Predecessor Queries in Constant Time?

Marek Karpinski * Yakov Nekrich †

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 $o(N)$ space and $O(1)$ amortized time data structure for arbitrary N and 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], [FW94], [A95], [H98], [AT00], [BF02], [BCKM01]. In this paper we present a static data structure that supports predecessor queries in $O(1)$ amortized time.

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

*Dept. of Computer Science, University of Bonn. E-mail marek@cs.uni-bonn.de. Work partially supported by a DFG grant, Max-Planck Research Prize, and IST grant 14036 (RAND-APX).

†Dept. of Computer Science, University of Bonn. E-mail yasha@cs.uni-bonn.de. Work partially supported by IST grant 14036 (RAND-APX).

proves that $\lceil \log n \rceil$ comparisons are necessary. This lower bound had for a long time been believed to be also the lower bound for the integer predecessor problem. 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 [FW94], 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]. 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 $\Omega(\sqrt{\log n/\log\log n})$ query time in the worst case. In [BF02] the authors also presented a matching upper bound, and transformed it 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].

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]). 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.

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 $\log n$ queries in time $O(\log n \log\log n)$. 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 RAM model with word size b, so that the size of the universe $N = 2^b$. To the best of our knowledge, this is the first algorithm that uses $o(N)$ space and words with $O(\log N)$ bits, and achieves $O(1)$ amortized query time.

If the universe is bounded (e.g., $\log\log N = o(\sqrt{\log n})$), our approach 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 and constant amortized time data structure.

The main idea of our method is to search in 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.

After some preliminary definitions in Section 2, we give an overview of our method in Section 3. In Section 3 an $O(n^{2\sqrt{\log n}+2})$ space and $O(1)$ amortized time data structure 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, 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 $|S_i| = n$, $S_i \subset [0, 2^b - 1]$, and $q \leq \sqrt{\frac{w}{b}}$, there is a data structure that uses $O(bq\prod_{i=1}^{q} 2^{\lceil \log n_i \rceil + 1})$ bits, can be constructed in $O(q\prod_{i=1}^{q} 2^{\lceil \log n_i \rceil + 1})$ 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.*

This Lemma is a straightforward extension of Lemma 4.1 in [BF02]. 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 (slightly modified) 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. Both 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 tree 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 $x^u \in H(S)$, we look for a predecessor of x^l in D_{x^u}. If x^l is smaller than all elements in D_{x^u}, the predecessor of x is $pred(x^u.min)$. If $x^u \notin H(S)$, the predecessor of x is $m.max$, where m is the node in T corresponding to the predecessor of x^u in D_0. 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 then 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]).

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

We start with 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 in spirit of [BF02] we can reduce the size of the keys for all elements of the query set in parallel. During the second stage we find the predecessors of $\sqrt{\log n}$ elements from $[0, 2b/2^{\sqrt{\log n}} - 1]$. 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. 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 $O(1)$ amortized time.

The main idea of the algorithm presented in this paper is to search for predecessors of $\sqrt{\log n}$ elements *simultaneously*. 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. If key size $b < w/2^{\sqrt{\log n}}$, the packed B-tree of degree $2^{\sqrt{\log n}}$ can be used to find the predecessor of a single element in $O(\sqrt{\log n})$ time. In our algorithm, we use a similar approach to find predecessors of *all* $\sqrt{\log n}$ elements in $O(\sqrt{\log n})$ time.

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 large, 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.*

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 $(x_1^1)^u \in H(S_1^1)?, (x_2^1)^u \in H(S_2^1)?, \ldots, (x_q^1)^u \in H(S_q^1)?$. If $(x_i)^u \notin H(S_i^1)$, $H(S_i^1)$ is searched for the predecessor of $(x_i)^u$, otherwise $S_{(x_i)^u}$ must be searched for the predecessor of $(x_i)^l$.

