusion nodes generally both simplify and improve the application of fusion nodes in dynamic settings, most notably in the original fusion trees [8] which with the original fusion nodes had to switch to a different technique towards the bottom of the trees (c.f. Section IV), yet never got optimal results for large word-lengths. Fusion nodes are useful in many different settings as illustrated by Willard [18] for computational geometry.

Contents and techniques: Our new dynamic fusion node takes starting point in ideas and techniques from [14], [8] bringing them together in a tighter solution that for sets of size up to $k = w^{1/4}$ supports all operations in constant time using only standard instructions. In Section II we review the previous techniques, emphasizing the parts and ideas that we will reuse, but also pointing to their limitations. In Section III, we describe the new dynamic fusion nodes, constituting our core technical contribution. An important idea is that we use matching with don't cares to code the search in a compressed trie. In Section IV, we show how to use our dynamic fusion nodes in a fusion tree for arbitrary set size n, supporting all operations in time $O(\log n / \log k) = O(\log n / \log w)$. This is $O(1)$ for $n = w^{O(1)}$. Because our fusion nodes are dynamic, we avoid the dynamic binary search tree at the bottom of the original fusion trees [8]. Our fusion trees are augmented to handle rank and select, using techniques from [19], [20]. In Section V we describe in more details how we get matching lower bounds from [9], [10]. Finally, in Section VI, we complete the picture for randomized dynamic predecessor search with smaller key lengths using the techniques from [12].

II. Notation and review of previous techniques

We let lg denote $\log_2$ and define $[m] = \{0, ..., m-1\}$. Integers are represented in w-bit words with the least significant bit the to the right. We shall use `0` and `1` to denote the bits 0 and 1, contrasting the numbers 0 and 1 that in words are padded with $w-1$ leading `0`s. When working with w-bit integers, we will use the standard arithmetic operations $+$, $-$, and $\times$, all working modulo 2^w. We will use the standard bit-wise Boolean operations: $\wedge$ (and), $\vee$ (or), $\oplus$ (exclusive-or), and $\neg$ (bit-negation). We also have left shift $\ll$ and right shift $\gg$.

As is standard in assignments, e.g., in C [1], if we assign an ℓ_0-bit number x to a ℓ_1-bit number y and $\ell_0 < \ell_1$, then $\ell_1 - \ell_0$ leading `0`s are added. If $\ell_0 > \ell_1$, the $\ell_0 - \ell_1$ leading bits of x are discarded.

Fields of words: Often we view words as divided into fields of some length f. We then use $x\langle i \rangle_f$ to denote the ith field, starting from the right with $x\langle 0 \rangle_f$ the right most field. Thus x represents the integer $\sum_{i=0}^{w-1} 2^i x\langle i \rangle_1$. Note that fields can easily be masked out using regular instructions, e.g.,

$$x\langle i \rangle_f = (x \gg (i \times f)) \wedge ((1 \ll f) - 1).$$

A field assignment like $x\langle i \rangle_f = y$ is implemented as

$$x = (x \wedge \neg m) \vee ((y \ll (i \times f)) \wedge m) \text{ where}$$
$$m = ((1 \ll f) - 1) \ll (i \times f)$$

[2][8] describe it as $O(\log n / \log b + \log b)$ for any $b \leq w^{1/6}$, from which they get $O(\log n / \log \log n)$ since $w \geq \log n$.

Similarly, in constant time, we can mask out intervals of fields, using

$$x\langle i..j\rangle_f = (x \gg (i \times f)) \wedge ((1 \ll ((j-i) \times f)) - 1),$$
$$x\langle i..*\rangle_f = (x \gg (i \times f)).$$

For two-dimensional divisions of words into fields, we use the notation

$$x\langle i, j\rangle_{g \times f} = x\langle i \times g + j\rangle_f.$$

Finding the most and least significant bits in constant time: We have an operation msb(x) that for an integer x computes the index of its most significant set bit. Fredman and Willard [8] showed how to implement this in constant time using multiplication, but msb can also be implemented very efficiently by assigning x to a floating point number and extract the exponent. A theoretical advantage to using the conversion to floating point numbers is that we avoid the universal constants depending on w used in [8].

Using msb, we can also easily find the least significant bit of x as lsb$(x) = $ msb$((x-1) \oplus x)$.

Small sets of size k: Our goal is to maintain a dynamic set of some size $k = w^{\Theta(1)}$. Ajtai et al. [14] used $k = w/\log w$ whereas Fredman and Willard [8] used $k = w^{1/6}$. We will ourselves use $k = w^{1/4}$. The exact value makes no theoretical difference, as our overall bound is $O(\log n/\log k) = O(\log n/\log w)$, which is $O(1)$ for $k = w^{\Omega(1)}$ and $n = w^{O(1)}$.

For simplicity, we assume below that k is fixed, and we define various constants based on k, e.g., $(0^{k-1}1^k)^k$. However, using doubling, our constants can all be computed in $O(\log k)$ time. We let k be a power of two, that we double as the sets grow larger. Then the cost of computing the constants in the back ground is negligible.

Indexing: We will store our key set S in an unsorted array KEY with room for k w-bit numbers. We will also maintain an array $INDEX$ of $\lceil \lg k \rceil$-bit indices so that $INDEX\langle i\rangle_{\lceil \lg k\rceil}$ is the index in KEY of the key in S of rank i in the sorted order. An important point here is that $INDEX$ with its $k\lceil \lg k\rceil$ bits fits in a single word. Selection is now trivially implemented as

$$\text{select}(i) = KEY\left[INDEX\langle i\rangle_{\lceil \lg k\rceil}\right].$$

When looking for the predecessor of a query key x, we will just look for its rank i and return select(i).

Consider the insertion of a new key x, supposing that we have found its rank i. To support finding a free slot in KEY, we maintain a bit map $bKEY$ over the used slots, and use $j = $ msb$(bKEY)$ as the first free slot. To insert x, we set $KEY[j] = x$ and $bKEY\langle j\rangle_1 = 0$. To update $INDEX$, we set $INDEX\langle i+1..k\rangle_{\lceil \lg k\rceil} = INDEX\langle i..k-1\rangle_{\lceil \lg k\rceil}$ and $INDEX\langle i\rangle_{\lceil \lg k\rceil} = j$. Deleting a key is just reversing the above.

Binary trie: A binary trie [21] for w-bit keys is a binary tree where the children of an internal node are labeled `0` and `1`, and where all leaves are at depth w. A leaf corresponds to the key represented by the bits along the path to the root, the bit closest to the leaf being the least significant. Let $T(S)$ be the trie of a key set S. We define the level $\ell(u)$ of an internal trie node u to be the height of its children. To find the predecessor of a key x in S we follow the path from the root of $T(S)$, matching the bits of x. When we get to a node u in the search for x, we pick the child labeled $x\langle \ell(u)\rangle_1$. If $x \notin S$, then at some stage we get to a node u with a single child v where $x\langle \ell(u)\rangle_1$ is opposite the bit of v. If $x\langle \ell(u)\rangle_1 = 1$, the predecessor of x in S the largest key below v. Conversely, if $x\langle \ell(u)\rangle_1 = 0$, the successor of x in S is the smallest key below v.

