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 $O(nw^{4/5})$.

This standard bucketing trick shows that we can get indices of smaller size by constructing a "fusion tree node" of larger outdegree. That is we seek a data structure, which we refer to as a *word-index*, that by using $O(1)$ words can answer predecessor queries with respect to as many keys as possible.

Our main contribution is such a word index that can handle $w/\log w$ keys (rather than $w^{1/5}$ for fusion trees).[1] However, this new highly compact index requires $\Theta(\log w)$ operations per search (versus the $O(1)$ operations required by Fusion trees).

Using these word indices we obtain, as described above, a (deterministic) data structure that, for any n, w, answers a predecessor query with an optimal number of probes (within an $O(1)$ factor), and requires only $O(n \log w)$ extra bits. We remark that we only probe $O(1)$ non-index words (which is true of Pătraşcu–Thorup data structures as well, with minor modifications). The penalty we pay is an additional $O(\log w)$ in the time complexity.

Space efficient data structures have been extensively studied. *Implicit data structures* are data structures that require no extra space other than the space required for the data itself (see e.g. [10]). Succinct data structures (see e.g. [18]) do make use of extra space, but no more than $o(1)$ of the information theoretic lower bound required to represent the data.

Indices of small size are particularly motivated today by the multicore shared memory architectures abundant today [8,19]. When multiple cores access shared cache/memory, contention arises. Such contention is deviously problematic because it may cause serialization of memory accesses, making a mockery of multicore parallelism. Multiple memory banks and other hardware are attempts to deal with such problems, to various degrees. Thus, the goals of reducing the index size, so it fits better in fast caches, reducing the number of probes extraneous to the index, and the number of probes within the index, become critical.

1.1. The structure of this paper

- Section 2 describes our succinct predecessor search data-structure and give a high level description of the components required, a detailed description of which is given in the following sections.
- In Section 3 we describe the β-node, an $O(1)$ word index that allows us to search amongst $w/\log w$ keys (one word each) in expected $O(1)$ probes and expected $O(\log w)$ time. The β-node is much easier to describe than the γ-node described in the following section.
- Section 4 describes the γ-node, a deterministic $O(1)$ word index with which one can search amongst $w/\log w$ keys in worst case $O(1)$ probes and worst case $O(\log w)$ time. Searching with a γ-node requires the bit-selector of Section 5.
- Section 5 describes our efficient bit-selectors: an $O(1)$ word data structure that allows one to extract a sequence of $\Theta(w/\log w)$ bits from a word in time $O(\log w)$.
- Section 6 discusses the time required to construct α, β, and γ-nodes.
- In Section 7 we list several open questions that arise from our work.

2. High level overview of our results and their implications

Computation model: We assume a RAM model of computation with w bits per word. A key (or query) is one word (w bits long). We can operate on the registers using at least a basic instruction set consisting of (as defined in [7]): Direct and indirect addressing, conditional jump, and a number of *inter-register operations*, including addition, subtraction, bitwise Boolean operations and left and right shifts. All operations are unit cost. Moreover, our γ-node construction do not require multiplication.

We give three variants of high outdegree single word indices which we call α-nodes, β-nodes, and γ-nodes. Each of these structures index $w/\log w$ keys and answer predecessor queries using only $O(1)$ w-bit words, $O(\log w)$ time, and $O(1)$ extra-index probes (in expectation for α and β-nodes, worst case for γ-nodes) to get at most two of the $w/\log w$ keys.

The α-node is simply a z-fast trie [4] applied to $w/\log w$ keys. Given the small number of keys, the z-fast trie can be simplified. A major component of the z-fast trie involves translating a prefix match (between a query and a trie node) to the query rank. As there are only $w/\log w$ keys involved, we can discard this part of the z-fast trie and store ranks explicitly in $O(1)$ words.

The β-node is arguably simpler than the z-fast trie, and is based on a different set of ideas. It has the same performance as the α-nodes and is described in Section 3.

Our final variant, γ-nodes, has the advantages that it is deterministic and gives worst case $O(1)$ non-index probes, and, furthermore, requires no multiplication.

To get the γ-nodes we introduce highly efficient bit-selectors (see Section 2.2) that may be of independent interest. Our bit-selectors are data structures that allow one to select a (fixed) multiset of $w/\log w$ bits from a w-bit input word. The data structure requires $O(1)$ words and computes the resulting selection in time $O(\log w)$ while performing only $O(1)$ probes to memory.

[1] The $w/\log w$ keys take more than $O(1)$ words but are not considered part of the word index.

We remark that a much simpler alternative to γ-nodes would make use of Benes networks on w inputs (see, *e.g.*, [14]) to do bit extraction. However – the näive implementation of such Benes networks requires both $\Theta(\log w)$ w-bit words and $\Theta(\log w)$ probes to memory.

If multiplication is not in the instruction set and only "standard operations" are allowed (so-called standard AC(0) model), then any $O(1)$ time predecessor search requires space $\exp(w)$ where w is the number of bits per word [20]. This means that it would be impossible to derive an improved γ-node (or Fusion tree node) with $O(1)$ time predecessor search without the multiplication operator.

2.1. Succinct predecessor data structure

As mentioned in the introduction we obtain using our word indices a predecessor data structure that requires $O(n \log w)$ bits in addition to the input keys. The idea is standard and simple: We divide the keys into consecutive chunks of size $w/\log w$ keys each. We index each chunk with one of our word indices and index the chunks (that is the first key in each chunk) using another linear space data structure. This has the following consequences depending upon the linear space data structure which we use to index the chunks. (We henceforth refer to our γ-nodes, but similar results can be obtained using either α or β-nodes in expectation.)

Fusion trees $+\ \gamma$-nodes: This data structure answers predecessor queries with $O(\log_w n)$ probes, and $O(\log_w n + \log w)$ time.

The optimal structure of Pătraşcu and Thorup $+\ \gamma$-nodes: Here the number of probes to answer a query is optimal, the time is $O(\#\text{probes} + \log w)$.

y-fast-trie $+\ \gamma$-nodes: This gives an improvement upon the recently introduced [probabilistic] z-fast-trie, [4,6] (we omit the "probabilistic" prefix hereinafter). The worst-case probes and query time improves from $O(\log n)$ to $O(1)$ probes and $O(\log w)$ query time, and the data structure is deterministic.