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 $(x_1^j)^u \in H(S_1^j)?, (x_2^j)^u \in H(S_2^j)?, \ldots, (x_q^j)^u \in H(S_q^j)?$. We set $X^{j+1} = <x_1^{j+1}> \ldots <x_q^{j+1}>$, where $x_i^{j+1} = (x_i^j)^u$ if $x_i^j \notin H(S_i^j)$, otherwise $x_i^{j+1} = (x_i^j)^l$. $S_i^{j+1} = H(S_i^j)$, if $x_i^j \notin H(S_i^j)$, and $S_i^{j+1} = (S_i^j)_{(x_i)^u}$, if $x_i^j \in H(S_i^j)$.

`Detailed Description of Stage 1.` Words X^j consist of q words of size b. 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 :

1. 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 $|S_{i_t}^j| \leq 4$.

2. 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 $S_{i_k}^J$

3. 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 $x_1 \in H(S_{i_1}^j)?, x_2 \in H(S_{i_2}^j)?, \ldots, x_q \in H(S_{i_q}^j)?$.

4. Array DEL with q elements; $DEL[t]$ contains a pointer to data structure
$D(S_{i_1}^j, \ldots, S_{i_{t-1}}^j, S_{i_{t+1}}^j, \ldots, S_{i_q}^j)$.

5. Array $NEXT$ with less than $\prod_{k=1}^{q} 4|S^j_{i_k}|$ 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 $\mathcal{C}$ 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.

Consider a round j, and suppose that the current set tuple is $S^j_1, S^j_2, \ldots, S^j_q$. For every element i of $LIST[M(S^j_1, S^j_2, \ldots, S^j_q)]$ we do the following:
1.x^j_i is extracted from X^j. We set $A := (X^j \gg (b(q-i-1)))AND(1^{b'})$ and find the predecessor of A in S^j_i. The predecessor of x^j_i in S^j_i can be found in constant time, since $|S^j_i| \leq 4$.
2. We delete x^j_i from X^j by $X^j := (X^j \text{ AND } 1^{(b(q-i-1))}) + ((X^j \gg (q-i)b) \ll (q-i-1)b)$, and decrement q by 1.
Then we extract all x^j_i such that $x^j_k < MIN(S^j_{i_k})$. We perform a multiple comparison of X^j with $MIN(S^j_{i_1}, S^j_{i_2}, \ldots, S^j_{i_q})$ and store the result in word C, such that the $(q+1-k)b$-th bit of C is 1 if and only if $x^j_k < \min(S^j_{i_k})$. Details will be given in the full version of this papers. 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 $(x^j_1)^u \in H(S^j_1)?, (x^j_2)^u \in H(S^j_2)?, \ldots, (x^j_q)^u \in H(S^j_q)?$. 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 $(X^j)^u$. The resulting word $(X^j)^u$ consists of the prefixes of length $b'/2$ of elements $x^j_1, x^j_2, \ldots, x^j_q$. Using $(X^j)^u$ and $L(S^j_1, S^j_2, \ldots, S^j_q)$, query $(x^j_1)^u \in H(S^j_1)?, (x^j_2)^u \in H(S^j_2)?, \ldots, (x^j_q)^u \in H(S^j_q)?$ can be answered in $O(1)$ time. The result is stored in word R such that $(q+1-i)b$-th least significant bit of R is 1, iff $(x^j_i)^u \in H(S^j_i)$, and all other bits of R are 0. We also construct word $(X^j)^l$ that consists of suffixes of $x^j_1, x^j_2, \ldots x^j_q$ of length $b'/2$. $(X^j)^l$ is computed by $(X^j)^l = X^j$ 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 $X^{j+1} = (X^l \text{ AND } R'') + (X^u \text{ AND } R')$.

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 $S^j_1, S^j_2, \ldots, S^j_q$. As shown in the proof of Lemma 1, word $P = h_1(x^j_1)h_2(x^j_2)\ldots h_q(x^j_q)$ can be computed in constant time. For every such P, we store in $NEXT[P]$ a pointer to data structure $D(S^{j+1}_1, S^{j+1}_2, \ldots, S^{j+1}_q))$, such that $S^{j+1}_i = H(S^j_i)$, if $x^j_i \notin H(S^j_i)$, and $S^{j+1}_i = (S^j_i)_{(x_i)^u}$, if $x^j_i \in H(S^j_i)$. Array $NEXT$ has less than $\prod_{k=1}^{q} 4|S^j_{i_k}|$