Compressed binary trie: In a compressed binary trie, or Patricia trie [22], we take the trie and short cut all paths of degree-1 nodes so that we only have degree-2 branch nodes and leaves, hence at most $2k-1$ nodes. This shortcutting does not change the level ℓ of the remaining nodes.

To search a key x in the compressed trie $T^c(S)$, we start at the root. When we are at node u, as in the trie, we go to the child labeled $x\langle \ell(u)\rangle_1$. Since all interior nodes have two children, this search always finds a unique leaf corresponding to some key $y \in S$, and we call y the match of x.

The standard observation about compressed tries is that if x matches $y \in S$, then no key $z \in S$ can have a longer common prefix with x than y. To see this, first note that if $x \in S$, then the search must end in x, so we may assume that x matches some $y \neq x$. Let j be the most significant bit where x and y differ, that is, $j = $ msb$(x \oplus y)$. If z had a longer common prefix with x then it would branch from y at level j, and then the search for x should have followed the branch towards z instead of the branch to y. The same observation implies that there was no branch node on level j above the leaf of y. Let v be the first node below level j on the search path to y. As for the regular trie we conclude that if $x\langle \ell(u)\rangle_1 = 1$, the predecessor of x in S the largest key below v. Conversely, if $x\langle \ell(u)\rangle_1 = 0$, the successor of x in S is the smallest key below v.

To insert the above x, we would just insert a branch node u on level j above y. The parent of u is the previous parent of v. A new leaf corresponding to x is the $x\langle j\rangle_1$-child of u while v is the other child.

Ajtai et al.'s cell probe solution: We can now easily describe the cell probe solution of Ajtai et al. [14]. We are going to reuse the parts that can be implemented on a word RAM, that is, the parts that only need standard word operations.

As described previously, we assume that our set S is stored in an unordered array KEY of k words plus a single word $INDEX$ such that $KEY[INDEX\langle i\rangle_{\lceil \lg k\rceil}]$ is the key of rank i in S.

We represent the compressed trie in an array T^c with $k-1$ entries representing the internal nodes, the first entry being the root. Each entry needs its level in $[w]$ plus an index in $[k]$ to each child, or nil if the child is a leaf. We thus need $O(\log w)$ bits per node entry, or $O(k \log w) = o(w)$ bits for the whole array T^c describing the compressed trie. The ordering of children induces an ordering of the leaves, and leaf i, corresponds to key $KEY[INDEX\langle i\rangle_{\lceil \lg k\rceil}]$.

In the cell-probe model, we are free to define arbitrary word operation involving a constant number of words, including new operations on words representing compressed tries. We define a new word operation $TrieSearch(x, T^c)$ that given a key x and a correct representation of a compressed trie T^c, returns the index i of the leaf matching x in T^c. We then look up $y = KEY[INDEX\langle i\rangle_{\lceil \lg k\rceil}]$, and compute $j = \text{msb}(x \oplus y)$. A second new word operation $TriePred(x, T^c, j)$ returns the rank r of x in S, assuming that x branches of from T^c at level j.

To insert a key x in the above data structure, we first compute the rank r of x in S and then we insert x in KEY and $INDEX$, as described previously. A third new word operation $TrieInsert(T^c, x, j, r)$ changes T^c, inserting a new leaf at rank r, adding the appropriate branch node on level j with child $x\langle j\rangle_1$ being the new leaf.

Summing up, in the cell probe model, it is easy to support all our operations on a small dynamic set of integers, exploiting that a compressed trie can be coded in a word, hence that we can navigate a compressed trie using specially defined word operations. Ajtai et al. [14] write "*Open Question.* We conclude by stating the following problem. Our query algorithm is not very realistic, as searching through a trie requires more than constant time in actuality".

Fredman and Willard's static fusion nodes with compressed keys: Fredman and Willard [8] addressed the above problem, getting the searching in to constant time using only standard instructions, but with no efficient updates. Their so-called *fusion node* supports constant time predecessor searches among up to $w^{1/6}$ keys. However, with k keys, it takes $O(k^4)$ time for them to construct a fusion node.

Fredman and Willard proved the following lemma that we shall use repeatedly:

Lemma 2.1: Let $mb \leq w$. If we are given a b-bit number x and a word A with m b-bit numbers stored in sorted order, that is, $A\langle 0\rangle_b < A\langle 1\rangle_b < \cdots < A\langle m-1\rangle_b$, then in constant time, we can find the rank of x in A, denoted $\text{rank}(x, A)$. ∎

The basic idea behind the lemma is to first use multiplication to create x^m consisting of m copies of x; then use a subtraction to code a coordinate-wise comparison $A\langle i\rangle_b < x$ for every i, and finally use msb to find the largest such i.

Next, they note that for the ordering of the keys in S, it suffices to consider bits in positions j such that for some $y, z \in S$, $j = \text{msb}(y \oplus z)$. These are exactly the positions where we have branch node somewhere in the trie. Let $c_0 < .. < c_{\ell-1}$ be these significant positions. Then $\ell < k$.

After computing some variables based on S, they show that in constant time, they can compress a key x so as to (essentially) only contain the significant positions (plus some irrelevant `0`s). Their compressed key $\hat{x}$ is of length $b \leq k^3$. Let $\hat{S}$ denote a sorted array with the compressed versions of the keys from S. With $k^4 \leq w$, the array fits in a single word, and they can therefore compute $i = \text{rank}(\hat{x}, \hat{S})$ in constant time. Then $\hat{S}\langle i\rangle_b$ is the predecessor of $\hat{x}$ in $\hat{S}$, and $\hat{S}\langle i+1\rangle_b$ is the successor. One of these, say $\hat{y} = \hat{S}\langle i\rangle_b$, has the longest common prefix with $\hat{x}$, and they argue that then the original key $y = KEY\left[INDEX\langle i\rangle_{\lceil \lg k\rceil}\right]$ must also have the longest common prefix with x in S.

Next, they compute $j = \text{msb}(x \oplus y)$ and see where it fits between the significant bits, computing $h = \text{rank}(j, (c_0, ..., c_\ell))$, that is, $c_h < j \leq c_{h+1}$. Referring to the compressed trie representation, i and h tells us exactly in which shortcut edge (u, v), the key x branches out. The predecessor of x is the largest key below y if $x\langle j\rangle_1 = 1$; otherwise, the successor is the smallest key below y. The important point here is that they can create a table $RANK$ that for each value of $i \in [k]$, $h \in [k]$, and $x\langle j\rangle_1 \in [2]$, returns the rank $r \in [k]$ of x in S. From the rank we get the predecessor $KEY\left[INDEX\langle r\rangle_{\lceil \lg k\rceil}\right]$, all in constant time.