The weak prefix search problem: In this problem the query is as follows. Given a bit-string p, such that p is the prefix of at least one key among the n input keys, return the range of ranks of those input keys having p as a prefix.

It is easy to modify the index of our predecessor data structures to a new data structure for "weak prefix search". We construct a word x containing the query p padded to the right with trailing zeros, and a word y containing the query p padded to the right with trailing ones. Searching for the rank of the successor of x in S and the rank of the predecessor of y in S gives the required range.

We note that we can carry out the search of the successor of x and the predecessor of y without accessing the keys indexed by the γ-nodes. As we will see, our γ-nodes implement a succinct blind trie. Searching a blind trie for the right rank of the predecessor/successor typically requires accessing one of the indexed keys. But, as implicitly used in [5], this access can be avoided if the query is a padded prefix of an indexed key such as x and y above. This implies that the keys indexed by the γ-nodes can in fact be discarded and not stored at all. We get a data structure of overall size $O(n \log w)$ bits for weak prefix search.

Belazzougui et al. [5] show that any data structure supporting "weak prefix search" must have size $\Omega(n \log w)$ bits. Hence, our index size is optimal for this related problem.

2.2. Introducing bit-selectors and building a (k,k)-bit selector

To construct the γ-nodes we define and construct bit selectors as follows. A (k, L) bit-selector, $1 \le k \le L \le w$, consists of a preprocessing phase and a query phase (see Fig. 1):

- The preprocessing phase: The input is a sequence of length k (with repetitions),

$$I = I[1], I[2], \ldots, I[k],$$

  where $0 \le I[j] \le w-1$ for all $j = 1, \ldots, k$. Given I, we compute the following:
  - A sequence of k strictly increasing indices, $0 \le j_1 < j_2 < \cdots < j_k \le L-1$, and,
  - An $O(1)$ word data structure, $D(I)$.
- The query phase: given an input word x, and using $D(I)$, produces an output word y such that

$$y_{j_\ell} = x_{I[\ell]}, \quad 1 \le \ell \le k,$$
$$y_m = 0, \quad m \in \{0, \ldots w-1\} - \{j_\ell\}_{\ell=1}^{k}.$$

One main technically difficult result is a construction for (k,k) bit-selectors for all $1 \le k \le w/\log w$ (Section 5). The bit selector query time is $O(\log w)$, while the probe complexity and space are constant.

Brodnik, Miltersen, and Munro [7] give an $O(\log w)$ word data structure with the functionality of a (k, L)-bit-selector, for any $1 \le k \le L \le w$. This data structure makes use of the Butterfly network encoding of permutations. The use of Benes networks to represent permutations also appears in Munro et al. [15].

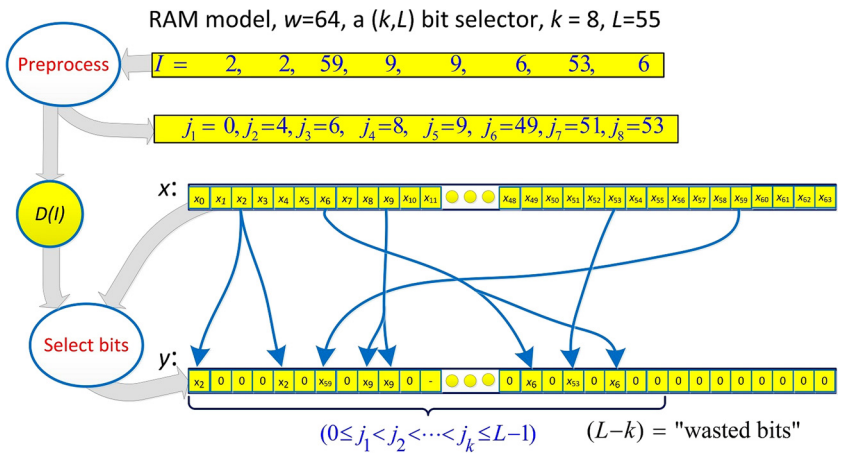

Fig. 1. A (k, L) bit-selector. Note that $I[\ell]$, $1 \le \ell \le k$, values can include repetitions and need not be in any order, however the j_ℓ sequence is an ascending subsequence of $1, 2, \ldots, L$.

	k	L	$\lvert D(I)\rvert$ in words	# Operations	Multiplication?
Fusion tree bit-selector	$1 \le k \le w^{1/5}$	k^4	$O(1)$	$O(1)$	Yes
Our bit-selector	$1 \le k \le w/\log w$	k	$O(1)$	$O(\log w)$	No

Fig. 2. The bit-selector used for Fusion Trees in [12,11] *vs.* our bit-selector.

Note that, for (k, k)-bit-selectors, it must be that $j_\ell = \ell - 1$, $1 \le \ell \le k$, independently of I. For a sequence of indices I, we define $x[I]$ to be the bits of x in these positions (ordered as in I), if I has multiplicities then $x[I]$ also has multiplicities. With this notation a (k, k) bit selector $D(I)$ computes $x[I]$ for a query x in $O(\log w)$ time.

A special case of a $(w^{1/5}, w^{4/5})$ bit-selector, where the sequence I is a strictly increasing sequence of indices, appears in [12] and lies at the core of the Fusion Tree data structure. Fig. 2 compares the fusion tree bit-selector with our construction.

We remark that Andersson, Miltersen, and Thorup [2] give an AC(0) implementation of fusion trees, *i.e.*, they use special purpose hardware to implement a (k, k) bit-selector (that produces a sketch of length k containing k bits of the key). Ignoring other difficulties, computing a [perfect] sketch in AC(0) is easy: just lead wires connecting the source bits to the target bits. With this interpretation, our bit-selector is a software implementation in $O(\log w)$ time that implements the special purpose hardware implementation of [2].

