Information and Computation 240 (2015) 12–30

Contents lists available at ScienceDirect

Information and Computation

www.elsevier.com/locate/yinco

Minimal indices for predecessor search ☆

Sarel Cohen *, Amos Fiat, Moshik Hershcovitch, Haim Kaplan

Tel-Aviv University, Israel

ARTICLE INFO

Article history:
Received 30 September 2013
Available online 2 October 2014

Keywords:
Successor search
Predecessor search
Succinct data structures
Cell probe model
Fusion trees
Tries
Word RAM model

ABSTRACT

We give a new predecessor data structure which improves upon the index size of the Pătraşcu–Thorup data structures, reducing the index size from $O(nw^{4/5})$ bits to $O(n \log w)$ bits, with optimal probe complexity. Alternatively, our new data structure can be viewed as matching the space complexity of the (probe-suboptimal) z-fast trie of Belazzougui et al. Thus, we get the best of both approaches with respect to both probe count and index size. The penalty we pay is an extra $O(\log w)$ inter-register operations. Our data structure can also be used to solve the weak prefix search problem, the index size of $O(n \log w)$ bits is known to be optimal for any such data structure.

The technical contributions include highly efficient single word indices, with out-degree $w/\log w$ (compared to $w^{1/5}$ of a fusion tree node). To construct these indices we device highly efficient bit selectors which, we believe, are of independent interest.

1. Introduction

A fundamental problem in data structures is the predecessor problem: given a RAM with w bit word operations, and n keys (each w bits long), give a data structure that answers predecessor queries efficiently. We distinguish between the space occupied by the n input keys themselves, which is $O(nw)$ bits, and the additional space required by the data structure which we call the *index*. The two other performance measures of the data structure which are of main interest are how many accesses to memory (called *probes*) it performs per query, and the query time or the total number of machine operations performed per query, which could be larger than the number of probes. We can further distinguish between probes to the index and probes to the input keys themselves. The motivation is that if the index is small and fits in cache, probes to the index would be cheaper. We focus on constructing a data structure for the predecessor problem that requires sublinear $o(nw)$ extra bits.

The simplest predecessor data structure is a sorted list, this requires no index, and performs $O(\log n)$ probes and $O(\log n)$ operations per binary search. This high number of probes that are widely dispersed can makes this solution inefficient for large data sets.

Fusion trees of Fredman and Willard [12] (see also [11]) reduce the number of probes and time to $O(\log_w n)$. A fusion tree node has outdegree $B = w^{1/5}$ and therefore fusion trees require only $O(nw/B) = O(nw^{4/5})$ extra bits. Variants of fusion tree nodes with larger fanout ($B = w^{1/3}$) appear in [17].

☆ Research was partially supported by the Israel Science Foundation grant no. 822-10 and the Israeli Centers of Research Excellence (I-CORE) program (Center No. 4/11).

* Corresponding author.
E-mail addresses: sarelco@gmail.com (S. Cohen), fiat@tau.ac.il (A. Fiat), moshik1@gmail.com (M. Hershcovitch), haimk@cs.tau.ac.il (H. Kaplan).

http://dx.doi.org/10.1016/j.ic.2014.09.005
0890-5401/

Table 1
Requirements of various data structures for the predecessor problem. The word length is w and the number of keys is n. Indexing groups of $w/\log w$ consecutive keys with our new word indices we can reduce the space of any of the linear space data structures above to $O(n \log w)$ bits while keeping the number of probes the same and increasing the query time by $O(\log w)$.

Data structure	Ref.	Index size (in bits)		# Non-index probes	Total # probes	# operations
Binary search		–		$O(\log n)$	$O(\log n)$	$O(\#\text{probes})$
van Emde Boas	[21]	$O(2^w)$		$O(1)$	$O(\log w)$	$O(\#\text{probes})$
x-fast trie	[22]	$O(nw^2)$		$O(1)$	$O(\log w)$	$O(\#\text{probes})$
y-fast trie	[22]	$O(nw)$		$O(1)$	$O(\log w)$	$O(\#\text{probes})$
x-fast trie on "splitters" poly(w) apart	Folklore	$O(n/\text{poly}(w))$		$O(\log w)$	$O(\log w)$	$O(\#\text{probes})$
Fusion trees	[12]	$O(nw^{4/5})$		$O(1)$	$O(\frac{\log n}{\log w})$	$O(\#\text{probes})$
Beame and Fich	[3]	$\Theta(n^{1+\epsilon} w)$		$O(1)$	$O(\frac{\log w}{\log\log w})$	$O(\#\text{probes})$
Grossi et al.	[13]	$\Theta(n \log w) + \Theta(2^{c \cdot w})$ ($c < 1$ constant)		$O(1)$	$O(\frac{\log n}{\log w})$	$O(\#\text{probes})$
z-fast trie	[4,6,5]	$O(n \log w)$	$\frac{\text{exp.}}{\text{w.c.}}$	$\frac{O(1)}{O(\log n)}$	$\frac{O(\log w)}{O(\log n)}$	$O(\#\text{probes})$
Pătraşcu and Thorup	[16]	$O(nw)$ or $O(nw^{4/5})$		$O(1)$	Optimal given linear space	$O(\#\text{probes})$
Pătraşcu and Thorup + γ-nodes	This paper	$O(n \log w)$		$O(1)$	Optimal given linear space	$O(\#\text{probes} + \log w)$