Fredman and Willard [8] construct the above fusion node in $O(k^4)$ time. While queries take constant time, updates are no more efficient than computing the fusion node from scratch.

III. Dynamic fusion nodes

We are now going to present our dynamic fusion node that for given $k \leq w^{1/4}$ maintains a dynamic set S of up to k w-bit integers, supporting both updates and queries in constant time using only standard operations on words, thus solving the open problem of Ajtai et al. [14].

As before, we assume that our set S is stored in an unordered array KEY of k words plus a single word $INDEX$ such that $\text{select}(i) = KEY[INDEX\langle i\rangle_{\lceil \lg k\rceil}]$ is the key of rank i in S. As described in the introduction, we can easily compute predecessor and successor using rank and select.

A. Matching with don't cares

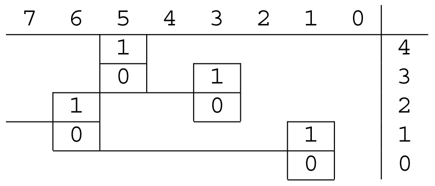
One of the problems making Fredman and Willard's fusion nodes dynamic is that when a key is inserted, we may get a new significant position j. Then, for every key $y \in S$, we have to insert $y\langle j\rangle_1$ into the compressed key $\hat{y}$, and there seems to be no easy way to support this change in constant time per update. In contrast, with a compressed trie, when a new key is inserted, we only have to include a single new branch node. However, as stated by Ajtai et al. [14], searching the compressed trie takes more than constant time.

Here, we will introduce don't cares in the compressed keys so as to make a perfect simulation of the compressed trie search in constant time, and yet support updates in constant time. Introducing don't cares may seem rather surprising in that they normally make problems harder, not easier, but in our context of small sets, it turns out that using don't cares is just the right thing to do. The basic idea is illustrated in Figure III.1.

Initially, we ignore the key compression, and focus only on our simulation of the search in a compressed trie. For a key $y \in S$, we will only care about bits $y\langle j\rangle_1$ such that $j = \text{msb}(y \oplus z)$ for some other $z \in S$. An equivalent definition is that if the trie has a branch node u at some level j, then we care about position j for all keys y descending from u. We will now create a key $y^?$ from y using characters from $\{$`0`, `1`, `?`$\}$. Even though these characters are not just bits, we let $y^?\langle j\rangle_1$ denote the jth character of y. We set

7	6	5	4	3	2	1	0	
1	1	1	1	1	0	1	0	4
1	1	0	1	1	0	1	0	3
1	1	0	1	0	0	1	0	2
1	0	0	1	0	0	1	0	1
1	0	0	1	0	0	0	1	0

An array with four 8-bit keys with bit positions on top and ranks on the right.



Compressed trie as used by Ajtai et al. [14]

6	5	3	1	
1	1	1	1	4
1	0	1	1	3
1	0	0	1	2
0	0	0	1	1
0	0	0	0	0

Compressed keys as used by Fredman and Willard [8].

6	5	3	1	
1	1	?	?	4
1	0	1	?	3
1	0	0	?	2
0	?	?	1	1
0	?	?	0	0

Compressed keys but with don't cares (?) in positions that are not used for branching in the (compressed) trie.

6	5	3	1	
1	1	x_3	x_1	4
1	0	1	x_1	3
1	0	0	x_1	2
0	x_5	x_3	1	1
0	x_5	x_3	0	0

When searching a key x, we replace don't cares at position j, with the jth bit x_j of x.

Figure III.1. Representations of a given key set.

$y^?\langle j\rangle_1 = ?$ if we do not care about position j in y; otherwise $y^?\langle j\rangle_1 = y\langle j\rangle_1$.

A query key x *matches* $y^?$ if and only if $x\langle j\rangle_1 = y^?\langle j\rangle_1$ for all j such that $y^?\langle j\rangle_1 \neq$?. Then x matches $y^?$ if and only if we get to y when searching x in the compressed trie over S. Since such a search in a compressed trie is always successful, we know that every x has a match $y^?$, $y \in S$.

Observation 3.1: Let $y^?$ be the match of x, and suppose $y \neq x$. Set $j = \text{msb}(x \oplus y)$. If $x < y$, then $x \wedge (\mathtt{1}^{w-j}\mathtt{0}^j)$ matches $z^?$ where z is the successor of x. If $x > y$, then $x \vee (\mathtt{0}^{w-j}\mathtt{1}^j)$ matches $z^?$ where z is the predecessor of x.

Proof: Since x matches $y^?$, we know that $y\langle j\rangle_1 =$?. Let (u, v) be the shortcut in the compressed trie bypassing position j. If $x < y$, the successor of x is the smallest key below v, which is exactly what we find when we search $x \wedge (\mathtt{1}^{w-j}\mathtt{0}^j)$ since we always pick the 0-child starting from v. Likewise if $x > y$, the predecessor of x is the largest key below y, which is exactly what we find when we search $x \vee (\mathtt{0}^{w-j}\mathtt{1}^j)$ since we always pick the 1-child starting from v. ■

Observation 3.1 implies that two matchings suffice to find the predecessor or successor of a key x. The second matching eliminates the need of the table RANK in the original static fusion nodes described above. This two step approach is similar to the "blind" compressed trie search for integers by Grossi et al. [23]. However, Grossi et al. do not use our matching with don't cares. Their data structure is static without updates, and they use tables of size $2^{\Omega(w)}$ like in Fredman and Willard's atomic heaps [15].

Next we observe that the key compression of fusion nodes does not affect the matching of keys with don't cares, for if we for some key care for position j, then by definition this is a significant position. Conversely this means that an insignificant position is a don't care in all keys, so skipping insignificant positions does not affect the matching, that is, x matches $y^?$ if and only if $\hat{x}$ matches $\hat{y}^?$ where $\hat{x}$ and $\hat{y}^?$ are the compressed version of x and $y^?$.

In Section III-B, we will show how to maintain a perfect compression which keeps the significant positions only. Thus, if $\ell \leq k$ is the number of significant positions, then $\hat{x}$ and $\hat{y}^?$ will both of length ℓ (recall that Fredman and Willard [8] had k^3 bits in their compressed keys, and they did not show how to update the compression when new keys got inserted). For consistent formatting, we view compressed keys as having k bits with $k - \ell$ leading 0s.