Our bit-selectors are optimal with respect to query time, when considering implementation on a "practical RAM" (no multiplication is allowed) as defined by Miltersen [14]. This follows from Brodnik et al. [7] (Theorem 17) who prove that in the "practical RAM" model, any (k, k)-bit-selector, with $k \ge \log^{10} w$, requires at least $\Omega(\log k)$ time per bit-selector query. (Observe that the bit-reversal of Theorem 17 in [7] is a special case of bit-selection.)

3. α-Nodes and β-nodes

The α-node, briefly mentioned in Section 2, is a simplification of the z-fast trie of [4], stored in a single word, and indexing $w/\log w$ keys.

The z-fast trie maps each input query into one of a linear number of strings, computed during preprocessing. The z-fast trie also makes use of monotone minimal perfect hash functions so as to compute the rank of this string, and this rank is used as an index to another data structure to compute the rank of the predecessor to the query. α-nodes do not use these hash functions as ranks can be explicitly stored within the same space bounds (up to a constant factor): the relative rank of a key amongst $w/\log w$ keys requires $O(\log w)$ bits and thus all the ranks can be explicitly stored within a single word. In summary, such α-nodes are a randomized $O(1)$ word data structure with expected $O(\log w)$ time for predecessor search amongst $w/\log w$ keys.

A possibly even simpler approach, with the same time and space bounds, is the β-node. Like the α-node, this is a randomized $O(1)$ word index. Its expected query time is $O(\log w)$ and its expected number of probes is $O(1)$.

Both the α-node and the β-node have inferior worst-case probe complexity when compared to the γ-node (described in the next section, Section 4). Neither α-nodes nor β-nodes make use of the bit-selectors (Section 5), both make use of signature comparisons.

The reason for introducing the β-node is that (arguably) it is even simpler than the α-node since it omits the complexity of the "fat binary search" of the z-fast trie. In any case, while simplicity may be a matter of taste, the techniques are different. While the α-node uses techniques similar to the x-fast trie, the β-node is based upon the (simpler?) binary search tree.

The basic idea behind the β-node is to consider the full binary trie for the $w/\log w$ keys, it has a $(1/3, 2/3)$ separator which represents some prefix p. We partition the $w/\log w$ keys into those that start with p and those that do not. Each of the parts contains at least $1/3$ and at most $2/3$ of the keys. We store the signature of p at the root of the tree, and continue recursively. As the signatures are $O(\log w)$ bits long, and there are $O(w/\log w)$ relevant prefixes, the β-node fits into $O(1)$ w-bit words.

More formally, let S_0 be a set of n w-bit binary strings stored consecutively in ascending order in the memory. We describe the β-structure, which is a randomized succinct index data-structure, which supports $\text{rank}_{S_0}(x)$ queries for any $x \in \{0,1\}^w$ (recall that $\text{rank}_S(x)$ is the rank of x in S). Its size is $O(n(\log w + \log n) \cdot \frac{1}{w})$ w-bit words ($O(n(\log w + \log n) \cdot \frac{1}{w})$ bits), the query time is $O(\log n)$ (in expectation and w.h.p.), and the number of probes it makes outside the index is $O(1)$ (in expectation and w.h.p.).

"With high probability (w.h.p.)" above means that the probability that a query will take $O(\log n)$ time and $O(1)$ probes outside the index is at least $1 - \frac{1}{n}$. By a β-node we refer to a β-structure with $n = \frac{w}{\log w}$ keys. The size of a β-node is $O(1)$, the query time is $O(\log w)$ (in expectation and w.h.p.) and the number of probes outside the index is $O(1)$ (in expectation and w.h.p.).

We start by describing a non-succinct version of a β-structure, which we refer to as B-structure, and then we describe how to transform the non-succinct B-structure into a succinct β-structure which occupies only $O(n(\log w + \log n))$ bits.

3.1. The prefix partitioning lemma

Let us start by defining a prefix-partition operator $\sqsubseteq$: Let $S \in \{0,1\}^*$ be an arbitrary set of binary strings, and let $p \in \{0,1\}^*$ be a binary string, we partition S into two subsets: $S_{\sqsubseteq p}$ and $S_{\not\sqsubseteq p}$.

$S_{\sqsubseteq p}$ is the set of all the elements of S which start with p, and $S_{\not\sqsubseteq p} = S - S_{\sqsubseteq p}$, is the set of all the elements of S which don't start with p.

The following lemma proves that there exists a prefix p which partitions S into two approximately equal subsets.

Theorem 3.1. *For every set S of binary strings, $|S| \geq 2$, there exists a binary string $p \in \{0,1\}^*$ s.t. $\frac{1}{3}|S| \leq |S_{\sqsubseteq p}| \leq \frac{2}{3}|S|$ and $\frac{1}{3}|S| \leq |S_{\not\sqsubseteq p}| \leq \frac{2}{3}|S|$.*

Proof. Let initially $p = \epsilon$ be the empty string. While ($|S_{\sqsubseteq p}| > \frac{2}{3}|S|$), if $|S_{\sqsubseteq p0}| \geq |S_{\sqsubseteq p1}|$ add a 0-bit to the end of p, otherwise add a 1-bit to the end of p. We stop the loop at the first time that $|S_{\sqsubseteq p}| \leq \frac{2}{3}|S|$, and it is easy to verify that at that point we also have that $|S_{\not\sqsubseteq p}| \leq \frac{2}{3}|S|$. □

3.2. Construction of a B-structure

Let S_0 be the initial set of n w-bit binary strings. Assume without loss of generality that $0^w \in S_0$. We can assume that, since if $0^w \notin S_0$ then $\text{rank}_{S_0}(x) = \text{rank}_{S_0 \cup \{0\}}(x) - 1$ for every $x > 0$. The B-structure is a binary tree containing a prefix of the strings in S_0 in each internal node, and at most 3 keys of S_0 at each leaf.

