Predecessor Queries in Constant Time? ⋆

Marek Karpinski [a,1] Yakov Nekrich [a,2]

[a] *Dept. of Computer Science , University of Bonn*

Abstract

In this paper we design a new static data structure for *batched* predecessor queries. In particular, our data structure supports $O(\sqrt{\log n})$ queries in $O(1)$ time per query and requires $O(n^{\varepsilon\sqrt{\log n}})$ space for any $\varepsilon > 0$. This is the first $O(1)$ time data structure for batched queries that uses $N^{o(1)}$ space for arbitrary $N = \omega(n^{\varepsilon\sqrt{\log n}})$ where N is the size of the universe. We also present a data structure that answers $O(\log\log N)$ predecessor queries in $O(1)$ time per query and requires $O(n^{\varepsilon \log\log N})$ space for any $\varepsilon > 0$. The method of solution relies on a certain way of searching for predecessors of all elements of the query *in parallel*.

In a general case, our approach leads to a data structure that supports $p(n)$ queries in $O(\sqrt{\log n}/p(n))$ time per query and requires $O(n^{p(n)})$ space for any $p(n) = O(\sqrt{\log n})$, and a data structure that supports $p(N)$ queries in $O(\log\log N/p(N))$ time per query and requires $O(n^{p(N)})$ space for any $p(N) = O(\log\log N)$.

1 Introduction

Given a set A of integers, the predecessor problem consists in finding for an arbitrary integer x the biggest $a \in A$, such that $a \leq x$. If x is smaller than all elements in A, a default value is returned. This fundamental problem was considered in a number of papers, e.g., [AL62], [EKZ77], [FW93], [A95], [H98], [AT00], [BF02], [BCKM01]. In this paper we present a static data structure that supports *batched* predecessor queries in $O(1)$ amortized time.

⋆ A preliminary version of this paper appeared in the Proceedings of the 13th Annual European Symposium on Algorithms (ESA 2005), pp. 238-248.

[1] E-mail marek@cs.uni-bonn.de. Work partially supported by a DFG grant, Max-Planck Research Prize, and IST grant 14036 (RAND-APX).

[2] E-mail yasha@cs.uni-bonn.de. Work partially supported by IST grant 14036 (RAND-APX).

In the *comparison model*, if only comparisons between pairs of elements are allowed, the predecessor problem has time complexity $O(\log n)$, where n is the number of elements. A standard information-theoretic argument proves that $\lceil \log n \rceil$ comparisons are necessary. However in [E77], [EKZ77] a data structure supporting predecessor queries in $O(\log \log N)$ time, where N is the size of the universe, was presented. Fusion trees, presented by Fredman and Willard [FW93], support predecessor queries in $O(\sqrt{\log n})$ time, independently of the size of the universe. This result was further improved in other important papers, e.g., [A95], [AT00],[BF02].

An important computational model for proving lower bounds is the cell-probe model [Y81]. In this model the data is stored in *cells*, and each of the cells consists of a fixed number of bits. Only the number of accesses to memory cells is counted; hence, the cell-probe model is more powerful than the RAM model. Ajtai[A88] has shown that no polynomial space data structure answers predecessor queries in worst case constant time. In [Mil94] a $\Omega(\sqrt{\log \log N})$ lower bound was proven for N the size of the universe, and in [MNSW98] a $\Omega(\log^{1/3} n)$ lower bound was proven. In the paper of Beame and Fich [BF02], it was shown that any data structure using $n^{O(1)}$ words of $(\log N)^{O(1)}$ bits, requires $\Omega(\sqrt{\log n / \log \log n})$ and $\Omega(\log \log N / \log \log \log N)$ query time in the worst case. This result was extended to randomized data structures in [S03]. It can also be shown that the $\Omega(\sqrt{\log n / \log \log n})$ lower bound remains valid if the data structure consists of $n^{\log^{O(1)} n}$ words of $(\log N)^{O(1)}$ bits. This can be shown by a simple change of parameters in the proof of [S03]; we provide the proof of this statement in the Appendix. All the above mentioned lower bounds are in the cell-probe model. In [BF02] the authors also presented a matching upper bounds for the RAM model and polynomial space, and transformed the first result into a linear space and $O(\sqrt{\log n / \log \log n})$ time data structure, using the exponential trees of Andersson and Thorup [A96],[AT00]. Recently, new upper and lower bounds for several important cases of the predecessor problem (such as linear or almost linear space data structures) were shown by Pătraşcu and Thorup [PT06].

Ajtai, Fredman and Komlòs [AFK84] have shown that if word size is $n^{\Omega(1)}$, then predecessor queries have time complexity $O(1)$ in the cell probe model ([Y81]). The same result can be obtained for the RAM model using the fusion tree [FW93]. Obviously, there exists a $O(N)$ space and $O(1)$ query time static data structure for the predecessor queries. Brodnik, Carlsson, Karlsson, and Munro [BCKM01] presented a constant time and $O(N)$ space dynamic data structure. But their data structure uses an unusual notion of the word of memory: an individual bit may occur in a number of different words.

It is interesting to compare the predecessor problem and the sorting problem with respect to the word size. In [AHNR95] it was shown that there is a

randomized linear time algorithm for the integer sorting problem, if the word size $w = \Omega(\log^{2+\varepsilon} n)$ for any $\varepsilon > 0$. On the other hand, if $w = O(\log n)$ we also can sort n integers in $O(n)$ time. The predecessor problem can be solved in $O(1)$ time and polynomial space in the RAM model, if $w = O(\log n)$ or $w = \Omega(n^{\varepsilon})$ (using [FW93]). Thus there are "good" solutions for both problems, if the word size is either small or very big. However, in the case of the predecessor problem the gap between "good" word sizes is significantly larger.

While in real-time applications every query must be processed as soon as it is known to the data base, in many other applications we can collect a number of queries and process the set of queries simultaneously. In this scenario, the size of the query set is also of interest. Andersson [A95] presented a static data structure that uses $O(n^{\varepsilon \log n})$ space and answers $\log n$ predecessor queries in time $O(\log n \log\log n)$; here and further ε denotes an arbitrary positive constant. Batched processing is also considered in e.g., [GL01], where batched queries to unsorted data are considered.

In this paper we present a static data structure that uses $O(n^{p(n)})$ space and answers $p(n)$ queries in $O(\sqrt{\log n})$ time, for any $p(n) = O(\sqrt{\log n})$. In particular, we present a $O(n^{\varepsilon\sqrt{\log n}})$ space data structure that answers $\sqrt{\log n}$ queries in $O(\sqrt{\log n})$ time. The model used is the RAM model with word size w, and the size of the universe is $N = 2^w$. To the best of our knowledge, this is the first algorithm that uses $N^{o(1)}$ space and words with $O(\log N)$ bits, and achieves $O(1)$ amortized query time for batched queries and for arbitrary $N = \omega(n^{\varepsilon\sqrt{\log n}})$. From the extension of the lower bound of [BF02], [S03] it follows that the $O(1)$ query time cannot be achieved by a $O(2^{\log^{O(1)} n})$ space data structure, if the size of the query set is $o(\sqrt{\log n/\log\log n})$.

Our approach also leads to a $O(n^{p(N)})$ space data structure that answers $p(N)$ queries in time $O(\log\log N)$, where $p(N) = O(\log\log N)$. Thus, there exists a data structure that answers $\log\log N$ queries in $O(\log\log N)$ time and uses $O(n^{\varepsilon \log\log N})$ space. For instance, for $N = n^{\log^{O(1)} n}$, there is a $O(n^{\varepsilon \log\log n})$ space data structure that answers a batch of $\log\log n$ queries in $O(1)$ time per query.

The main idea of our method is to search in a certain way for predecessors of all elements of the query set *simultaneously*. We reduce the key size for all elements by multiple membership queries in the spirit of [BF02]. When the key size is sufficiently small, predecessors can be found by multiple comparisons. A similar approach was also used in the paper of Andersson[A95] that was the starting point of our investigation.

After some preliminary definitions in Section 2, we give an overview of our method in Section 3. In Section 3 a $O(n^{2\sqrt{\log n}+2})$ space and $O(1)$ amortized

time data structure for batched queries is also presented. We generalize this result and describe its improvements in Section 4.