To check if $\hat{x}$ matches $\hat{y}^?$, we will create $\hat{y}^{\hat{x}}$ such that $\hat{y}^{\hat{x}}\langle h\rangle_1 = \hat{y}\langle h\rangle_1$ if $\hat{y}^?\langle h\rangle_1 \neq$? while $\hat{y}^{\hat{x}}\langle h\rangle_1 = \hat{x}\langle h\rangle_1$ if $\hat{y}^?\langle h\rangle_1 =$?. Then $\hat{x}$ matches $\hat{y}^?$ if and only if $\hat{x} = \hat{y}^{\hat{x}}$.

Note that if $y, z \in S$ and x is any w-bit key, then $y < z \iff \hat{y}^{\hat{x}} < \hat{z}^{\hat{x}}$. This is because the bits from x do not introduce any new branchings between the keys $y, z \in S$.

We will create $\hat{y}^{\hat{x}}$ for all $y \in S$ simultaneously in constant time. To this end, we maintain two $k \times k$ bit matrices $BRANCH$ and $FREE$ that together represent the stored compressed keys with don't cares. Let y be the key of rank i in S, and let c_h be a significant position. If $\hat{y}^?\langle h\rangle_1 \neq$?, then $BRANCH\langle i, h\rangle_{k\times 1} = \hat{y}\langle h\rangle_1$ and $FREE\langle i, h\rangle_{k\times 1} =$ 0. If $\hat{y}^?\langle h\rangle_1 =$?, then $BRANCH\langle i, h\rangle_{k\times 1} =$ 0 and

$FREE\langle i,h\rangle_{k\times 1} = \mathtt{1}$. For a compressed key $\hat{x}$, let $\hat{x}^k$ denote x repeated k times, obtained multiplying x by the constant $(\mathtt{0}^{k-1}\mathtt{1})^k$. Then $BRANCH \vee (\hat{x}^k \wedge FREE)$ is the desired array of the $\hat{y}^{\hat{x}}$ for $y \in S$, and we define

$$\text{match}(x) = \text{rank}(\hat{x}, BRANCH \vee (\hat{x}^k \wedge FREE)).$$

Here rank is implemented as in Lemma 2.1, and then match(x) gives the index of the $y^?$ matching x. By Observation 3.1, we can therefore compute the rank of x in S as follows.

rank(x) :

- $i = \text{match}(x)$.
- $y = KEY\left[INDEX\langle i\rangle_{\lceil \lg k\rceil}\right]$.
- if $x = y$ return i.
- $j = \text{msb}(x \oplus y)$.
- $i_0 = \text{match}(x \wedge (\mathtt{1}^{w-j}\mathtt{0}^j))$.
- $i_1 = \text{match}(x \vee (\mathtt{0}^{w-j}\mathtt{1}^j))$.
- if $x < y$, return $i_0 - 1$.
- if $x > y$, return i_1.

Inserting a key: We will only insert a key x after we have applied rank(x) as above, verifying that $x \notin S$. This also means that we have computed j, i_0 and i_1, where j denotes the position where x should branch off as a leaf child. The keys below the other child are exactly those with indices $i_0, ..., i_1$.

Before we continue, we have to consider if j is a new significant position, say with rank h among the current significant positions. If so, we have to update the compression function to include j, as we shall do it in Section III-B. The first effect on the current compressed key arrays is to insert a column h of don't cares. This is a new column of $\mathtt{0}$s in the $BRANCH$ array, and a column of $\mathtt{1}$s in the $FREE$ array. To achieve this effect, we first have to shift all columns $\geq h$ one to the left. To do this, we need some simple masks. The $k \times k$ bit matrix M_h with column h set is computed as

$$M_h = (\mathtt{0}^{k-1}\mathtt{1})^k \ll h,$$

and the $k \times k$ bit matrix $M_{i:j}$ with columns $i, ..., j$ set are computed as

$$M_{i:j} = M_{j+1} - M_i.$$

This calculation does not work for $M_{0:k-1}$ which instead is trivially computed as $M_{0:k-1} = \mathtt{1}^{k^2}$. For $X = BRANCH, FREE$, we want to shift all columns $\geq h$ one to the left. This is done by

$$X = (X \wedge M_{0:h-1}) \vee ((X \wedge M_{h:k-1}) \ll 1).$$

To fix the new column h, we set

$$FREE = FREE \vee M_h$$
$$BRANCH = BRANCH \wedge \neg M_h.$$

We have now made sure that column h corresponding to position $j = c_h$ is included as a significant position in $BRANCH$ and $FREE$. Now, for all keys with indices $i_0, ..., i_1$, we want to set the bits in column h to $y\langle j\rangle_1$. To do this, we compute the $k \times k$ bit mask $M^{i_0:i_1}$ with rows $i_0, ..., i_1$ set

$$M^{i_0:i_1} = (1 \ll ((i_1+1)\times k)) - (1 \ll (i_0 \times k))$$

and then we mask out column h within these rows by

$$M_h^{i_0:i_1} = M^{i_0:i_1} \wedge M_h.$$

We now set

$$FREE = FREE \wedge \neg M_h^{i_0:i_1}$$
$$BRANCH = BRANCH \vee (M_h^{i_0:i_1} \times y\langle j\rangle_1)$$

We now need to insert the row corresponding to $\hat{x}^?$ with $r = \text{rank}(x)$. Making room for row r in $X = BRANCH, FREE$, is done by

$$X = (X \wedge M^{0:r-1}) \vee ((X \wedge M^{r:k-1}) \ll k).$$

We also need to create the new row corresponding to $\hat{x}^?$. Let $\hat{y}^?$ be the compressed key with don't cares that x matched. Then

$$\hat{x}^?\langle 0..h-1\rangle_1 = \mathtt{?}^h$$
$$\hat{x}^?\langle h\rangle_1 = x\langle j\rangle_1$$
$$\hat{x}^?\langle h+1..k-1\rangle_1 = \hat{y}^?\langle h+1..k-1\rangle_1$$

With i the index of y after the above insertion of row r for x, we thus set

$$BRANCH\langle r, 0..h-1\rangle_{k\times 1} = \mathtt{0}^h$$
$$FREE\langle r, 0..h-1\rangle_{k\times 1} = \mathtt{1}^h$$
$$BRANCH\langle r, h\rangle_{k\times 1} = x\langle j\rangle_1$$
$$FREE\langle r, h\rangle_{k\times 1} = \mathtt{0}$$
$$BRANCH\langle r, h+1..k-1\rangle_{k\times 1} = BRANCH\langle i, h+1..k-1\rangle_{k\times 1}$$
$$FREE\langle r, h+1..k-1\rangle_{k\times 1} = FREE\langle i, h+1..k-1\rangle_{k\times 1}$$

This completes the update of $BRANCH$ and $FREE$. To delete a key, we just have to invert the above process. We still have to explain our new perfect compression.