We define the B-structure recursively for a subset $S \subseteq S_0$ (starting with $S = S_0$). Let p be a prefix of a key in S as in Lemma 3.1, define $p_{min}, p_{max} \in S$ to be the minimum and maximum keys respectively in S which start with p. Store p at the root of the B-structure of S, the right child is a B-structure of $S_R = S_{\sqsubseteq p} \cup \text{StrictPredecessor}(S, p_{min})$ (we define here $\text{StrictPredecessor}(S, p_{min})$ to be the maximal key in S which is smaller than p_{min}, or p_{min} itself if it is the minimal key in S), the left child is a B-structure of $S_L = S_{\not\sqsubseteq p} \cup p_{max}$. We "associate" S with the root, S_R with its right child, and S_L with its left child. When $|S| \leq 3$, we stop the recursion and store the (at most 3) keys of S in the leaf of the B-structure.

According to Lemma 3.1, $|S_R|, |S_L| \leq \frac{2}{3}|S| + 1$, and since $|S_L| + |S_R| \leq |S| + 2$, we get that the height of the resulting tree is $O(\log |S_0|)$, and the number of nodes in the tree is $O(|S_0|)$.

3.3. Querying the B-structure

Let $x \in \{0,1\}^w$ be the query word. Start the search in the root. The root contains a prefix p, if x starts with p continue the search in the right child, otherwise, continue the search in the left child. Continue the search similarly in every internal node that we reach, until we reach a leaf.

In the leaf at most 3 keys are stored, denote them by $s_1 < s_2 < s_3$. If $s_1 \leq x < s_2$ output $\text{rank}_{S_0}(s_1)$ (that is, the rank of s_1 in S_0). If $s_2 \leq x < s_3$ output $\text{rank}_{S_0}(s_2)$. Otherwise, $x > s_3$ output $\text{rank}_{S_0}(s_3)$.

3.4. Correctness of the B-structure

Given $x \in \{0,1\}^w$ we need to prove that the output of the search procedure is $\text{rank}_{S_0}(x)$.

Let $y = \text{Pred}(S_0, x)$ be the predecessor of x in S_0. At the end of the search we reach a leaf which stores between 1 and 3 elements of S_0. We need to prove that y is one of these keys. Let $(v_0, v_1, \ldots, v_k)$ be the root-to-leaf path traversed during the search. Let p_i be the prefix stored at v_i and let S_i be the subset of S_0 associated with v_i, for $i = 0, 1, \ldots, k$. We prove by induction on i, for $i = 0, 1, \ldots, k$, that y is a member of S_i. This is correct at the root ($i = 0$), since $y \in S_0$ by our assumption that $0^w \in S_0$. For the inductive step, we assume $y \in S_i$, and prove that $y \in S_{i+1}$.

Lemma 3.2. *Let $y = \text{Pred}(S_0, x)$. If $y \in S_i$ then $y \in S_{i+1}$.*

Proof. Let $p_{min}, p_{max} \in S_i$ be the minimum and maximum keys in S_i respectively which start with p_i.

If v_{i+1} is a right child, then x starts with p_i and $S_{i+1} = (S_i)_{\sqsubseteq p_i} \cup \text{StrictPredecessor}(S_i, p_{min})$. If $y \in (S_i)_{\sqsubseteq p_i}$ we are done, otherwise it must be that $y = \text{StrictPredecessor}(S_i, p_{min})$ since x starts with p_i and y is its predecessor in S_i.

If v_{i+1} is a left child, then x doesn't start with p_i and $S_{i+1} = (S_i)_{\not\sqsubseteq p_i} \cup p_{max}$. If $x < p_{min}$ then y doesn't start with p_i and hence $y \in (S_i)_{\not\sqsubseteq p_i}$. If $x > p_{max}$ then either $y = p_{max}$, or $y > p_{max}$ and then $y \in (S_i)_{\not\sqsubseteq p_i}$. We get that in all the cases, $y \in (S_i)_{\not\sqsubseteq p_i} \cup p_{max}$ as required. □

3.5. Making it succinct: from B-structure to β-structure

We now describe the succinct variant of the B-structure, which we call β-structure. Its index occupies $O(n(\log w + \log n))$ bits, and its search time is $O(\log n)$ (w.h.p.), and the number of probes outside the index is $O(1)$ (w.h.p.).

Each node of the β-structure occupies $O(\log w + \log n)$ bits, defined as follows:

- **Inner nodes:** Replace every prefix p in the B-structure with a pair $<|p|, h(p)>$, $|p|$ is the length of the prefix p ($\log w$ bits) and $h(p)$ being a signature of p of length ($2 \log n$) bits, computed using a universal hash function. Given p, computing $h(p)$ requires $O(1)$ time. To test if x starts with p check if $h(x[1..|p|]) = h(p)$. If so, then x starts with p with probability at least $1 - 2^{-2\log n} = 1 - (\frac{1}{n})^2$, otherwise x doesn't start with p.
- **Leaves:** In the leaves, replace the $O(1)$ (at most 3) keys stored at each leaf with their rank in S_0 ($O(\log n)$ bits). In the search procedure, when reaching a leaf, retrieve these keys (from the static sorted list of the keys of S_0) using their ranks in $O(1)$ word-accesses.

Finally, at the end of the search assume the algorithm suggests that $\text{rank}_{S_0}(x) = i$. We can test if it's correct using $O(1)$ word-accesses by checking that $s_i \le x < s_{i+1}$ (recall the notation $S_0 = s_1 < s_2 < \ldots < s_n$). With probability at most $2^{-2\log n} \cdot \log n < \frac{1}{n}$ we will get an error at some node along the search path. When an error is detected we do a binary search to find the predecessor of x among the static set of n keys, this takes $O(\log n)$ time and probes. So the binary search takes $o(1)$ on average. When no error occurs, the search time is $O(\log n)$ (this happens with probability at least $1 - \frac{1}{n}$). Hence, the query time is $O(\log n)$ (in expectation and w.h.p.), and we accesses only $O(1)$ words outside the index (in expectation and w.h.p.).

The following theorem summarizes the main result of this section.

Theorem 3.3. *Given a set S_0 of n w-bit binary strings stored consecutively in ascending order in the memory. The β-structure, described above, supports $\text{rank}_{S_0}(x)$ queries for any $x \in \{0,1\}^w$ in $O(\log n)$ time (in expectation and w.h.p.), and it makes $O(1)$ probes outside the index (in expectation and w.h.p.). The size of the β-structure is $O(n(\log w + \log n))$ bits.*

4. γ-Nodes

In this section we use the $(w/\log w, w/\log w)$-bit-selector, described in Section 5, to build a *γ-node* and prove the following theorem.