2 Preliminaries and Notation

In this paper we use the RAM model of computation that supports addition, multiplication, division, bit shifts, and bitwise boolean operations in $O(1)$ time. Here and further w denotes the word size; b denotes the size of the keys, and we assume without loss of generality that b is a power of 2. *Query set* $Q = \{x_1, x_2, \ldots, x_q\}$ is the set of elements whose predecessors should be found. Left and right bit shift operations are denoted with $\ll$ and $\gg$ respectively, i.e. $x \ll k = x \cdot 2^k$ and $x \gg k = x \div 2^k$, where $\div$ is the integer division operation. Bitwise logical operations are denoted by AND, OR, XOR, and NOT. If x is a binary string of length k, where k is even, x^{U} denotes the prefix of x of length $k/2$, and x^{L} denotes the suffix of x of length $k/2$.

In the paper of Beame and Fich [BF02], it is shown how multiple membership queries, can be answered simultaneously in $O(1)$ time, if the word size is sufficiently large. The following statement will be extensively used in our construction.

Lemma 1 *Given sets $S_1, S_2, \ldots, S_q$ such that $|S_i| = n_i$, $S_i \subset [0, 2^b - 1]$, and $q \leq \sqrt{\frac{w}{b}}$, there is a data structure that uses $O(bq \prod_{i=1}^{q}(4n_i))$ bits, can be constructed in $O(q \prod_{i=1}^{q}(4n_i))$ time, and answers q queries $p_1 \in S_1?, p_2 \in S_2?, \ldots, p_q \in S_q?$ in $O(1)$ time.*

This Lemma is a straightforward extension of Lemma 4.1 in [BF02] for the case when the sets S_i are of different size; for completeness, we provide its proof in the Appendix.

A predecessor query on a set S of integers in the range $[0, 2^b - 1]$ can be reduced in $O(1)$ time to a predecessor query on set S' with at most $|S|$ elements in the range $[0, 2^{b/2} - 1]$. This well known idea and its variants are used in van Emde Boas data structure [E77], x-fast trie [W83], as well as in the number of other important papers, e.g.,[A95],[A96], [AT00].

In this paper the following variant of this construction will be used. Consider a binary trie T for elements of S. Let $T_0 = T$. Let $H(S)$ be the set of non-empty nodes of T_0 on level $b/2$. That is, $H(S)$ is the set of prefixes of elements in S of length $b/2$. If $|S| \leq 4$, elements of S are stored in a list and predecessor queries can obviously be answered in constant time. Otherwise, a data structure that answers membership queries $e' \in H(S)?$ in constant time is stored. Using hash functions, such a data structure can be stored in $O(n)$ space. A recursively

defined data structure $(D)_u$ contains all elements of $H(S)$. For every $e' \in H(S)$ data structure $(D)_{e'}$ is stored; $(D)_{e'}$ contains all length $b/2$ suffixes of elements $e \in S$, such that e' is a prefix of e. D_u and all $D_{e'}$ contain keys in the range $[0, 2^{b/2} - 1]$. $(S)_u$ and $(S)_{e'}$ denote the sets of elements in $(D)_u$ and $(D)_{e'}$ respectively. For every node v of the global trie T that corresponds to an element stored in a data structure on some level, we store $v.min$ and $v.max$, the minimal and maximal leaf descendants of v in T. All elements of S are also stored in a doubly linked list, so that the predecessor $pred(x)$ of every element x in S can be found in constant time.

Suppose we are looking for a predecessor of $x \in [0, 2^b - 1]$. If $x^{\mathrm{U}} \in H(S)$, we look for a predecessor of x^{L} in $D_{x^{\mathrm{U}}}$. If x^{L} is smaller than all elements in $D_{x^{\mathrm{U}}}$, the predecessor of x is $pred(x^{\mathrm{U}}.min)$. If $x^{\mathrm{U}} \notin H(S)$, the predecessor of x is $m.max$, where m is the node in T corresponding to the predecessor of x^{U} in $(D)_u$. Using i levels of the above data structure a predecessor query with key length b can be reduced to a predecessor query with key length $b/2^i$ in $O(i)$ time. We will call data structures that contain keys of length $b/2^i$ level i data structures, and the corresponding sets of elements will be called level i sets.

It was shown before that if word size w is bigger than bk, then predecessor queries can be answered in $O(\log n/\log k)$ time with help of packed B-trees of Andersson [A95] (see also [H98]). Using the van Emde Boas construction described above, we can reduce the key size from w to $w/2^{\sqrt{\log n}}$ in $O(\sqrt{\log n})$ time. After this, the predecessor can be found in $O(\log n/\sqrt{\log n}) = O(\sqrt{\log n})$ time ([A95]).

The data structure described in the next section has space complexity $O(n^{2\sqrt{\log n}+2})$. Hence, to address the elements of our data structure in constant time, we need the word size $w \geq (2\log^{3/2} n + 2\log n)$. But in the case $w \leq (2\log^{3/2} n + 2\log n)$, the universe size is $N = O(n^{2\sqrt{\log n}+2})$

Fact 1 *If the universe size $N = n^{O(\sqrt{\log n})}$, then for any $\varepsilon > 0$ there exists a $O(1)$ time and $O(n^{\varepsilon\sqrt{\log n}})$ static data structure for predecessor queries.*

This statement can be proven by simply constructing a trie with node degree $n^{\varepsilon\sqrt{\log n}-1}$. If $N = n^{c\sqrt{\log n}}$ for some constant c, then the trie height is $(1/\varepsilon)\cdot c = O(1)$, and the predecessor of an arbitrary x can be found in $O(1)$ time. Thus if $w \leq (2\log^{3/2} n + 2\log n)$, we can answer predecessor queries in $O(1)$ time using a $O(n^{\varepsilon\sqrt{\log n}})$ space data structure. In the rest of this paper we assume w.l.o.g. that $w \geq (2\log^{3/2} n + 2\log n)$

3 An $O(1)$ amortized time data structure

We start with a global overview of our algorithm. During the first stage of our algorithm $\sqrt{\log n}$ predecessors queries on keys $x_i \in [0, 2^b - 1]$ are reduced in a certain way to $\sqrt{\log n}$ predecessor queries in $[0, 2^{b/2\sqrt{\log n}} - 1]$. The first phase is implemented using the van Emde Boas [E77] construction described in Section 2. But by performing multiple membership queries, as described in Lemma 1, the key size of $\sqrt{\log n}$ elements can be reduced from b to $b/2^{\sqrt{\log n}}$ in $O(\sqrt{\log n})$ time. When the size of the keys is sufficiently reduced, the predecessors of all elements in the query set can be quickly found. During the second stage we find the predecessors of $\sqrt{\log n}$ elements from $[0, 2^{b/2\sqrt{\log n}} - 1]$. Since $2^{\sqrt{\log n}}$ elements can be now packed into one machine word, we can use the packed B-trees of Andersson and find the predecessor of an element of the query set in $O(\log n/\log(2^{\sqrt{\log n}})) = O(\sqrt{\log n})$ time. In our algorithm, we follow the same approach, but we find the predecessors of *all* elements of the query set *in parallel*. This allows us to achieve $O(\sqrt{\log n})$ time for $\sqrt{\log n}$ elements, or $O(1)$ time per query.

Simultaneous search for predecessors is also used in the data structure for batched queries of [A95]. In [A95] the query set consists of $O(\log n)$ elements, and the van Emde Boas data structure is applied to each element of the query set. The key size of all elements can be reduced to $w/\log n$ in $O(\log n \log\log n)$ time. After this, the predecessors of all elements can be found in $O(\log n)$ time simultaneously. In our approach, we reduce key sizes of all elements simultaneously, using multiple membership queries and the method described in Theorem 1. This allows us to achieve constant query time and reduce the space from $O(n^{\varepsilon \log n})$ to $O(n^{\varepsilon\sqrt{\log n}})$. Besides that, the size of the query set is also reduced from $O(\log n)$ to $O(\sqrt{\log n})$. It can be shown that this size of the query set is almost optimal, i.e. constant time per query cannot be achieved by a $O(n^{\sqrt{\log n}})$ space data structure if the query set contains $o(\sqrt{\log n/\log\log n})$ elements.