B. Perfect dynamic key compression

As Fredman and Willard [8], we have a set of at most k significant bit positions $c_0, ..., c_{\ell-1} \in [w]$, $c_0 < c_1 < \cdots < c_{\ell-1}$. For a given key x, we want to construct, in constant time, an ℓ-bit compressed key $\hat{x}$ such that for $h \in [\ell]$, $\hat{x}\langle h\rangle_1 = x\langle c_h\rangle_1$. We will also support that significant positions can be added or removed in constant time, so as to be added or excluded in future compressions.

Fredman and Willard had a more loose compression, allowing the compressed key to have some irrelevant zeros between the significant bits, and the compressed keys to have length $O(k^3)$. Here, we will only keep the significant bits.

Our map will involve three multiplications. The first multiplication pack the significant bits in a segment of length $O(w/k^2)$. The last two multiplications will reorder them and place them consecutively.

C. Packing

We pack the significant bits using the approach of Tarjan and Yao for storing a sparse table [24]. We have a w-bit word B where the significant bits are set. For a given key x, we will operate on $x \wedge B$.

For some parameter b, we divide words into blocks of b bits. For our construction below, we can pick any b such that $k^2 \leq b \leq w/k^2$. We are going to pack the significant bits into a segment of $2b$ bits, so a smaller b would seem to be an advantage. However, if we later want a uniform algorithm that works when k is not a constant, then it is important that the number of blocks is $k^{O(1)}$. We therefore pick the maximal $b = w/k^2$.

We are looking for w/b shift values $s_i < k^2/2$ such that no two shifted blocks $B\langle i\rangle_{k^2} \ll s_i$ have a set bit in the same position. The packed key will be the $2b$ bit segment

$$\mu(x) = \sum_{i=0}^{w/b}((x \wedge B)\langle i\rangle_b \ll s_i).$$

Since we are never adding set bits in the same position, the set bits will appear directly in the sum. The shifts s_i are found greedily for $i = 0,..,w/b-1$. We want to pick s_i such that $B\langle i\rangle_b \ll s_i$ does not intersect with $\sum_{j=0}^{i-1}(B\langle j\rangle_b \ll s_j)$. A set bit p in $B\langle i\rangle_b$ collides with set bit q in $\sum_{j=0}^{i-1}(B\langle j\rangle_b \ll s_j)$ only if $s_i = p - q$. All together, we have at most k set bits, so there at most $k^2/4$ shift values s_i that we cannot use. Hence there is always a good shift value $s_i < k^2/2$.

So compute the above sum in constant time, we will maintain a word $BSHIFT$ that specifies the shifts of all the blocks. For $i = 0, ..., w/b - 1$, we will have

$$BSHIFT\langle w/b - 1 - i\rangle_b = 1 \ll s_i.$$

Essentially we wish to compute the packing via the product $(x \wedge B) \times BSHIFT$, but then we get interference between blocks. To avoid that we use a mask $E_b = (\mathtt{0}^b\mathtt{1}^b)^{w/(2b)}$ to pick out every other block, and then compute:

$$\begin{aligned}\mu(x) = &((x \wedge B \wedge E_b) \times (BSHIFT \wedge E_b)) + \\ &(((x \wedge B) \gg b) \wedge E_b) \times ((BSHIFT \gg b) \wedge E_b)) \\ &\gg (w - 2b).\end{aligned}$$

Resetting a block: If new significant bit j is included in B, we may have to change the shift s_i of the block i containing bit j. With b a power of two, $i = j \gg \lg b$.

The update is quite simple. First we remove block i from the packing by setting

$$BSHIFT\langle w/b - 1 - i\rangle_b = 0.$$

Now $\mu(B)$ describes the packing of all bits outside block i. To find a new shift value s_i, we use a constant S_0 with bk^2 bits, where for $s = 0,..,k^2/2 - 1$,

$$S_0\langle s\rangle_{2b} = 1 \ll s.$$

Computing the product $S_0 \times B\langle i\rangle_b$, we get that

$$(S_0 \times B\langle i\rangle_b)\langle s\rangle_{2b} = B\langle i\rangle_b \ll s$$

We can use s as a shift if and only if

$$\mu(B) \wedge (S_0 \times B\langle i\rangle_b)\langle s\rangle_{2b} = 0.$$

This value is always bounded by $(1 \ll 3b/2)$, so

$$\begin{aligned}&\mu(B) \wedge (S_0 \times B\langle i\rangle_b)\langle s\rangle_{2b} = 0 \iff 0 \neq \\ &(1 \ll 3b/2) \wedge ((1 \ll 3b/2) - (\mu(B) \wedge (S_0 \times B\langle i\rangle_b)\langle s\rangle_{2b})).\end{aligned}$$

We can thus compute the smallest good shift value s_i by

$$\begin{aligned}X =& (1 \ll 3b/2)^{k^2/2} \wedge \\ &((1 \ll 3b/2)^{k^2/2} - (\mu(B)^{k^2/2} \wedge (S_0 \times B\langle i\rangle_b)) \\ s_i =& \operatorname{lsb}(X) \gg \lg(2b).\end{aligned}$$

All that remains is to set

$$BSHIFT\langle i\rangle_b = 1 \ll s_i.$$

Then $\mu(x)$ gives the desired updated packing.

D. Final reordering

We now have all significant bits packed in $\mu(x)$ which has $2b \leq 2w/k^2$ bits, with significant bit h in some position μ_h. We want to compute the compressed key $\hat{x}$ with all significant bits in place, that is, $\hat{x}\langle h\rangle_1 = \hat{x}\langle c_h\rangle_1$.

The first step is to create a word $LSHIFT$ with k fields of $4b$ bit fields where $LSHIFT\langle h\rangle_{4b} = 1 \ll (2b - \mu_h)$. Then $(LSHIFT \times \mu(x)) \gg 2b$ is a word where the hth field has the hth significant bit in its first position. We mask out all other bits using $((LSHIFT \times \mu(x)) \gg 2b) \wedge (\mathtt{0}^{4b-1}\mathtt{1})^k$.

Finally, to collect the significant bits, we multiply with the $k \times 4b$ bit constant S_1 where for $h \in [k]$, $S_1\langle k-1-h\rangle_{4b} = \mathtt{1} \ll h$. Now the compressed key can be computed as

$$\begin{aligned}\hat{x} =& \left(S_1 \times \left(((LSHIFT \times \mu(x)) \gg 2b) \wedge (\mathtt{0}^{4b-1}\mathtt{1})^k\right)\right) \\ &\gg ((k-1) \times 4b).\end{aligned}$$

Updates: Finally we need to describe how to maintain $LSHIFT$ in constant time per update. To this end, we will also maintain a $k \times \lg w$ bit word C with the indices of the significant bits, that is, $C\langle h\rangle_{\lg w} = c_h$.