Theorem 4.1. *A γ-node answers predecessor queries over a static set S of at most $w/\log w$ w-bit keys. The γ-node consists of $O(1)$ w-bit words (in addition to the input S) and answers predecessor queries with $O(1)$ word probes, and $O(\log w)$ operations.*

We describe the γ-node data structure in stages, beginning with a *slow γ-node* below. A slow γ-node is defined as a γ-node but performs $O(w/\log w)$ operations rather than $O(\log w)$.

4.1. Construction of slow γ-nodes

We build a blind trie over the set of keys $S = y_1 < y_2 < \cdots < y_k$, $k \le w/\log w$. We denote this trie by $T(S)$. The trie $T(S)$ is a full binary tree with k leaves, each corresponds to a key, and $k-1$ internal nodes. (We do not think of the keys as part of the trie.) We store $T(S)$ in $O(1)$ w-bit words. (The keys, of course require $|S|w$ bits.) $T(S)$ has the following structure:

1. Key y_i corresponds to the ith leaf from left to right. We store i in this leaf and denote this leaf by $\ell(y_i)$.
2. Each internal node of $T(S)$ has pointers to its left and right children.
3. An internal node u includes a bit index, i_u, in the range $0, \ldots, w-1$, i_u is the length of the longest common prefix of the keys associated with the leaves in the subtree rooted at u.
4. Keys associated with descendants of the left-child of u have bit i_u equal to zero. Analogously, keys associated with descendants of the right-child of u have bit i_u equal to one.

A recursive construction of $T(S)$ above is as follows: let p be the longest common prefix of all keys, store $|p|$ in the root, a pointer to a recursively constructed left subtree for those keys whose prefix is $p\|0$, and to a recursively constructed right subtree for those keys whose prefix is $p\|1$.

In addition to $T(S)$, we assume that the keys in S are stored in memory, consecutively in sorted order.

Indices both in internal nodes and leaves are in the range $0, \ldots, w-1$ and thereby require $O(\log w)$ bits. Since $T(S)$ has $O(w/\log w)$ nodes, a pointer to a node also requires $O(\log w)$ bits. Thus, in total, each node in $T(S)$ requires only $O(\log w)$ bits. It follows that $T(S)$ (internal nodes and leaves) requires only $O(w)$ bits (or, equivalently, can be packed into $O(1)$ words).

Fundamentally, a *blind-search* follows a root to leaf path in the blind trie $T(S)$, ignoring intermediate bits. Searching $T(S)$ for a query x always ends at a leaf of the trie (which contains the index of some key). Let $\mathrm{bs}(x, S)$ denote the index stored at this leaf, and let $\mathrm{bkey}(x)$ be $y_{\mathrm{bs}(x,S)}$. *I.e.*, blind search for query x in $T(S)$ leads to a leaf that points to $\mathrm{bkey}(x)$. In general, $\mathrm{bkey}(x)$ is *not* the answer to the predecessor query, but it does have the longest common prefix of x amongst all keys in S. (See [9]. See also [1] for the first use of blind trie search for predecessor search.)

To arrive at the predecessor of x, we retrieve $\mathrm{bkey}(x)$ and compute its longest common prefix with x. Let b be the next bit of x, after $\mathrm{LCP}(x, \mathrm{bkey}(x))$. We use b to pad the remaining bits, let $\|$ denote concatenation, and let

$$z = \mathrm{LCP}\big(x, \mathrm{bkey}(x)\big) \big\| b^{w-|\mathrm{LCP}(x,\mathrm{bkey}(x))|}.$$

Finally, we perform a second blind-search on z. The result of this second search gives us the index of the predecessor or the successor of x.

Overall, the number of probes required for such a search is $O(1)$. However, the computation time is equal to the length of the longest root to leaf path in $T(S)$, which is $O(w/\log w)$.

4.2. Improving the running time

Using our $(w/\log w, w/\log w)$-bit-selector we can reduce the search time in the blind trie from $O(w/\log w)$ to $O(\log w)$ operations while still representing the trie in $O(1)$ words. For that we change the first part of the query, that is the *blind-search* for $\mathrm{bs}(x, S)$ (the index of $\mathrm{bkey}(x)$). Rather than walking top down along a path in the trie we use a binary search as follows.

We need the following notation. Let $|\tau|$ denote the number of vertices along some path τ in $T(S)$. Any node $u \in T(S)$, (internal node or leaf), defines a unique root to u path in $T(S)$. Denote this path by $\pi_u = v_1, v_2, \ldots, v_{|\pi_u|}$ where v_1 is the root, $v_{|\pi_u|} = u$, and v_i is the parent of v_{i+1} in $T(S)$.

For any node $u \in T(S)$ let I_u be the sequence of indices i_v for all internal nodes v along π_u. Also, let ζ_u be a sequence of zeros and ones, one entry per edge in π_u, zero for an edge pointing left, one otherwise. For all $1 \le q \le |S|$ we define $\pi_q = \pi_{\ell(y_q)}$, $I_q = I_{\ell(y_q)}$, and $\zeta_q = \zeta_{\ell(y_q)}$. The following lemma is straightforward.

Lemma 4.2. *For any index $1 \le q \le |S|$, query x, we have that*

$$\begin{aligned}
&x[I_q] \text{ is lexicographically greater than } \zeta_q \quad \Rightarrow \quad \mathrm{bkey}(x) > y_q,\\
&x[I_q] = \zeta_q \quad \Rightarrow \quad \mathrm{bkey}(x) = y_q,\\
&x[I_q] \text{ is lexicographically smaller than } \zeta_q \quad \Rightarrow \quad \mathrm{bkey}(x) < y_q.
\end{aligned}$$

Based on Lemma 4.2, given query x, we can do binary search to find $\mathrm{bs}(S, x)$:

$L \leftarrow 1$, $R \leftarrow |S|$, $q \leftarrow \lfloor (L+R)/2 \rfloor$
while $x[I_q] \neq \zeta_q$ **do**
  if $x[I_q] > \zeta_q$ **then** $L \leftarrow q+1$
  else $R \leftarrow q-1$
  end if
  $q \leftarrow \lfloor (L+R)/2 \rfloor$
end while
return q

Lemma 4.3. *The above binary search algorithm returns* $\mathrm{bs}(x, S)$ *and has* $O(\log w)$ *iterations.*

Next we show how to implement each iteration of this binary search and compare $x[I_q]$ and ζ_q in $O(1)$ time while keeping the trie stored in $O(1)$ words.