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 $t = 2^{\sqrt{logn}}$. Consider an arbitrary combination of level p sets $S^p_{i_1}, \ldots, S^p_{i_q}$ and packed B-trees $T_{i_1}, T_{i_2}, \ldots, T_{i_q}$ for these sets. Nodes of T_{i_j} have degree $\min(2^{\sqrt{\log n}}, |S^p_{i_j}|)$. The root of a *multiple B-tree* contains all elements of the roots of packed B-trees for $S^p_{i_1}, \ldots, S^p_{i_q}$. Every node of the multiple B-tree that contains nodes $n_1, n_2, \ldots n_q$ has $(2^{\sqrt{\log n}})^q$ 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 $S^p_{i_1}, \ldots, S^p_{i_q}$ 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 $v^1_1, \ldots, v^t_1, v^1_2, \ldots, v^t_2, v^1_q, \ldots, v^t_q$ respectively, then $K_v = 0v^1_1 0v^2_1 \ldots 0v^t_1 \ldots\ldots 0v^1_q 0v^2_q \ldots 0v^t_q$. Values $v^1_i, \ldots, v^t_i$, 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 $\sqrt{\log n}$ rounds 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 $s = 2^{\sqrt{\log n}}(b'+1)$, $s < b$. We construct the word $X' = \prec x_1 \succ \prec x_2 \succ \ldots \prec x_q \succ$, where $\prec x_i \succ = (0 \ll x_i \gg)^{2^{\sqrt{\log n}}}$, and $\ll x_i \gg$ is a string of length b' with value x_i. 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 $r : \mathcal{R} \to [1, 2n]$ (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.

`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 $O(\prod_{k=1}^{q} 4|S_{i_k}^j|)$ 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 $O(\prod_{k=1}^{q} 4|S_{i_k}^j|)$ space. The total space for all data structures $D(S_{i_1}^j, S_{i_2}^j, \ldots, S_{i_q}^j)$ is

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

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 $\prod_{k=1}^{q} 4|S_{i_k}^j|$ is less than $n^q 2^{2q}$. Hence the sum (1) is smaller than $n^{2q}2^{(j+2)q}$. Summing up by $j = 1, \ldots, \sqrt{\log n}$ and $q = 1, \ldots, \sqrt{\log n}$, 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 $S_{i_1}^p, S_{i_2}^p, \ldots, S_{i_q}^p$. Every leaf in this multiple B-tree corresponds to some combination of elements from $S_{i_1}^p, S_{i_2}^p, \ldots, S_{i_q}^p$. Hence, the number of leaves is $O(\prod_{k=1}^{q} |S_{i_k}^p|)$, and the total number of nodes is also $O(\prod_{k=1}^{q} |S_{i_k}^p|)$. Using the same arguments as above, the total number of elements in all multiple B-trees can be estimated as $\sum_{q=1}^{\sqrt{\log n}} n^{2q}2^{(p+2)q} = O(n^{2\sqrt{\log n}+2})$. Hence, the total space is $O(n^{2\sqrt{\log n}+2})$.

A data structure $D(S_{i_1}^j, S_{i_2}^j, \ldots, S_{i_q}^j)$ used in stage 1 can be constructed in $O(\prod_{k=1}^{q} 4|S_{i_k}^j|)$ time. Hence, all $D(S_{i_1}^j, S_{i_2}^j, \ldots, S_{i_q}^j)$ 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 $S_{i_1}^p, S_{i_2}^p, \ldots, S_{i_q}^p$ can be constructed in $O(\prod_{k=1}^{q} |S_{i_k}^p|)$ time. Therefore, all multiple B-trees can be constructed in $O(n^{2\sqrt{\log n}+2})$ time. $\square$

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 reduction by multiple queries.