We can view updates as partitioned in two parts: one is to change the shift of block i, and the other is to insert or remove a significant bit. First we consider the changed shift of block i from s_i to s'_i. If block i has no significant bits, that is, if $B\langle i\rangle_b = 0$, there is nothing to be done. Otherwise, the index of the first significant bit in block i is computed as $h_0 = \operatorname{rank}(i \times b, C)$, and the index of the last one is computed as $h_1 = \operatorname{rank}((i+1) \times b, C) - 1$.

In the packing, for $h = h_0,..,h_1$, significant bit h will move to $\mu'_h = \mu_h + s'_i - s_i$. The numbers μ_h are not stored explicitly, but only as shifts in the sense that $LSHIFT\langle h\rangle_{4b} = \mathtt{1} \ll (2b - \mu_i)$. The change in shift is the same for all $h \in [h_0, h_1]$, so all we have to do is to set

$$LSHIFT\langle h_0..h_1\rangle_{4b} = LSHIFT\langle h_0..h_1\rangle_{4b} \ll (s_b - s'_b),$$

interpreting "$\ll (s_b - s'_b)$" as "$\gg (s'_b - s_b)$" if $s_b < s'_b$.

The other operation we have to handle is when we get a new significant bit h is introduced at position $c_h \in [w]$, that is, $B\langle c_h \rangle_1 = \texttt{1}$. First we have to insert c_h in C, setting

$$C\langle h+1..*\rangle_{\lg w} = C\langle h..*\rangle_{\lg w} \text{ and } C\langle h\rangle_{\lg w} = c_h.$$

Significant bit h resides in block $i = c_h \gg \lg b$, and the packing will place it in position $\mu_i = c_h - i \times b + s_i$. Thus for the final update of $LSHIFT$, we set

$$\begin{aligned} LSHIFT\langle h+1..*\rangle_{4b} =& LSHIFT\langle h..*\rangle_{4b} \\ LSHIFT\langle h\rangle_{4b} =& 1 \ll (2b - \mu_i). \end{aligned}$$

Removal of significant bit h is accomplished by the simple statements:

$$\begin{aligned} C\langle h..*\rangle_{\lg w} =& C\langle h+1..*\rangle_{\lg w} \\ LSHIFT\langle h..*\rangle_{4b} =& LSHIFT\langle h+1..*\rangle_{4b}. \end{aligned}$$

This completes our description of perfect compression.

IV. Dynamic fusion trees with operation time $O(\log n/\log w)$

We will now show how to maintain a dynamic set S of n w-bit integers, supporting all our operations in $O(\log n/\log w)$ time. Our fusion nodes of size k is used in a fusion tree which is a search tree of degree $\Theta(k)$. Because our fusion nodes are dynamic, we avoid the dynamic binary search tree at the bottom of the original dynamic fusion trees [8]. Our fusion trees are augmented to handle rank and select using the approach of Dietz [19] for maintaining order in a linked list. Dietz, however, was only dealing with a list and no keys, and he used tabulation that could only handle nodes of size $O(\log n)$. To exploit a potentially larger word length w, we will employ a simplifying trick of Pătraşcu and Demaine [20, §8] for partial sums over small weights. For simplicity, we are going to represent the set $S \cup \{-\infty\}$ where $-\infty$ is a key smaller than any real key.

Our dynamic fusion node supports all operations in constant time for sets of size up to $k = w^{1/4}$. We will create a fusion search tree with degrees between $k/16$ and k. The keys are all in the leaves, and a node on height i will have between $(k/4)^i/4$ and $(k/4)^i$ descending leaves. It standard that constant time for updates to a node imply that we can maintain the balance for a tree in time proportional to the height when leaves are inserted or deleted. When siblings are merged or split, we build new node structures in the background, adding children one at the time (see, e.g., [13, §3] for details). The total height is $O(\log n/\log w)$, which will bound our query and update time.

First, assume that we only want to support predecessor search. For each child v of a node u, we want to know the smallest key descending from v. The set S_u of these smallest keys is maintained by our fusion node. This includes KEY_u and $INDEX_h$ as described in Section II. The ith key is found as $KEY_u[INDEX_u\langle i\rangle_{\lceil \lg k\rceil}]$. We will have another array $CHILD_u$ that gives us pointers to the children of u, where $CHILD_u$ uses the same indexing as KEY_u. The ith child is thus pointed to by $CHILD_u[INDEX_u\langle i\rangle_{\lceil \lg k\rceil}]$.

When we come to a fusion node u with a key x, we get in constant time the index $i = \text{rank}_u(x)$ of the child v such that x belongs under, that is, the largest i such that $KEY[INDEX\langle i\rangle_{\lceil \lg k\rceil}] < x$. Following $CHILD_u[INDEX\langle i\rangle_{\lceil \lg k\rceil}]$ down, we eventually get to the leaf with the predecessor of x in S.

Now, if we also want to know the rank of x, then for each node u, and child index i of u, we want to know the number $N_u[i]$ of keys descending from the children indexed $< i$. Adding up these numbers from all nodes on the search path gives us the rank of x (as a technical detail, the count will also count $-\infty$ for one unless the search ends $-\infty$).

The issue is how we can maintain the numbers $N[i]$; for when we add a key below the child indexed i, we want to add one to $N[j]$ for all of $j > i$. As in [20, §8] we will maintain $N_u[i]$ as the sum of two numbers. For each node u, we will have a regular array M_u of k numbers, each one word, plus an array D_u of k numbers between $-k$ and k, each using $\lceil \lg k\rceil + 1$ bits. We will always have $N_u[i] = M_u[i] + D_u\langle i\rangle_{\lceil \lg k\rceil+1}$. As described in [20, §8], it easy in constant time to add 1 to $D_u\langle j\rangle_{\lceil \lg k\rceil+1}$ for all $j > i$, simply by setting

$$D_u\langle i+1..*\rangle_{\lceil \lg k\rceil+1} = D_u\langle i+1..*\rangle_{\lceil \lg k\rceil+1} + (\texttt{0}^{\lceil \lg k\rceil}\texttt{1})^{k-i}.$$

Finally, consider the issue of selecting the key with rank r. This is done recursively, staring from the root. When we are at a node u, we look for the largest i such that $N_u[i] < r$. Then we set $r = r - N_u[i]$, and continue the search from child i of u. To find this child index i, we adopt a nice trick of Dietz [19] which works when u is of height $h \geq 2$. We use M_u as a good approximation for N_u which does not change too often. We maintain an array Q_u that for each $j \in [k]$ stores the largest index $i' = Q[j]$ such that $M_u[i'] < jk^{h-1}/4^h$. Then the right index i is at most 5 different from $i' = Q_u[r4^h/k^{h-1}]$, so we can check the 10 relevant indices i exhaustively. Maintaining Q_u is easy since we in round-robin fashion only change one $M_u[j]$ at the time, affecting at most a constant number of $Q_u[j]$. For nodes of height 1, we do not need to do anything; for the children are the leaf keys, so select r just returns the child of index r.