For this we devise a sequence I of bit indices, of length $w/\log w$. Prior to running the binary search we use the bit-selector of Section 5 to compute $x[I]$ and later we use $x[I]$ to construct $x[I_q]$ in every iteration in $O(1)$ time. We extract $x[I_q]$ from $x[I]$ and retrieve ζ_q using $O(1)$ additional words. The details are as follows.

The $O(1)$ words which form the γ-node: Associated with every $1 \le q \le |S|$ is an interval $[L_q, R_q]$ such that $q = \lfloor (L_q + R_q)/2 \rfloor$. During the course of any binary search, whenever q is chosen as a splitting point, $L = L_q$ and $R = R_q$. Let $\pi_q = u_1, u_2, \ldots, u_t = \ell(y_q)$, $t = |\pi_q|$, be the path from the root u_1 to the leaf $\ell(y_q)$ (as defined above).

Define $j_L(q)$ to be the length of the longest common prefix of π_q and π_{L_q}. Define $j_R(q)$ to be the length of the longest common prefix of π_q and π_{R_q}. *I.e.*, $u_{j_L(q)}$ is the lowest common ancestor of the leaves $\ell(y_q)$ and $\ell(y_{L_q})$, and $u_{j_R(q)}$ is the lowest common ancestor of the leaves $\ell(y_q)$ and $\ell(y_{R_q})$.

Let $j(q) = \max\{j_R(q), j_L(q)\}$, and let $\tilde{\pi}_q$ be the suffix of π_q starting at node $u_{j(q)+1}$, and let $\tilde{I}_q$ be the suffix of I_q starting at $I_q[j(q)+1]$. (These are the indices stored in $u_{j(q)+1}, u_{j(q)+2}, \ldots, u_{t-1}$.) Similarly, let $\tilde{\zeta}_q$ be the suffix of ζ_q, starting at index $j(q)$.

Given S, for every $1 \le q \le |S|$ we precompute and store the following data: $j_L(q)$, $j_R(q)$, $\tilde{I}_q$, $\tilde{\zeta}_q$. It is easy to verify that $O(1)$ words suffice to store the $4|S|$ values above. Indeed, $j_L(q)$ and $j_R(q)$ are indices in $1, \ldots, |S|$, $O(\log w)$ bits each. As the number of keys $|S| \le w/\log w$, all the $j_L(q)$'s, and $j_R(q)$'s fit in $O(1)$ words. Since $\tilde{\pi}_q$ paths are pairwise disjoint, the sum of their path lengths is $O(|S|) = O(w/\log w)$. Hence, storing all the sequences $\tilde{\zeta}_q$, $1 \le q \le w/\log w$, requires no more than $O(w/\log w)$ bits. We store the $\tilde{\zeta}_q$'s concatenated in increasing order of q in a single word Z.

The sequence I for which we construct the bit selector is the concatenation of the $\tilde{I}_q$ sequences, in order of q. It follows that I is a sequence of $O(w/\log w)$ $\log w$-bit indices. The bit selector $D(I)$ is also stored as part of the γ-node.

For each q we also compute and store the index s_q of the starting position of $\tilde{\zeta}_q$ in Z. This is the same as the index of the starting position of I_q in I. Clearly all these indices s_q can be stored in a single word.

Implementing the blind search: As we mentioned, given x as a query to the γ-node, we compute $x[I]$ (once) from x and $D(I)$, which requires $O(\log w)$ operations and no more than $O(1)$ probes.

At the start of an iteration of the binary search, we have a new value of q, and access to the following values, all of whom are in $O(1)$ registers from previous iterations:

$$x[I], \quad j_L(q) \quad j_R(q) \quad L_q, \quad R_q, \quad \zeta_{L_q}, \quad \zeta_{R_q}, \quad x[I_{L_q}], \quad x[I_{R_q}].$$

We now compute ζ_q and $x[I_q]$. We retrieve $j_L(q)$ and $j_R(q)$ from the data-structure, and we also retrieve $x[\tilde{I}_q]$ from $x[I]$ and $\tilde{\zeta}_q$ from Z (note that $x[\tilde{I}_q]$ is stored consecutively in $x[I]$ and $\tilde{\zeta}_q$ is stored consecutively in Z, and we use s_q to know where they start).

If $j_L(q) \ge j_R(q)$, we compute $x[I_q] \leftarrow (x[I_L][1, \ldots, j_L(q)]) \| (x[\tilde{I}_q])$ and $\zeta_q \leftarrow (\zeta_L[1, \ldots, j_L(q)]) \| (\tilde{\zeta}_q)$.

Analogously, if $j_L(q) < j_R(q)$, and we compute

$$x[I_q] \leftarrow \big(x[I_R]\big[1, \ldots, j_R(q)\big]\big) \big\| \big(x[\tilde{I}_q]\big)$$
$$\zeta_q \leftarrow \big(\zeta_R\big[1, \ldots, j_R(q)\big]\big) \| (\tilde{\zeta}_q).$$

All these operations are easily computed using $O(1)$ SHIFT, AND, OR operations.

5. Bit selectors

5.1. An overview with a running example

In this section we describe both the preprocessing and selection operations for our bit-selectors. We sketch the selection process, which makes use of $D(I)$, the output of the preprocessing. A more extensive description and figures can be found in Section 5.2.

As described in Section 2.2 and Fig. 1, $D(I)$ is the output of the preprocessing stage, it consists of $O(1)$ words and includes precomputed constants used during the selection process. Moreover, the total working memory required throughout the selection process is $O(1)$ words.