In the following lemma we show, how $\sqrt{\log n}$ queries can be answered in $O(\sqrt{\log n})$ time, if the word size is sufficiently larger than the key size, that is $w = \Omega(b \log n)$. Later in this section we will show that the same time bound can be achieved in the case $w = \Theta(b)$

Lemma 2 *If word size $w = \Omega(b \log n)$, where b is the size of the keys, there exists a data structure that answers $\sqrt{\log n}$ predecessor queries in $O(\sqrt{\log n})$ time, requires space $O(n^{2\sqrt{\log n}+2})$, and can be constructed in $O(n^{2\sqrt{\log n}+2})$ time.*

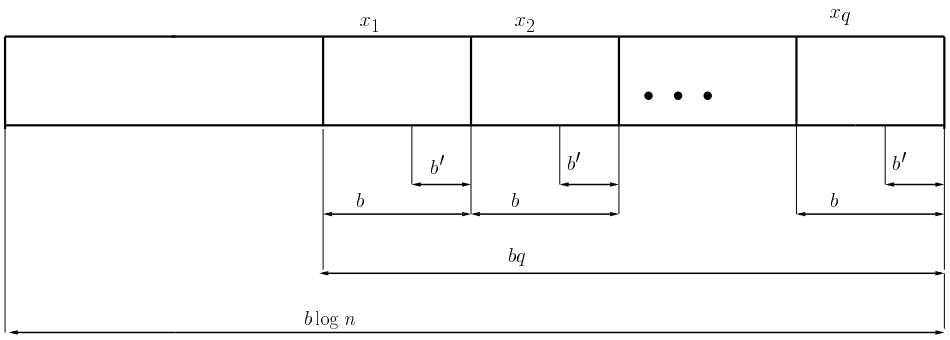

Fig. 1. The structure of word X^j

PROOF. Suppose we look for predecessors of elements $x_1, x_2, \ldots, x_p$ with $p = \sqrt{\log n}$. The algorithm consists of two stages :
`Stage 1. Range reduction.` During this stage the size of all keys is simultaneously reduced by multiple look-ups.
`Stage 2. Finding predecessors.` When the size of the keys is small enough, predecessors of all keys can be found by multiple comparisons in packed B-trees.
`Stage 1.` We start by giving a high level description; a detailed description will be given below. Since the word size w is $\log n$ times bigger than the key size b, $\sqrt{\log n}$ membership queries can be performed "in parallel" in $O(1)$ time. Therefore, it is possible to reduce the key size by a factor 2 in $O(1)$ time *simultaneously* for all elements of the query set.

The range reduction stage consists of $\sqrt{\log n}$ rounds. During round j the key size is reduced from $b/2^{j-1}$ to $b/2^j$. By b' we denote the key size during the current round; $<u>$ denotes the string of length b with value u.

Let $X = <x_1> \ldots <x_q>$ be a word containing all elements of the current query set. We set $X^1 = <x_1^1> \ldots <x_q^1>$, where $x_i^1 = x_i$, and we set $S_i^1 = S$ for $i = 1, \ldots, q$. During the first round we check whether prefixes of $x_1, x_2, \ldots, x_q$ of length $b/2$ belong to $H(S)$, i.e. we answer multiple membership query $(x_1^1)^{\mathrm{U}} \in H(S_1^1)?, (x_2^1)^{\mathrm{U}} \in H(S_2^1)?, \ldots, (x_q^1)^{\mathrm{U}} \in H(S_q^1)?$. If $(x_i)^{\mathrm{U}} \notin H(S_i^1)$, $H(S_i^1)$ is searched for the predecessor of $(x_i)^{\mathrm{U}}$, otherwise $S_{(x_i)^{\mathrm{U}}}$ must be searched for the predecessor of $(x_i)^{\mathrm{L}}$.

Now consider an arbitrary round j. At the beginning of the j-th round, we check whether some of the sets $S_1^j, S_2^j, \ldots, S_q^j$ contain less than five elements. For every i, such that $|S_i^j| \leq 4$, $< x_i^j >$ is deleted from the query set. After this we perform a multiple membership query $(x_1^j)^{\mathrm{U}} \in H(S_1^j)?, (x_2^j)^{\mathrm{U}} \in H(S_2^j)?, \ldots, (x_q^j)^{\mathrm{U}} \in H(S_q^j)?$. We set $X^{j+1} = < x_1^{j+1} > \ldots < x_q^{j+1} >$, where $x_i^{j+1} = (x_i^j)^{\mathrm{U}}$ if $x_i^j \notin H(S_i^j)$, otherwise $x_i^{j+1} = (x_i^j)^{\mathrm{L}}$. $S_i^{j+1} = H(S_i^j)$, if $x_i^j \notin H(S_i^j)$, and $S_i^{j+1} = (S_i^j)_{(x_i)^{\mathrm{U}}}$, if $x_i^j \in H(S_i^j)$.

`Detailed Description of Stage 1.` Words X^j, $j = 1, 2, \ldots, \sqrt{\log n}$, are of

size $b \log n$ (see Fig. 1). The rightmost qb bits of X^j consist of q components of size b, where $q \leq \sqrt{\log n}$; the leftmost $b \log n - qb$ bits of X^j contain no data at the beginning of the j-th round. In b' rightmost bits of the i-th component (i.e. in bits $ib+1, ib+2, \ldots ib+b'$ of X^j) the key x_i^j is stored; the other bits in the i-th component are auxiliary bits that contain no data.

Let *set tuple* $S_1^i, S_2^i, \ldots, S_q^i$ be an arbitrary combination of sets of elements of level i data structures (the same set can occur several times in a set tuple). For every $q \in [1, \sqrt{\log n}]$ and every set tuple $S_{i_1}^j, S_{i_2}^j, \ldots, S_{i_q}^j$, where $S_{i_k}^j$ are sets on level j, data structure $D(S_{i_1}^j, S_{i_2}^j, \ldots, S_{i_q}^j)$ is stored. $D(S_{i_1}^j, S_{i_2}^j, \ldots, S_{i_q}^j)$ consists of :

- mask $M(S_{i_1}^j, S_{i_2}^j, \ldots, S_{i_q}^j) \in [0, 2^q - 1]$. The $(q+1-t)$-th least significant bit of $M(S_{i_1}^j, S_{i_2}^j, \ldots, S_{i_q}^j)$ is 1, iff $|S_{i_t}^j| \leq 4$.
- word $MIN(S_{i_1}^j, S_{i_2}^j, \ldots, S_{i_q}^j) =< m_1 >< m_2 > \ \ldots \ < m_q >$, where $m_k = \min(S_{i_k}^j)$ is the minimal element in $S_{i_k}^j$
- if $M(S_{i_1}^j, S_{i_2}^j, \ldots, S_{i_q}^j) = 0$, data structure $L(S_{i_1}^j, S_{i_2}^j, \ldots, S_{i_q}^j)$, which allows to answer multiple queries $x_1 \in H(S_{i_1}^j)?, x_2 \in H(S_{i_2}^j)?, \ldots, x_q \in H(S_{i_q}^j)?$.
- Array DEL with q elements; $DEL[t]$ contains a pointer to data structure $D(S_{i_1}^j, \ldots, S_{i_{t-1}}^j, S_{i_{t+1}}^j, \ldots, S_{i_q}^j)$.
- Array $NEXT$ with less than $\prod_{k=1}^q 4|S_{i_k}^j|$ elements;

For every $F \in [0, 2^q - 1]$, list $LIST[F]$ is stored; $LIST[F]$ contains all indices i, such that the $(q+1-i)$-th least significant bit of F is 1. We store a one-to-one hash function $c : \mathcal{C} \rightarrow [0, 2^q - 1]$, where $\mathcal{C}$ is the set of integers $v \in [0, 2^{qb} - 1]$, such that the ib-th bit of v is either 1 or 0, and all other bits of v are 0. List $BACKLIST[F]$ contains all indices i, such that the $(q+1-i)b$-th least significant bit of $c^{-1}(F)$ is 1. We store pointers to $LIST[F]$ and $BACKLIST[F]$ for all F in arrays $LIST[]$ and $BACKLIST[]$ respectively.