Another predecessor data structure is the y-fast trie of Willard [22]. It requires linear space ($O(nw)$ extra bits) and $O(\log w)$ probes and time per query.

Grossi et al. (Lemma 3.3 in [13]) give a predecessor data structure that is highly efficient in space and number of probes – given a large precomputed table (exponential in the word size) which can be shared amongst multiple predecessor data structures.

Pătraşcu and Thorup [16] solve the predecessor problem optimally (to within an $O(1)$ factor) for any possible point along the probe count/space tradeoff, and for any value of n and w. However, they do not distinguish between the space required to store the input and the extra space required for the index. They consider only the total space which cannot be sublinear.

Pătraşcu and Thorup's linear space data structure for predecessor search is an improvement of three previous data-structures and achieves the following bounds.

1. For values of n such that $\log n \in [0, \frac{\log^2 w}{\log\log w}]$ their data structure is a fusion tree and therefore the query time is $O(\log_w n)$. This bound increases monotonically with n.
2. For n such that $\log n \in [\frac{\log^2 w}{\log\log w}, \sqrt{w}]$ their data structure is a generalization of the data structure of Beame and Fich [3] that is suitable for linear space, and has the bound $O(\frac{\log w}{\log\log w - \log\log\log n})$. This bound increases from $O(\frac{\log w}{\log\log w})$ at the beginning of this range to $O(\log w)$ at the end of the range.
3. For values of n such that $\log n \in [\sqrt{w}, w]$ their data structure is a slight improvement of the van Emde Boas (vEB) data structure [21] and has the bound of $O(\max\{1, \log(\frac{w - \log n}{\log w})\})$. This bound decreases with n from $O(\log w)$ to $O(1)$.

The x-fast-trie [22] consists of a trie over the keys and a perfect hash table mapping the set of all prefixes of the n w-bit keys to $O(1)$ words containing the prefix and a pointer to the vertex in the trie associated with this prefix. Given a query x, and using a binary search on the length of x (in every iteration we check if the corresponding prefix of x is in the hash-table), we can find the longest common prefix (LCP) of x with any of the n keys. From the node in the trie representing this longest common prefix there is a pointer to the successor or predecessor of x, and as the keys are stored in a doubly-linked list we can go from one to the other. Since there are $O(nw)$ prefixes in the hash-table, the size of this data-structure is $O(nw)$ words.

In y-fast-tries [22] space requirements are further reduced to $O(n)$ w-bit words by choosing $\Theta(\frac{n}{w})$ keys in the x-fast-trie (approximately evenly spaced). A binary search tree is used to represent the keys between consecutive elements in the trie (there are about $O(w)$ of these). Both the x-fast trie and the y-fast tries perform predecessor/successor queries in $O(\log w)$ time.

For detailed descriptions of the x-fast trie and y-fast trie see [22,3]

A recent data structure of Belazzougui et al. [4] called the probabilistic z-fast trie, reduces the extra space requirement to $O(n \log w)$ bits, but requires a (suboptimal) expected $O(\log w)$ probes (and $O(\log n)$ probes in the worst case). See Table 1 for a detailed comparison between various data structures for the predecessor problem with respect to the space and probe parameters under consideration.

Consider the following multilevel scheme to reduce index size: (a) partition the keys into consecutive sets of $w^{1/5}$ keys, (b) build a Fusion tree index structure for each such set (one w bit word), and (c) index the smallest key in every such group using any linear space data structure. The number of fusion tree nodes that we need $n/w^{1/5}$ and the total space required for these nodes and the data structure that is indexing them is $O(nw^{4/5})$.