Partition the sequence $\sigma = 0, 1, \ldots, w-1$ into $w/\log w$ blocks (consecutive, disjoint, subsequences of σ), each of length $\log w$. Let B_j denote the jth block of a word, *i.e.*, $B_j = j \log w, j \log w + 1, \ldots, (j+1)\log w - 1$, $0 \le j \le w/\log w - 1$.

Given an input word x and the precomputed $D(I)$, the selection process goes through the seven phases sketched below.

As an aid to following the description, we follow the various stages with a working example. Our example has word length $w = 16$ bits, thus – a bit index requires $\log w = 4$ bits, I consists of $w/\log w = 4$ indices (with repetitions). A "block" consists of $\log w = 4$ bits, and there are $w/\log w = 4$ blocks.

Our running example has $x = \mathbf{1}000\ 1101\ 1110\ \mathbf{0}01\mathbf{1}$, $I = 0, 15, 12, 15$. Ergo, the output of the selection process should be $x[I] = 1101$.

Phase 0: Zero irrelevant bits. We take the mask M with ones at positions in I, and set $x = x$ AND M. For our example this gives

Input : $M = \mathbf{1}000\ 0000\ 0000\ \mathbf{1}00\mathbf{1}$, $x = \mathbf{1}000\ 1101\ 1110\ \mathbf{0}01\mathbf{1}$;
Phase 0: $M = \mathbf{1}000\ 0000\ 0000\ \mathbf{1}00\mathbf{1}$, $x = \mathbf{1}000\ 0000\ 0000\ \mathbf{0}00\mathbf{1}$.

Phase 1: Packing blocks to the left: All bits of x whose index belongs to some block are shifted to the left within the block. We modify the mask M accordingly. Let the number of such bits in block j be b_j. This phase transforms M and x as follows:

Phase 0: $M = \mathbf{1}000\ 0000\ 0000\ \mathbf{1}00\mathbf{1}$, $x = \mathbf{1}000\ 0000\ 0000\ \mathbf{0}00\mathbf{1}$;
Phase 1: $M = \mathbf{1}000\ 0000\ 0000\ \mathbf{11}00$, $x = \mathbf{1}000\ 0000\ 0000\ \mathbf{01}00$.

Note that $b_0 = 1$, $b_1 = b_2 = 0$, and $b_3 = 2$. The two bits to be extracted from block 3 are shifted to be the leftmost two bits in block 3. Phase 1 requires $O(\log w)$ operations on a constant number of words (or registers). See Section 5.2.2 and Figs. 3, 4 and 5 for full details.

Phase 2: Sorting the blocks in descending order of b_j (defined in Phase 1 above). This phase transforms M and x as follows:

Phase 1: $M = \mathbf{1}000\ 0000\ 0000\ \mathbf{11}00$, $x = \mathbf{1}000\ 0000\ 0000\ \mathbf{01}00$;
Phase 2: $M = \mathbf{11}00\ \mathbf{1}000\ 0000\ 0000$, $x = \mathbf{01}00\ \mathbf{1}000\ 0000\ 0000$.

Technically, Phase 2 uses a Benes network to sort the blocks in descending order of b_j, in our running example this means block 3 should come first, then block 0, then blocks 2 and 3 in arbitrary order. Brodnik, Miltersen, and Munro [7] show how to simulate a Benes network on bits of a word, we extend this so as to sort entire blocks of $\log w$ bits.

The precomputed $D(I)$ includes $O(1)$ words to encode this Benes network. Phase 2 requires $O(\log w)$ bit operations on $O(1)$ words. See Section 5.2.3 and Fig. 6 for full details.

Phase 3: Dispersing bits: reorganize the word produced in Phase 2 so that each of the different bits whose index is in I will occupy the leftmost bit of a unique block. As there may be less distinct indices in I than blocks, some of the blocks may be empty, and these will be the rightmost blocks. This process requires $O(\log w)$ word operations to reposition the bits. This phase transforms M and x as follows:

Phase 2: $M = \mathbf{11}00\ \mathbf{1}000\ 0000\ 0000$, $x = \mathbf{01}00\ \mathbf{1}000\ 0000\ 0000$;
Phase 3: $M = \mathbf{1}000\ \mathbf{1}000\ \mathbf{1}000\ 0000$, $x = \mathbf{0}000\ \mathbf{1}000\ \mathbf{1}000\ 0000$.

See Section 5.2.4 and Fig. 7 for full details.

Phase 4: Packing bits. The goal now is to move the bits positioned by Phase 3 at the leftmost bits of the leftmost r blocks (r being the number of indices in I without repetitions) to be the leftmost r bits. Again, by appropriate bit manipulation, this can be done with $O(\log w)$ word operations. This phase transforms M and x as follows:

Phase 3: $M = \mathbf{1}000\ \mathbf{1}000\ \mathbf{1}000\ 0000$, $x = \mathbf{0}000\ \mathbf{1}000\ \mathbf{1}000\ 0000$;
Phase 4: $M = \mathbf{111}0\ 0000\ 0000\ 0000$, $x = \mathbf{011}0\ 0000\ 0000\ 0000$.

We remark that if $r = k$, *i.e.*, if I contains no duplicate indices, then we can skip Phases 5 and 6 whose purpose is to duplicate those bits required several times in I. See Section 5.2.5 and Fig. 8 for full details.

Phase 5: Spacing the bits. Once again, we simulate a Benes network on the k leftmost bits. The purpose of this permutation is to space out and rearrange the bits so that bits which appear multiple times in I are placed so that multiple copies can be made.

In our running example, Phase 5 changes neither M nor x, but this is coincidental – for other inputs ($I' \neq I$) Phase 5 would not be the identity function. Phase 5 is yet another application of a Benes network and requires $O(\log w)$ word operations. See Section 5.2.6 and Fig. 9 for full details.

Phase 6: Duplicating bits – we duplicate the bits for which space was prepared during Phase 5. This phase transforms M and x as follows:

Phase 5: $M = \mathbf{111}0\ 0000\ 0000\ 0000$, $x = \mathbf{011}0\ 0000\ 0000\ 0000$;
Phase 6: $M = \mathbf{1111}\ 0000\ 0000\ 0000$, $x = \mathbf{0111}\ 0000\ 0000\ 0000$.