Consider a round j, and suppose that the current set tuple is $S_1^j, S_2^j, \ldots, S_q^j$. For every element i of $LIST[M(S_1^j, S_2^j, \ldots, S_q^j)]$ we do the following:
1.x_i^j is extracted from X^j. We set $A := (X^j \gg (b(q-i)))AND(1^{b'})$ and find the predecessor of A in S_i^j. The predecessor of x_i^j in S_i^j can be found in constant time, since $|S_i^j| \leq 4$.
2. We delete x_i^j from X^j by $X^j := (X^j \text{ AND } 1^{(b(q-i))}) + ((X^j \gg (q-i+1)b) \ll (q-i)b)$, and decrement q by 1.
Then we extract all x_k^j such that $x_k^j < MIN(S_k^j)$. We perform a multiple comparison of X^j with $MIN(S_{i_1}^j, S_{i_2}^j, \ldots, S_{i_q}^j)$ and store the result in word C, such that the $(q+1-k)b$-th bit of C is 1 if and only if $x_k^j < \min(S_k^j)$. This multiple comparison can be performed in constant time. We compute $f = c(C)$ and process every element of $BACKLIST[f]$ in the same way as elements of $LIST[F]$ were processed.

Now, a multiple query $(x_1^j)^{\mathrm{U}} \in H(S_1^j)?, (x_2^j)^{\mathrm{U}} \in H(S_2^j)?, \ldots, (x_q^j)^{\mathrm{U}} \in H(S_q^j)?$. must be processed. We compute X^j AND $(0^{b-b'}1^{b'/2}0^{b'/2})^q$ and bit shift the result $b'/2$ bits to the right to get $(X^j)^{\mathrm{U}}$. The resulting word $(X^j)^{\mathrm{U}}$ consists of the prefixes of length $b'/2$ of elements $x_1^j, x_2^j, \ldots, x_q^j$. Using $(X^j)^{\mathrm{U}}$ and $L(S_1^j, S_2^j, \ldots, S_q^j)$, query $(x_1^j)^{\mathrm{U}} \in H(S_1^j)?, (x_2^j)^{\mathrm{U}} \in H(S_2^j)?, \ldots, (x_q^j)^{\mathrm{U}} \in H(S_q^j)?$ can be answered in $O(1)$ time. The result is stored in word R such that the $(q+1-i)b$-th least significant bit of R is 1, iff $(x_i^j)^{\mathrm{U}} \in H(S_i^j)$, and all other bits of R are 0. We also construct word $(X^j)^{\mathrm{L}}$ that consists of suffixes of $x_1^j, x_2^j, \ldots x_q^j$ of length $b'/2$. $(X^j)^{\mathrm{L}}$ is computed by $(X^j)^{\mathrm{L}} = X^j$ AND $(0^{b-b'/2}1^{b'/2})^q$. We compute the words $R' = (R \gg (b-1)) \times 1^b$ and $R'' = R'$ XOR 1^{qb}. Now we can compute $X^{j+1} = (X^{\mathrm{L}}$ AND $R'') + (X^{\mathrm{U}}$ AND $R')$.

The pointer to the next data structure can be computed in a similar way. Let $h_1, h_2, \ldots, h_q$ be hash functions for the sets $S_1^j, S_2^j, \ldots, S_q^j$. As shown in the proof of Lemma 1, word $P = h_1(x_1^j)h_2(x_2^j)\ldots h_q(x_q^j)$ can be computed in constant time. For every such P, we store in $NEXT[P]$ a pointer to data structure $D(S_1^{j+1}, S_2^{j+1}, \ldots, S_q^{j+1}))$, such that $S_i^{j+1} = H(S_i^j)$, if $x_i^j \notin H(S_i^j)$, and $S_i^{j+1} = (S_i^j)_{(x_i)^{\mathrm{U}}}$, if $x_i^j \in H(S_i^j)$. Array $NEXT$ has less than $\prod_{k=1}^q 4|S_{i_k}^j|$ elements.

After $\sqrt{\log n}$ rounds, the range of the key values is reduced to $[0, 2^{b/(2\sqrt{\log n})} - 1]$.

`Stage 2. Finding Predecessors` Now we can find the predecessors of elements using the approach of packed B-trees (cf. [H98],[A95]). Since more than $\sqrt{\log n}2^{\sqrt{\log n}}$ keys fit into a machine word, each of current queried values can be compared with $2^{\sqrt{\log n}}$ values from the corresponding data structure. Hence after at most $\sqrt{\log n}$ rounds the search will be completed. In this paper we consider an extension of the approach of packed B-trees for a simultaneous search in several data structures, called *a multiple B-tree*.

Let $p = \sqrt{\log n}$ and $t = 2^{\sqrt{\log n}}$. Consider an arbitrary combination of level p sets $S_{i_1}^p, \ldots, S_{i_q}^p$ and packed B-trees $T_{i_1}, T_{i_2}, \ldots, T_{i_q}$ for these sets. Nodes of T_{i_j} have degree $\min(2^{\sqrt{\log n}}, |S_{i_j}^p|)$. The root of a *multiple B-tree* contains all elements of the roots of packed B-trees for $S_{i_1}^p, \ldots, S_{i_q}^p$. Every node of the multiple B-tree that contains nodes $n_1, n_2, \ldots n_q$ has at most $(2^{\sqrt{\log n}})^q$ children, which correspond to all possible combinations of children of $n_1, n_2, \ldots n_q$ (only non-leaf nodes among $n_1, \ldots, n_q$ are considered). Thus a node of the *multiple B-tree* on the level k is an arbitrary combination of nodes of packed B-trees for sets $S_{i_1}^p, \ldots, S_{i_q}^p$ on level k. In every node v, word K_v is stored. If node v corresponds to nodes $v_1, v_2, \ldots, v_q$ of packed B-trees with values $v_1^1, \ldots, v_1^t, v_2^1, \ldots, v_2^t, \ldots, v_q^1, \ldots, v_q^t$ respectively, then $K_v = 0v_1^1 0v_1^2 \ldots 0v_1^t \ldots\ldots 0v_q^1 0v_q^2 \ldots 0v_q^t$. Values $v_i^1, \ldots, v_i^t$, for $i = 1, \ldots, q$, are stored in K_v in an ascending order. In every node we also store an array $CHILD$

with $2n$ elements. Besides that in every node v an array $DEL(v)[]$ and mask $M(v) \in [0, 2^{q+1}-1]$ are stored; they have the same purpose as the array DEL and mask M in the first stage of the algorithm: the $(q+1-k)$-th least significant bit of $M(v)$ is 1, iff the node of T_{i_k} stored in v is a leaf node. The height of the multiple B-tree is $O(\sqrt{\log n})$.

Now we show how every component of X can be compared with $2^{\sqrt{\log n}}$ values in constant time. Let $X = <x_1><x_2> \ldots <x_q>$ be the query word after the completion of Stage 1. Although the length of $<x_i>$ is b, the actual length of the keys is b', and b' is less than $b/(2^{\sqrt{\log n}})$. Let $s = 2^{\sqrt{\log n}}(b'+1)$, then $s \leq b$. We construct the word $X' = \prec x_1 \succ \prec x_2 \succ \ldots \prec x_q \succ$, where each $\prec x_i \succ$ consists of $2^{\sqrt{\log n}}$ copies of x_i divided by 0, and each copy is of length b', i.e.

$$\prec x_i \succ = \underbrace{0x_i \quad \overbrace{0x_i}^{b'+1 \text{ bits}} \quad \ldots 0x_i}_{2^{\sqrt{\log n}} \text{ times}}$$

To achieve this, we copy X, shift the copy $b'+1$ bits to the left, and add the result to X. The result is copied, shifted $2b'+2$ bits to the left, and so on. We repeat this $\sqrt{\log n}$ times to obtain $2^{\sqrt{\log n}}$ copies of each key value x_i.