The search for predecessors of elements $x_1, x_2, \ldots, x_q$ consists of $\log\log n$ rounds. During the i-th round the length of the keys 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 monotonously. For $q \le 2^{(j-1)/2}$ 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 $x^j_1, \ldots, x^j_q$ are stored in $\sqrt{\log n}/2^{(j-1)/2}$ words. Let $t = 2^{(j-1)/2}$. For simplicity, we assume the all words X^j_i contain t of keys during each round. Consider an arbitrary word $X^j_i = x^j_{(i-1)t}, x^j_{(i-1)t+1}, \ldots, x^j_{it-1}$, where $i = 1, \ldots, \sqrt{\log n}/2^{(j-1)/2}$. In the same way as in Lemma 2, words $(X^j_i)^u$ and $(X^j_i)^l$ can be computed. Using the corresponding data structure L, query $(x_1)^u \in H(S^j_{i_1})?, (x_2)^u \in H(S^j_{i_2})?, \ldots, (x_q)^u \in H(S^j_{i_q})?$ 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 duplicate the number of keys stored in one word. Since the key size has decreased by factor 4 during the two previous rounds, word X^j_i is of the form $0^{3b''}x^j_{(i-1)t}0^{3b''}x^j_{(i-1)t+1}\ldots 0^{3b''}x^j_{it-1}$, where $b'' = b'/4$ and $x^j_k \in \{0,1\}^{b''}$. We construct for each X^j_i a word $\tilde{X}^j_i$ of the form $x^j_{(i-1)t+1}x^j_{(i-1)t+2}\ldots x^j_{it-1}$. First X^j_i is multiplied with $(0^{tb'-1}1)^t$ to get $\overline{X}^j_i$. Then we perform bitwise AND of $\overline{X}^j_i$ with a word $(0^{tb'}1^{b'})^t$ and store the result in $\hat{X}^j_i$. $\hat{X}^j_i$ is of the form $x^j_{(i-1)t+1}0^{tb'+3b''}x^j_{(i-1)t+2}0^{tb'+3b''}\ldots 0^{tb'+3b''}x^j_{it}$. We can obtain $\tilde{X}^j_i$ from $\hat{X}^j_i$; details will be provided in the full version of the paper.

Finally, we duplicate the number of keys in a word by setting $X^{j+1}_i = \tilde{X}^j_{2i} \ll t_{2i}b' + \tilde{X}^j_{2i+1}$, 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.

$\square$

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.

Corollary 1 *For any $\varepsilon > 0$, there exists 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 $O(n^{\varepsilon\sqrt{\log n}})$.*

Proof: Set $p(n) = (\varepsilon/2)\sqrt{\log n} - 4$ and apply Theorem 2. $\square$

Corollary 2 *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 $O(n^{\varepsilon p(n)})$.*

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 3 *For any $\varepsilon > 0$, there exists 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 $O(n^{\varepsilon\log\log N})$.*

Corollary 4 *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 $O(n^{\varepsilon p(N)})$.*

5 Conclusion

In this paper we have presented a data structure for predecessor queries. This data structures allow us to answer predecessor queries faster than the lower bound of [BF02] at the cost of higher space requirements.

Suppose that n elements are stored in data structure A in sorted order, and query set Q also contains n elements. 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 $O(\log\log n)$ time per query.

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

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.

[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)

[FW94] M. L. Fredman, D. E. Willard, *Trans-Dichotomous Algorithms for Minimum Spanning Trees and Shortest Paths*, J. Comput. Syst. Sci. 48(3): 533-551 (1994).

[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.

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

[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.
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:
$A[j_1, j_2, \ldots, j_q] = \lt a_{q,j_q} \gt \lt 0 \gt \ldots \lt 0 \gt \lt a_{1,j_1} \gt$
$R[j_1, j_2, \ldots, j_q] = \lt 2^{r_{q,j_q}} \gt \lt 0 \gt \ldots \lt 0 \gt \lt 2^{r_{1,j_1}} \gt$
$P[j_1, j_2, \ldots, j_q] = p_{1,j_1} \ldots p_{q,j_q}$
$M[[j_1, j_2, \ldots, j_q] = \lt x_1 \gt \ldots \lt x_q \gt$, 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 $Z = \lt z_1 \gt \ldots \lt z_q \gt$, 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 ofE' 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 time $O(q \prod 2^{\lceil \log n_i \rceil + 1})$. □