This completes the description of our dynamic fusion tree supporting rank, select, and predecessor search, plus updates, all in time $O(\log n/\log w)$.

V. Lower bounds

First we note that dynamic cell-probe lower bounds hold regardless of the available space. The simple point is that if we start with an empty set and insert n elements in $O(\log n)$ amortized time per element, then, using perfect hashing, we could make an $O(n \log n)$ space data structure, that in constant time could tell the contents of all cells written based on the integers inserted. Any preprocessing done before the integers were inserted would be a universal constant, which is "known" to the cell probe querier. The cost of the next dynamic query can thus be matched by a static data structure using $O(n \log n)$ space.

We claimed that Fredman and Saks [9] provides an $\Omega(\log n/\log w)$ lower bound for dynamic rank and select. Both follow by reduction from dynamic prefix parity where we have an array B with n bits, starting, say, with all zeros. An update flips bit i, and query asks what is the xor of the first j bits. By [9, Theorem 3], for $\log n \leq w$, the amortized operation time is $\Omega(\log n/\log w)$.

Rank: Fredman and Saks mention that a lower bound for rank follows by an easy reduction: simply consider a dynamic set $S \subseteq [n]$, where $i \in S$ iff $B[i] = 1$. Thus we add i to S when we set $B[i]$, and delete i from S when we unset $B[i]$. The prefix parity of $B[0,..,j]$ is the parity of the rank of j in S.

Select: Our reduction to selection is bit more convoluted. We assume that n is divisible by 2. For dynamic select, we consider a dynamic set S of integers from $[n(n+1)]$. which we initialize as the set S_0 consisting of the integers

$$S_0 = \{i(n+1) + j + 1 \mid i, j \in [n]\}$$

We let bit $B[i]$ in prefix parity correspond to integer $i(n+1) \in S$. When we want to ask for prefix parity of $B[0,..,j]$, we set y =select$((j+1)n - 1)$. If there are r set bits in $B[0,..,j]$, then $y = (j+1)(n+1) - 1 - r$, so we can compute r from y.

To understand that the above is a legal reduction, note that putting in S_0 which is a fixed constant does not help. While it is true that the data structure for dynamic select can do work when S_0 is being inserted, this is worth nothing in the cell probe model, for S_0 is a fixed constant, so in the cell-probe, if this was useful, it could just be simulated by the data structure for prefix parity, at no cost.

Predecessor: The query time for dynamic predecessor with logarithmic update time cannot be better than that of static predecessor using polynomial space, for which Beame and Fich [10, Theorem 3.6 plus reductions in Sections 3.1-2] have proved a lower bond of $\Omega(\log n/\log w)$ if $\log n/\log w = O(\log(w/\log n)/\log\log(w/\log n))$. But the condition is equivalent to $\log n/\log w = O(\log w/\log\log w)$. To see the equivalence, note that if $w \geq \log^2 n$ then $\log(w/\log n) = \Theta(\log w)$ and if $w \leq \log^2 n$, both forms of the conditions are satisfied. Sen and Vankatesh [25] have proved that the lower bound holds even if we allow randomization.

VI. Optimal randomized dynamic predecessor

From [12], [26] we know that with key length ℓ and word length w, the optimal (randomized) query time for static predecessor search using $\tilde{O}(n)$ space for n keys is within a constant factor of

$$\max\{1, \min\begin{cases} \frac{\log n}{\log w} \\ \dfrac{\log \frac{\ell}{\log w}}{\log\left(\log \frac{\ell}{\log w} \middle/ \log \frac{\log n}{\log w}\right)} \\ \log \frac{\ell - \lg n}{\log w} \end{cases}$$

This is then also a lower bound for the dynamic case with logarithmic update times, for we can use a dynamic data structure to build a static data structure, simply by inserting the n keys. However, in the introduction, we actually replaced the last branch $\log \frac{\ell - \lg n}{\log w}$ with the higher bound $\log \frac{\log(2^\ell - n)}{\log w}$. To get this improvement from the static lower bound, we do as follows. First note that we can assume that $\log n \geq \ell/2$, for otherwise the bounds are asymptotically the same. We now set $n' = \sqrt{2^\ell - n} < n$ and $\ell' = \lfloor \lg(2^\ell - n) \rfloor$. For a set $S' \subseteq [2^{\ell'}]$ of size n' using $\tilde{O}(n')$ space, the static lower bound states that the query time is $\Omega(\log \frac{\ell' - \lg n'}{\log w}) = \Omega(\log \frac{\log(2^\ell - n)}{\log w})$. To create such a data structure dynamically, first, in a preliminary step, for $i = 1, ..., n - n'$, we insert the key $2^\ell - i$. We have not yet inserted any keys from $[2^{\ell'}]$. Now we insert the keys from S'. It is only the $\tilde{O}(|S'|)$ cells written after we start inserting S' that carry any information about S'. Everything written before is viewed as a universal constant not counted in the cell-probe model. The newly written cells and their addresses, form a near-linear space representation of S', to which the static cell-probe lower bound of $\Omega(\log \frac{\log(2^\ell - n)}{\log w})$ applies.

The main result of this paper is that the top $O(\log n/\log w)$ branch is possible. Essentially this was the missing dynamic bound for predecessor search if we allow randomization.

Going carefully through Section 5 of the full version of [12], we see that the lower two branches are fairly easy to implement dynamically if we use randomized hash tables with constant expected update and query time (or alternatively, with constant query time and constant amortized expected update time [27]). The last branch is van Emde Boas' data structure [11], [28] with some simple tuning. This includes the space saving Y-fast tries of Willard [29], but using our dynamic fusion nodes for $w^{\Theta(1)}$-sized bottom trees that are searched and updated in constant time. This immediately gives a bound of $O(\log \frac{\ell}{\log w})$. The improvement to $O(\log \frac{\log(2^\ell - n)}{\log w})$ is obtained by observing that for each key stored, both neighbors are at distance at most $2^\ell - n$, implying a commong prefix missing at most $\lg(2^\ell - n)$ bits.

To implement the middle part, we follow the small space static construction from [12, Full version, Section 5.5.2]. For certain parameters q and h, [12] uses a special data structure [12, Full version, Lemma 20] (which is based on [10]) using $O(q^{2h})$ space, which is also the construction time. They use $q = n^{1/(4h)}$ for a $O(\sqrt{n})$ space and construction time. The special data structure only has to change when a subproblem of size $\Theta(n/q) = \Theta(n^{1-1/(4h)})$ changes in size by a constant factor, which means that the $O(\sqrt{n})$ construction time gets amortized as sub-constant time per key update. This is similar to the calculation behind exponential search trees [13] and we can use the same de-amortization. Other than the special table, each node needs a regular hash table like van Emde Boas' data structure, as discussed above.

For a concrete parameter choice, we apply our dynamic fusion node at the bottom to get $O(w)$ free space per key.