To compare values stored in X' with values stored in node v, we compute $R = (K_v - X')\text{AND } W$, where W is a word every $(b'+1)$-th bit of which is 1, and all other bits are 0. Let $\mathcal{R}$ be the set of possible values of R. Since values $v_i^1, \ldots, v_i^t$ are sorted, $|\mathcal{R}| = (2^{\sqrt{\log n}})^{\sqrt{\log n}} = n$. Hence, a hash function $r : \mathcal{R} \rightarrow [1, 2n]$ (one for all nodes) can be constructed. The search continues in a node $v' = CHILD[r(R)]$. Since the height of multiple B-tree is $O(\sqrt{\log n})$, predecessors are found in $O(\sqrt{\log n})$ time.

Observe that if $b < 2^{\sqrt{\log n}}$, there is no need for Stage 2. This fact is further exploited in Theorem 3.

`Space Analysis` First we analyze the space used during the Stage 1. In an arbitrary data structure $D(S_{i_1}^j, S_{i_2}^j, \ldots, S_{i_q}^j)$, $L(S_{i_1}^j, S_{i_2}^j, \ldots, S_{i_q}^j)$ and array $NEXT$ use $O(q \prod_{k=1}^q 4|S_{i_k}^j|)$ space, mask $M(S_{i_1}^j, S_{i_2}^j, \ldots, S_{i_q}^j)$ uses constant space, and array DEL uses $O(q)$ space. Hence, $D(S_{i_1}^j, S_{i_2}^j, \ldots, S_{i_q}^j)$ uses $O(q \prod_{k=1}^q 4|S_{i_k}^j|)$ space. The total space for all data structures $D(S_{i_1}^j, S_{i_2}^j, \ldots, S_{i_q}^j)$ is

$$O(\sum_{\substack{i_1 \in [1,g], \\ \ldots, \\ i_q \in [1,g]}} q \prod_{k=1}^{q} 4|S_{i_k}^j|) \tag{1}$$

where g is the total number of sets S_i^j. Since $S_1^j + S_2^j + \ldots \leq n2^j$, the total number of terms in sum (1) does not exceed $(n2^j)^q$. Every product $q \prod_{k=1}^q 4|S_{i_k}^j|$ is

less than $n^q 2^{2q}$. Hence the sum (1) is smaller than $n^{2q}2^{(j+2)q}$. Summing up by $j = 1, \ldots, \sqrt{\log n}$, we get $\sum_{q=1}^{\sqrt{\log n}} \sum_{j=1}^{\sqrt{\log n}} n^{2q}2^{(j+2)q} \leq \sum_{q=1}^{\sqrt{\log n}} n^{2q}2^{(\sqrt{\log n}+3)q}$. The last expression does not exceed $n^{2\sqrt{\log n}}2^{(\sqrt{\log n}+3)\sqrt{\log n}+1} = O(n^{2\sqrt{\log n}+2})$. Therefore the total space used by all data structures in stage 1 is $O(n^{2\sqrt{\log n}+2})$.

Now consider a multiple B-tree for a set tuple $S^p_{i_1}, S^p_{i_2}, \ldots, S^p_{i_q}$. Every leaf in this multiple B-tree corresponds to some combination of elements from $S^p_{i_1}, S^p_{i_2}, \ldots, S^p_{i_q}$. Hence, the number of leaves is $O(\prod_{k=1}^q |S^p_{i_k}|)$, and the total number of nodes is also $O(\prod_{k=1}^q |S^p_{i_k}|)$. Using the same arguments as above, the total number of elements in all multiple B-trees does not exceed $\sum_{q=1}^{\sqrt{\log n}} n^{2q}2^{(p+2)q} = O(n^{2\sqrt{\log n}+2})$. Hence, the total space is $O(n^{2\sqrt{\log n}+2})$.

A data structure $D(S^j_{i_1}, S^j_{i_2}, \ldots, S^j_{i_q})$ used in stage 1 can be constructed in $O(\prod_{k=1}^q 4|S^j_{i_k}|)$ time. Hence, all $D(S^j_{i_1}, S^j_{i_2}, \ldots, S^j_{i_q})$ for fixed j and q can be constructed in $O(n^{2q}2^{(j+2)q})$, and all data structures for the stage 1 can be constructed in $O(n^{2\sqrt{\log n}+2})$ time. A multiple B-tree for set tuple $S^p_{i_1}, S^p_{i_2}, \ldots, S^p_{i_q}$ can be constructed in $O(\prod_{k=1}^q |S^p_{i_k}|)$ time. Therefore, all multiple B-trees can be constructed in $O(n^{2\sqrt{\log n}+2})$ time.
□

Now we consider the case when a machine word contains only b bits.

Theorem 1 *If word size $w = \Theta(b)$, where b is the size of the keys, there is a data structure that answers $\sqrt{\log n}$ predecessor queries in $O(\sqrt{\log n})$ time, requires space $O(n^{2\sqrt{\log n}+2})$, and can be constructed in $O(n^{2\sqrt{\log n}+2})$ time.*

PROOF. Using the van Emde Boas construction described in Section 2, the key size can be reduced from b to $b/\log n$ in $\log\log n\sqrt{\log n}$ time. However, we can speed-up the key size reduction by multiple queries. The preliminary key size reduction for elements $x_1, x_2, \ldots, x_q$ consists of $\log\log n+1$ rounds. During the i-th round the length of the keys b' is reduced from $b/2^{i-1}$ to $b/2^i$. Hence, during the i-th round $w/b' > 2^{i-1}$, and $2^{(i-1)/2}$ membership queries can be performed in constant time. Our range reduction procedure is similar to the range reduction procedure of Stage 1 of Lemma 2, but we do not decrease q if some data structure becomes small, and parameter q grows monotonically. Roughly speaking, the number of keys that are stored in a word and can be queried in constant time grows monotonically. For $q \leq 2^{(j-1)/2}$ and every $S^j_{i_1}, \ldots, S^j_{i_q}$, where $S^j_{i_k}$ are arbitrary sets on level j, data structure $D(S^j_{i_1}, S^j_{i_2}, \ldots, S^j_{i_q})$ described in Lemma 2 is stored.

The range reduction consists of $\log\log n + 1$ rounds. At the beginning of

round j, keys $x_1^j, \ldots, x_q^j$ are stored in $\sqrt{\log n}/2^{(j-1)/2}$ words. Let $t = 2^{(j-1)/2}$. For simplicity, we assume that all words X_i^j contain t keys during each round. Consider an arbitrary word $X_i^j = x_{(i-1)t+1}^j, x_{(i-1)t+2}^j, \ldots, x_{it}^j$, where $i = 1, \ldots, \sqrt{\log n}/2^{(j-1)/2}$. In the same way as in Lemma 2, words $(X_i^j)^{\mathrm{U}}$ and $(X_i^j)^{\mathrm{L}}$ can be computed. Using the corresponding data structure L, query $(x_1)^{\mathrm{U}} \in H(S_{i_1}^j)?, (x_2)^{\mathrm{U}} \in H(S_{i_2}^j)?, \ldots, (x_q)^{\mathrm{U}} \in H(S_{i_q}^j)?$ can be answered in constant time, and X^{j+1} can also be computed in constant time. At the end of round j, such that $j = 0(\mod 2)$, elements are regrouped. That is, we double the number of keys stored in one word. Observe that we cannot double the number of keys after each round, because we need space for q^2 keys in one word to perform q membership queries simultaneously. Since the key size has decreased by factor 4 during the two previous rounds, word X_i^j is of the form $0^{3b''}x_{(i-1)t+1}^j 0^{3b''}x_{(i-1)t+2}^j \ldots 0^{3b''}x_{it}^j$, where $b'' = b'/4$ and $x_k^j \in \{0,1\}^{b''}$. We construct for each X_i^j a word $\tilde{X}_i^j$ of the form $x_{(i-1)t+1}^j x_{(i-1)t+2}^j \ldots x_{it}^j$. First X_i^j is multiplied with $(0^{tb'-1}1)^t$ to get $\overline{X}_i^j$. Then we perform bitwise AND of $\overline{X}_i^j$ with a word $(0^{tb'}1^{b'})^t$ and store the result in $\hat{X}_i^j$. $\hat{X}_i^j$ is of the form $x_{(i-1)t+1}^j 0^{tb'+3b''} x_{(i-1)t+2}^j 0^{tb'+3b''} \ldots 0^{tb'+3b''} x_{it}^j$. We can obtain $\tilde{X}_i^j$ from $\hat{X}_i^j$: Firstly, we multiply $\hat{X}_i^j$ with $(10^{tb'+2b''-1})^{t-1}10^{tb'+3b''-1}$. Then, we bit shift the result $(t+1)(t-1)b' - (t-1)b''$ bits to the right and perform bitwise AND with $1^{tb''}$.

Finally, we double the number of keys in a word by setting $X_i^{j+1} = \tilde{X}_{2i}^j \ll tb' + \tilde{X}_{2i+1}^j$, for $i = 1, \ldots, \sqrt{\log n}/2^{(j-1)/2+1}$. Therefore after every second round the number of words decreases by factor 2. The total number of operations is limited by $2O(\sqrt{\log n}) \sum_{i=1}^{\lceil(\lceil \log \log n \rceil/2)\rceil} \frac{1}{2^{i-1}} = O(\sqrt{\log n})$.

Space requirement and construction time can be estimated in the same way, as in the proof of Lemma 2. When the key size is reduced to $k/\log n$, predecessors can be found using Lemma 2.
□

4 Other Results

In this section we describe several extensions and improvements of Theorem 1.

Theorem 2 *For $p(n) = O(\sqrt{\log n})$, there exists a data structure that answers $p(n)$ predecessor queries in time $O(\sqrt{\log n})$, uses space $O(n^{2p(n)+2})$, and can be constructed in time $O(n^{2p(n)+2})$*

Proof Sketch The proof is analogous to the proof of Theorem 1, but query set Q contains $p(n)$ elements. □

By setting $p(n) = (\varepsilon/2)\sqrt{\log n} - 1$ in Theorem 2, we obtain a data structure that answers $\sqrt{\log n}$ predecessor queries in time $O(\sqrt{\log n})$, uses space $O(n^{\varepsilon\sqrt{\log n}})$, and can be constructed in time $O(n^{\varepsilon\sqrt{\log n}})$ for any $\varepsilon > 0$.

In a similar way we can also obtain the following:

Corollary 1 *For any $\varepsilon > 0$ and $p(n) = O(\sqrt{\log n})$, there exists a data structure that answers $p(n)$ predecessor queries in time $O(\sqrt{\log n})$, uses space $O(n^{\varepsilon p(n)})$, and can be constructed in time $O(n^{\varepsilon p(n)})$.*

PROOF. We apply Theorem 2 to $p'(n) = (\varepsilon/2)p(n) - 1$

By setting $p(n) = \sqrt{\log n}$ in the above Corollary, we obtain a data structure that answers $\sqrt{\log n}$ predecessor queries in time $O(\sqrt{\log n})$, uses space $O(n^{\varepsilon\sqrt{\log n}})$, and can be constructed in time $O(n^{\varepsilon\sqrt{\log n}})$ for any $\varepsilon > 0$.

If the key size b is such that $\log b = o(\sqrt{\log n})$ (i.e. $\log\log N = o(\sqrt{\log n})$), then a more space efficient data structure can be constructed.

Theorem 3 *For $p(N) = O(\log\log N)$, there exists a data structure that answers $p(N)$ predecessor queries in time $O(\log\log N)$, uses space $O(n^{2p(N)+2})$, and can be constructed in time $O(n^{2p(N)+2})$.*

Proof Sketch The proof is analogous to the proof of Theorem 1, but query set Q contains $p(N)$ elements. We apply $\log\log N$ rounds of the Stage 1 (range reduction stage) from the proof of Theorem 1. After this, the current key size b' equals to 1 for all elements of the query set, and predecessors can be found in a constant time. □

Corollary 2 *For any $\varepsilon > 0$ and $p(N) = O(\log\log N)$, there exists a data structure that answers $p(N)$ predecessor queries in time $O(\log\log N)$, uses space $O(n^{\varepsilon p(N)})$, and can be constructed in time $O(n^{\varepsilon p(N)})$.*

If $p(N) = \log\log N$, we obtain a data structure that answers $\log\log N$ predecessor queries in time $O(\log\log N)$, uses space $O(n^{\varepsilon\log\log N})$, and can be constructed in time $O(n^{\varepsilon\log\log N})$.

The lower bound of [BF02] and [S03] can be extended to data structures with space $O(n^{\log^{O(1)} n})$ (s. Appendix B for the proof of this statement). Thus the worst case $\Omega(\sqrt{\log n/\log\log n})$ lower bound can be surpassed for $O(n^{p(n)})$ predecessor data structures where $p(n) = w(\sqrt{\log\log n})$ and $p(n) = O(\log n)$ if batched queries are used. Since $k > 1$ queries cannot be answered faster than 1 query, this means that the size of the query set in our data structure is almost optimal if constant time per query must be achieved.

5 Conclusion

In this paper we have presented data structures for batched predecessor queries. These data structures allow us to answer predecessor queries faster than the lower bound of [BF02] at the cost of higher space requirements and batch processing.

We can use the following simple trick to answer a batch of $n/\log^c n$ predecessor queries for any constant c in $O(\log\log n)$ time per query using linear space. Suppose that n elements are stored in data structure A in sorted order. We divide A into $\lceil n/\log^c n\rceil$ groups $A_1, A_2, A_{\lceil n/\log^c n\rceil}$; all groups, but the last one, are of size $\log^c n$. We select one representative from each subgroup and store them in a sorted list A'. Using an integer sorting algorithm (e.g. [H02]), we can sort n elements of query set Q in $O(n\log\log n)$ time, then merge them with elements of A', and find predecessors of elements from Q in A' in $O(n)$ time. Once we know the predecessor of some element $q \in Q$ in A', we can find its predecessor in A in $O(\log\log n)$ time. Using the same trick, we can answer e.g. $n/2^{\sqrt{\log n}}$ queries in $O(\log^{1/4} n)$ time: we divide A into groups of size $2^{\sqrt{\log n}}$ and store the predecessor data structure from [BF02] or [A95] for each group.

An existence of a linear (or polynomial) space data structure, which can answer $p(n)$ queries when $p(n)$ is significantly smaller than n (say, $p(n) = \sqrt{n}$) in time $o(\sqrt{\log n/\log\log n})$ per query is an interesting open problem.

Acknowledgment

The authors thank the anonymous reviewer for an observation that the lower bound for predecessor queries can be extended to the case of $n^{\log^{O(1)}}$ space. We also thank Pranab Sen for drawing our attention to [PT06].

References

[AL62] G. M. Adelson-Velskii, E.M. Landis, *An algorithm for the organization of information*, Dokladi Akademii Nauk SSSR, 146(2):1259-1262, 1962.

[A88] M. Ajtai, *A lower bound for finding predecessors in Yao's call probe model*, Combinatorica 8(3): 235-247 (1988).

[AFK84] M. Ajtai, M. L. Fredman, J. Komlòs, *Hash Functions for Priority Queues*, Information and Control 63(3): 217-225 (1984).

[ABR01] S. Alstrup, G. S. Brodal, T. Rauhe, *Optimal static range reporting in one dimension*, STOC 2001, pp. 476-482.

[A95] A. Andersson, *Sublogarithmic Searching without Multiplications*, FOCS 1995, pp. 655-663.

[A96] A. Andersson, *Faster Deterministic Sorting and Searching in Linear Space*, FOCS 1996, pp. 135-141

[AHNR95] A. Andersson, T. Hagerup, S. Nilsson, R. Raman, *Sorting in linear time?* STOC 1995, pp. 427-436.

[AT00] A. Andersson, M. Thorup, *Tight(er) worst-case bounds on dynamic searching and priority queues*, STOC 2000, pp. 335-342.

[AT02] A. Andersson, M. Thorup, *Dynamic Ordered Sets with Exponential Search Trees*, The Computing Research Repository (CoRR), cs.DS/0210006: (2002). Available at `http://arxiv.org/abs/cs.DS/0210006`.