Then, as in [12],

$$h = \frac{\lg \frac{\ell}{a}}{\lg \frac{\lg n}{a}} \Bigg/ \lg \frac{\lg \frac{\ell}{a}}{\lg \frac{\lg n}{a}} \quad \text{where } a = \lg w.$$

This yields the desired recursion depth of

$$O\left(\frac{\lg \frac{\ell}{\lg w}}{\lg h} + h \lg \frac{\lg n}{\lg w}\right) = O\left(\frac{\lg \frac{\ell}{\log w}}{\lg\left(\lg \frac{\ell}{\log w} \Big/ \lg \frac{\lg n}{\log w}\right)}\right).$$

Funding

Research of Mikkel Thorup partly supported by an Advanced Grant from the Danish Council for Independent Research under the Sapere Aude research carrier programme.

References

[1] B. Kernighan and D. Ritchie, *The C Programming Language*. Prentice Hall, 1978.

[2] L. J. Comrie, "The hollerith and powers tabulating machines," *Trans. Office Machinary Users' Assoc., Ltd*, pp. 25–37, 1929-30.

[3] A. I. Dumey, "Indexing for rapid random access memory systems," *Computers and Automation*, vol. 5, no. 12, pp. 6–9, 1956.

[4] S. A. Cook and R. A. Reckhow, "Time bounded random access machines," *Journal of Computer and System Sciences*, vol. 7, no. 4, pp. 354–375, 1973.

[5] W. J. Paul and J. Simon, "Decision trees and random access machines," in *Logic and Algorithmic: An International Symposium Held in Honour of Ernst Specker*. L'Enseignement Mathématique, Université de Genevè, 1982, pp. 331–340.

[6] A. C.-C. Yao, "Should tables be sorted?" *Journal of the ACM*, vol. 28, no. 3, pp. 615–628, 1981, see also FOCS'78.

[7] D. Kirkpatrick and S. Reisch, "Upper bounds for sorting integers on random access machines," *Theoretical Computer Science*, vol. 28, pp. 263–276, 1984.

[8] M. L. Fredman and D. E. Willard, "Surpassing the information theoretic bound with fusion trees," *Journal of Computer and System Sciences*, vol. 47, no. 3, pp. 424–436, 1993, see also STOC'90.

[9] M. L. Fredman and M. E. Saks, "The cell probe complexity of dynamic data structures," in *Proc. 21st ACM Symposium on Theory of Computing (STOC)*, 1989, pp. 345–354.

[10] P. Beame and F. E. Fich, "Optimal bounds for the predecessor problem and related problems," *Journal of Computer and System Sciences*, vol. 65, no. 1, pp. 38–72, 2002, see also STOC'99.

[11] P. van Emde Boas, R. Kaas, and E. Zijlstra, "Design and implementation of an efficient priority queue," *Mathematical Systems Theory*, vol. 10, pp. 99–127, 1977, conference version by van Emde Boas alone in FOCS'75.

[12] M. Pǎtraşcu and M. Thorup, "Time-space trade-offs for predecessor search," in *Proc. 38th ACM Symposium on Theory of Computing (STOC)*, 2006, pp. 232–240, full version: `http://arxiv.org/abs/cs/0603043`.

[13] A. Andersson and M. Thorup, "Dynamic ordered sets with exponential search trees," *Journal of the ACM*, vol. 54, no. 3, 2007, see also FOCS'96, STOC'00.

[14] M. Ajtai, M. L. Fredman, and J. Komlós, "Hash functions for priority queues," *Information and Control*, vol. 63, no. 3, pp. 217–225, 1984, see also FOCS'83.

[15] M. L. Fredman and D. E. Willard, "Trans-dichotomous algorithms for minimum spanning trees and shortest paths," *Journal of Computer and System Sciences*, vol. 48, no. 3, pp. 533–551, 1994, see also FOCS'90.

[16] M. Thorup, "On AC^0 implementations of fusion trees and atomic heaps," in *Proc. 14th ACM/SIAM Symposium on Discrete Algorithms (SODA)*, 2003, pp. 699–707.

[17] A. Andersson, P. B. Miltersen, and M. Thorup, "Fusion trees can be implemented with AC^0 instructions only," *Theoretical Computer Science*, vol. 215, no. 1-2, pp. 337–344, 1999.

[18] D. E. Willard, "Examining computational geometry, van Emde Boas trees, and hashing from the perspective of the fusion tree," *SIAM Journal on Computing*, vol. 29, no. 3, pp. 1030–1049, 2000, see also SODA'92.

[19] P. F. Dietz, "Optimal algorithms for list indexing and subset rank," in *Proc. 1st Workshop on Algorithms and Data Structures (WADS)*, 1989, pp. 39–46.

[20] M. Pǎtraşcu and E. D. Demaine, "Lower bounds for dynamic connectivity," in *Proc. 36th ACM Symposium on Theory of Computing (STOC)*, 2004, pp. 546–553.

[21] E. Fredkin, "Trie memory," *Communications of the ACM*, vol. 3, no. 9, pp. 490–499, 1960.

[22] D. R. Morrison, "PATRICIA - practical algorithm to retrieve information coded in alphanumeric," *Journal of the ACM*, vol. 15, no. 4, pp. 514–534, 1968.

[23] R. Grossi, A. Orlandi, R. Raman, and S. S. Rao, "More haste, less waste: Lowering the redundancy in fully indexable dictionaries," in *Proc. 26th Symposium on Theoretical Aspects of Computer Science (STACS)*, 2009, pp. 517–528.

[24] R. E. Tarjan and A. C.-C. Yao, "Storing a sparse table," *Communications of the ACM*, vol. 22, no. 11, pp. 606–611, 1979.

[25] P. Sen and S. Venkatesh, "Lower bounds for predecessor searching in the cell probe model," *Journal of Computer and System Sciences*, vol. 74, no. 3, pp. 364–385, 2008, see also ICALP'01, CCC'03.

[26] M. Pǎtraşcu and M. Thorup, "Randomization does not help searching predecessors," in *Proc. 18th ACM/SIAM Symposium on Discrete Algorithms (SODA)*, 2007, pp. 555–564.

[27] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. M. auf der Heide, H. Rohnert, and R. E. Tarjan, "Dynamic perfect hashing: Upper and lower bounds," *SIAM Journal on Computing*, vol. 23, no. 4, pp. 738–761, 1994, see also FOCS'88.

[28] K. Mehlhorn and S. Näher, "Bounded ordered dictionaries in O(log log n) time and O(n) space," *Inf. Process. Lett.*, vol. 35, no. 4, pp. 183–189, 1990.

[29] D. E. Willard, "Log-logarithmic worst-case range queries are possible in space $\Theta(N)$," *Information Processing Letters*, vol. 17, no. 2, pp. 81–84, 1983.