[BF02] P. Beame, F. E. Fich, *Optimal Bounds for the Predecessor Problem and Related Problems*, J. Comput. Syst. Sci. 65(1): 38-72 (2002).

[BCKM01] A. Brodnik, S. Carlsson, J. Karlsson, J. I. Munro, *Worst case constant time priority queue*, SODA 2001, pp. 523-528.

[CW79] L. Carter, M. N. Wegman, *Universal Classes of Hash Functions.* J. Comput. Syst. Sci. 18(2): 143-154 (1979).

[E77] P. van Emde Boas, *Preserving Order in a Forest in Less Than Logarithmic Time and Linear Space*, Inf. Process. Lett. 6(3): 80-82 (1977)

[EKZ77] P. van Emde Boas, R. Kaas, E. Zijlstra, *Design and Implementation of an Efficient Priority Queue*, Mathematical Systems Theory 10: 99-127 (1977)

[FW93] M. L. Fredman, D. E. Willard, *Surpassing the Information Theoretic Bound with Fusion Trees* , J. Comput. Syst. Sci. 47(3): 424-436 (1993).

[H98] T. Hagerup, *Sorting and Searching on the Word RAM*, STACS 1998, pp. 366-398.

[H02] Y. Han, *Deterministic sorting in O(n log log n) time and linear space*, STOC 2002,pp. 602-608

[GL01] B. Gum, R. Lipton, *Cheaper by the Dozen: Batched Algorithms.* 1st SIAM International Conference on Data Mining, 2001
Available at `http://www.math.grin.edu/ gum/papers/batched/`

[M84] K. Mehlhorn, *Data Structures and Algorithms 1: Sorting and Searching*, Springer 1984.

[Mil94] P. B. Miltersen, *Lower bounds for union-split-find related problems on random access machines*, STOC 1994, pp. 625-634.

[Mil99] P. B. Miltersen, *Cell probe complexity - a survey*, Pre-Conference Workshop on Advances in Data Structures at the 19th FSTTCS, 1999 Availble at http://citeseer.ist.psu.edu/miltersen99cell.html

[MNSW98] P. B. Miltersen, N. Nisan, S. Safra, A. Wigderson, *On Data Structures and Asymmetric Communication Complexity* J. Comput. Syst. Sci. 57(1): 37-49 (1998).

[PS80] W. J. Paul, S. Simon, *Decision Trees and Random Access Machines*, International Symposium on Logik and Algorithmic, Zürich, pp 331-340, 1980.

[PT06] M. Pătraşcu, M. Thorup, *Time-Space Trade-Offs for Predecessor Search*, to be published in STOC'06.

[S03] P. Sen, *Lower bounds for predecessor searching in the cell probe model*, IEEE Conference on Computational Complexity 2003, pp. 73-83.

[SV03] P. Sen, S. Venkatesh, *Lower bounds for predecessor searching in the cell probe model*, CoRR cs.CC/0309033: (2003).

[W83] D. E. Willard, *Log-Logarithmic Worst-Case Range Queries are Possible in Space Theta(N)*, Inf. Process. Lett. 17(2): 81-84 (1983)

[W84] D. E. Willard, *New Trie Data Structures Which Support Very Fast Search Operations*, J. Comput. Syst. Sci. 28(3): 379-394 (1984).

[Y81] A. C.-C. Yao *Should Tables Be Sorted?*, J. ACM 28(3): 615-628 (1981).

Appendix A.
Proof of the Lower Bound for the Predecessor Problem

In this Appendix we extend the lower bound of [BF02] and [S03] to data structures with $O(n\sqrt{\log n})$ space. Our proof is a straightforward extension of the lower bound of Sen and Venkatesh [SV03]. This result is obtained simply by changing the parameters in the proof of Theorem 1 in [SV03]. To make this paper self-sufficient, we provide the proof and all necessary definitions in this Appendix. A detailed description of the communication complexity, cell probe complexity, and their applications to the lower bound proofs can be found in e.g., [MNSW98], [Mil99], [SV03].

Recently, Pătraşcu and Thorup [PT06] generalized the lower bounds of [BF02] and [S03]. The result of this Appendix can also be obtained from [PT06].

Definition 1 *An (s, w, t) cell probe scheme for a static data structure problem $f : D \times Q \to A$ has two components: a* storage scheme *and a* query scheme *The storage scheme stores the data $d \in D$ as a table $T[d]$ of s cells with w bits each. Given a query $q \in Q$, the query scheme computes $f(d, q)$ by making at most t probes to $T[d]$, where each probe reads one cell at a time, and the probes can be adaptive.*

In the above definition D is the domain of possible data, Q is the domain of possible queries, and A is the domain of possible answers.

Definition 2 *A $[t; l_1, l_2, \ldots, l_t]^A$ ($[t; l_1, l_2, \ldots, l_t]^B$) communication protocol is the communication protocol in which Alice (Bob) starts the communication, the i-th message is l_i bits long, and the communication goes on for t rounds.*

Definition 3 *A $(t, a, b)^A$ ($(t, a, b)^B$) communication protocol is a $[t; l_1, l_2, \ldots, l_t]^A$ ($[t; l_1, l_2, \ldots, l_t]^B$) communication protocol, where Alice (Bob) starts, and $l_i = a$ for i odd and $l_i = b$ for i even ($l_i = b$ for i odd and $l_i = a$ for i even).*

Definition 4 *In the rank parity communication game $PAR_{p,q}$ Alice is given a bit string x of length p, Bob is given a set S of bit strings of length p, and they have to communicate and decide, whether the rank of x in S (i.e. the cardinality of the set $\{y \in S | y \leq x\}$) is odd or even. In the communication game $PAR^{(k),A}_{p,q}$, Alice is given k bit strings $x_1, x_2, \ldots, x_k$ each of length p, Bob is given a set S of bit strings of length p, $|S| \leq q$, an index i, such that $1 \leq i \leq k$, and copies of $x_1, x_2, \ldots, x_{i-1}$; they have to communicate and decide, whether the rank of x_i in S is odd or even. In the communication game $PAR^{(k),B}_{p,q}$, Alice is given a bit string of length p and an index i, such that $1 \leq i \leq k$, Bob is given k sets $S_1, S_2, \ldots, S_k$ of bit strings of length p, $|S_j| \leq q$, $1 \leq j \leq k$, and they have to communicate and decide, whether the*

rank of x in S_i is odd or even.

The (N, n) static predecessor problem is to store n elements from a universe of size N in a predecessor data structure.

Fact 2 *Let m be a positive integer such that m is a power of 2. Suppose there is a $(n^{\log^{O(1)} n}, \log^{O(1)} N, t)$ randomized cell probe scheme for the (N, n) static predecessor problem with error probability δ. Then the rank parity communication game $PAR_{\log N, n}$ has a $(2t + O(1), (\log n)^{O(1)}, \log^{O(1)} N)^A$ private coin randomized protocol with error probability δ.*

Proof Sketch Suppose that the set S is stored in the data structure. Using hash functions, we can determine for each element $y \in S$ its parity in $O(1)$ cell probes and $O(n)$ space. Thus, if there is a $(n^{O(\log^{O(1)} n)}, \log^{O(1)} N, t)$ cell probe scheme for the predecessor problem, there is also a $(n^{(\log^{O(1)} n)}, \log^{O(1)} N, t + O(1))$ scheme for the rank parity problem. Using [Mil94], this scheme can be converted into a $(2t + O(1), O(\log^{O(1)} n), \log^{O(1)} N)$ communication protocol for the rank parity problem.

The proofs of the following facts can be found in e.g., [SV03].

Fact 3 *Let k, p be positive integers such that $k|p$. A communication protocol with Alice starting for $PAR_{p,q}$ gives us a communication protocol with Alice starting for $PAR^{(k),A}_{p/k,q}$ with the same message complexity, number of rounds, and error probability.*

Fact 4 *Let k, q be positive integers such that $k|q$, and k is a power of 2. A communication protocol with Bob starting for $PAR_{p,q}$ gives us a communication protocol with Bob starting for $PAR^{(k),B}_{p-\log k-1,q/k}$ with the same message complexity, number of rounds, and error probability.*

The following Lemma is an improvement of the Round Elimination Lemma from [MNSW98] and is proven in [SV03].

Lemma 3 *Suppose $f : X \times Y \to Z$ is a function. Suppose the communication game $f^{(n),A}$ has a $[t; l_1, l_2 \ldots, l_t]^A$ public coin randomized protocol with error less than δ. Then there is a $[t-1; l_2, \ldots, l_t]^B$ public coin randomized protocol for f with error less than $\delta + (1/2)(2l_1 \ln 2/n)^{1/2}$.*

Theorem 4 *Suppose there is a $(n^{O(\sqrt{\log n})}, (\log N)^{O(1)}, t)$ randomized cell probe scheme for the (m, n) static predecessor problem with error probability less than $1/3$. Then, $t = \Omega(\frac{\log\log N}{\log\log\log N})$ and $t = \Omega(\sqrt{\frac{\log n}{\log\log n}})$.*

PROOF. The proof is analogous to the proof of Theorem 1 in [SV03]; only

certain parameters are chosen differently.

We choose $n = 2^{(\log\log N)^2/\log\log\log N}$. Let $c_1 = (2\ln 2)6^2$, $c_2, c_3 \geq 2c_4$. Let $a = (c_2 \log n)^{c_4}$, $b = (\log N)^{c_3}$, and $t = \frac{\log\log N}{(c_1+c_2+c_3)\log\log\log N}$.

By Fact 2,to prove the desired lower bounds it suffices to show that there is no $(2t, a, b)^A$ communication protocol for the rank parity communication game with error less than $1/3$.

Given a $(2t, a, b)^A$ communication protocol for $PAR_{\log N, n}$ with error less than δ, we can get a $(2t, a, b)$ communication protocol for $PAR^{(c_1at^2),A}_{\frac{\log N}{c_1at^2}, n}$ with error probability at most δ by Fact 3. Using Lemma 3, we get a $(2t-1, a, b)^B$ public coin communication protocol for $PAR_{\frac{\log N}{c_1at^2}, n}$ with error probability $\delta + (12t)^{-1}$. By Fact 4, we get a $(2t-1, a, b)^B$ communication protocol for $PAR^{(c_1bt^2),B}_{\frac{\log N}{c_1at^2} - \log(c_1bt^2) - 1, \frac{n}{c_1bt^2}}$ with error at most $\delta + (12t)^{-1}$. Since $\frac{\log N}{2c_1at^2} \geq \log(c_1bt^2) + 1$, this implies a $(2t-1, a, b)^B$ communication protocol for $PAR^{(c_1bt^2),B}_{\frac{\log N}{2c_1at^2}, \frac{n}{c_1bt^2}}$ with error at most $\delta + (12t)^{-1}$. We use Lemma 3 again, and get a $(2t-2, a, b)^A$ communication protocol for $PAR_{\frac{\log N}{2c_1at^2}, \frac{n}{c_1bt^2}}$ with error probability at most $\delta + 2(12t)^{-1}$.

We repeat the above steps t times and obtain a $(0, a, b)^A$ public coin randomized protocol for $PAR_{\frac{\log N}{(2c_1at^2)^t}, \frac{n}{(c_1bt^2)^t}}$ with error at most $\delta + 2t(12t)^{-1}$. Since $\delta < 1/3$, $\delta + 2t(12t)^{-1} < 1/2$. Besides that, $\frac{n}{(c_1bt^2)^t} = n^{\Omega(1)}$, and $\frac{\log N}{(2c_1at^2)^t} = \frac{\log N}{(2c_1(\log n)^{3/2}t^2)^t}$. Since $t = \frac{\log\log N}{(c_1+c_2+c_3)\log\log\log N}$ and $n = 2^{(\log\log N)^2/\log\log\log N}$, $(\log n)^{c_4}t^2 = O((\log\log N)^{c_4+2}) = O(2^{(c_4+2)\log\log\log N})$. Therefore $(2c_1at^2)^t = O(2^{(c_4+2)\log\log\log N \frac{\log\log N}{(c_1+c_2+c_3)\log\log\log N}} = O(2^{(c_4+2)\log\log N/(c_1+c_2+c_3)} = (\log N)^{c'}$ for some $c' < 1$, and $\frac{\log N}{(2c_1at^2)^t} = (\log N)^{\Omega(1)}$.

Hence, we obtain a zero-round protocol for a non-trivial communication problem $PAR_{\frac{\log N}{(2c_1at^2)^t}, \frac{n}{(c_1bt^2)^t}}$ with error less than $1/2$, which is a contradiction.

Appendix B.
Proof of Lemma 1

Lemma 1 *Given sets $S_1, S_2, \ldots, S_q$ such that $|S_i| = n_i$, $S_i \subset [0, 2^b - 1]$, and $q \leq \sqrt{\frac{w}{b}}$, there is a data structure that uses $O(bq\prod 2^{\lceil \log n_i \rceil + 1})$ bits, can be constructed in $O(q\prod 2^{\lceil \log n_i \rceil + 1})$ time, and answers q queries $p_1 \in S_1?, p_2 \in S_2?, \ldots, p_q \in S_q?$ in $O(1)$ time.*

PROOF. Let $r_i = \lceil \log n_i \rceil$. Following the presentation in [BF02], we construct two-level hash function $h_i : [0, 2^k - 1] \rightarrow [0, 2^{r_i+1} - 1]$ which are one-to-one on S_i. It is possible to find constants $a_i, a_{i,j_i}, p_{i,j_i}$, and r_{i,j_i}, for $i = 1, \ldots, q$ and $j_i = 0, \ldots, 2^{r_i+1} - 1$, and construct functions f_i, g_{i,j_i} and h_{i,j_i}, for $i = 1, \ldots, q$ and $j_i = 0, \ldots, 2^{r_i+1} - 1$, such that:
$f_i(x) = a_i x \mod 2^k \div 2^{k-1-r}$
$g_{i,j_i}(x) = a_{i,j_i} x \mod 2^k \div 2^{k-r_{i,j_i}}$
$h_i(x) = p_{i,f(x)} + g_{i,f(x)}(x)$
and functions h_i are one-to-one on S_i. Furthermore, we can construct four arrays A, R, P, and M, where:
$A[j_1, j_2, \ldots, j_q] = <a_{q,j_q}><0> \ldots <0><a_{1,j_1}>$
$R[j_1, j_2, \ldots, j_q] = <2^{r_q,j_q}><0> \ldots <0><2^{r_1,j_1}>$
$P[j_1, j_2, \ldots, j_q] = p_{1,j_1} \ldots p_{q,j_q}$
$M[[j_1, j_2, \ldots, j_q] = <x_1> \ldots <x_q>$, such that either $h_i(x_i) = j_i$ and $x_i \in S_i$, or $h_i(x_i) \neq j_i$ and $j_i \notin h_i(S_i)$.
Arrays A, R, P and M contain $\prod 2^{\lceil \log n_i \rceil + 1}$ elements. Given a word $Z = <z_1> \ldots <z_q>$, the string $H = h(z_1) \ldots h(z_q)$ can be computed in $O(1)$ time, as described in Lemma 4.1 in [BF02]. Now we can compare the b-bit components of $M[H]$ and Z. To do this, we compute $E = (M[H] \text{ OR } Z) \text{ AND } (01^{b-1})^q$, subtract E from $(10^{b-1})^q$ to get E'. Finally, compute AND of E' with $(10^{b-1})^q$ and $\tilde{E} = \text{NOT}(M[H] \text{ AND } Z)$. In the resulting string the $b(q + 1 - i)$-th bit is 1 if and only if $z_i \in S_i$ and all other bits are always 0.

Constants $a_i, a_{i,j_i}, p_{i,j_i}$, and r_{i,j_i}, and functions f_i, g_i, and h_i can be constructed in $O(\sum_{i=1}^{q} b 2^{2r_i})$ time. The arrays can be constructed in $O(q \prod 2^{\lceil \log n_i \rceil + 1})$ time. $\square$