Technically, Phase 6 makes use of shift and OR operations, where the shifts are decreasing powers of two. See Section 5.2.7 and Fig. 10 for full details.

Phase 7: Final positioning: The bits are all now in the k leftmost positions of a word, every bit appears the same number of times it's index appears in I, and we need to run one last Benes network simulation so as to permute these k bits. This permutation gives the final outcome. This phase transforms M and x as follows:

Phase 6: $M = \mathbf{1111}\ 0000\ 0000\ 0000$, $x = \mathbf{0111}\ 0000\ 0000\ 0000$;
Phase 7: $M = \mathbf{1111}\ 0000\ 0000\ 0000$, $x = \mathbf{1101}\ 0000\ 0000\ 0000$.

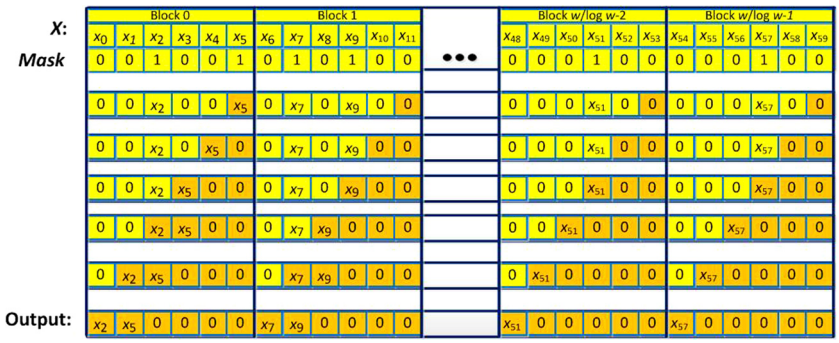

Fig. 3. An illustration of Phase 1.

Note the leftmost $|I| = w/\log w = 4$ bits of x contain the required output of the bit selector. See Section 5.2.8 and Fig. 11 for full details.

5.2. Formal description of the bit-selectors

We define the following notation. For sequences σ and τ, $\sigma \cap \tau$ denotes a subsequence of σ consisting of those values that appear somewhere in τ. For a sequence of indices σ, we define $x[\sigma]$ to be the bits of x in these positions (ordered as in σ), if σ has multiplicities then $x[\sigma]$ also has multiplicities.

An assignment to $x[\sigma]$, such as $x[\sigma] \leftarrow b_1, b_2, \ldots, b_{|\sigma|}$, $b_i \in \{0,1\}$, is shorthand notation for $x[\sigma[1]] \leftarrow b_1, x[\sigma[2]] \leftarrow b_2$, $\ldots, x[\sigma[|\sigma|]] \leftarrow b_{|\sigma|}$. (Assignment to $x[\sigma]$ makes sense if σ has no multiplicities.)

Also, given a word z, let $z \gg i$ denote a right shift of z by i bits, and $z \ll i$ a left shift by i bits.

Given an input word x and the precomputed $D(I)$, the selection process goes through the seven phases described below.

5.2.1. The ever changing x and I

As we process the various phases and sub phases of the bit selection, the original bits of x are permuted, duplicated, or set to zero.

Let x_0 be the original word and $I_0 = I$ be the original sequence of indices. Moreover, let x_t be the word x after phases 1 to t, and let I_t be a sequence of indices such that for all $j = 0, \ldots, k-1$, $x_t[I_t[j]] = x_0[I_0[j]]$. That is, if we order the bits of x_t as specified by I_t we get the same sequence of bits as if we order the bits of the input by I_0. Our goal is to have $I_7 = 1, \ldots, w/\log w$, so that x_7 is the correct output.

For any t, $0 < t \neq 6$, I_t is obtained by changing I_{t-1} so as to reflect the bit permutation performed on x_{t-1} to get x_t. These permutations need not be actually done, but are implicitly used by the bit selection algorithm.

During Phase 6, where bits are duplicated so that the number of copies of each bit is equal to the multiplicity of the index of the bit in I_0 (or I_5), I_6 is produced from I_5 by removing multiplicities and substituting $i + j - 1$ for the jth appearance of index i in I_5

It follows that for all $0 \le t \le 5$, $0 \le j \le k-1$, the multiplicity of $I_t[j]$ is equal to the multiplicity of $I_0[j]$. For $t = 6, 7$, $0 \le j \le k-1$, the multiplicity of $I_t[j]$ is one.

Initially, all bits not appearing in I_0 are set to zero simply by setting $x_0 \leftarrow x$ AND M where M is a mask with it's ith bit equal to one iff i appears in I.

The final output of the bit selection is a word containing $x_0[I] \| 0^{w-|I|}$.

For brevity, we use x as a continuously changing variable throughout the description of the different phases. The sequences I_t are needed during the preprocessing phase, the query phase requires only a constant number of precomputed words. We describe how the preprocessing phase keeps track of the various I_t sequences and the permutations applied to x implicitly through the description of the phases.

5.2.2. Phase 1: packing blocks to the left (Fig. 3)

We now describe the procedure for rearranging the bits of x so that for all blocks B, the bits $x[B \cap I]$ are assigned to the leftmost positions of $x[B]$, preserving their order. This will be done for all blocks in parallel by the inherent parallelism of word operations.

For block B, let $\mathrm{suff}_i(B)$ be the length i suffix of B.

Phase 1 requires $\log w$ subphases. We maintain the following invariant after subphase i, $1 \le i \le \log w$: The bits $x[\mathrm{suff}_i(B) \cap I]$ are assigned to the leftmost positions of $x[\mathrm{suff}_i(B)]$ and the other bits of $x[\mathrm{suff}_i(B)]$ are set to zero. Bits of x whose indices are not in $\mathrm{suff}_i(B)$ do not change. Note that this invariant initially holds for $i = 1$.

At subphase i, for $i = 2, \ldots, \log w$, for each block B whose ith largest index is not in I_0 we assign $x[\mathrm{suff}_{i-1}(B)] \| '0'$ to $x[\mathrm{suff}_i(B)]$.

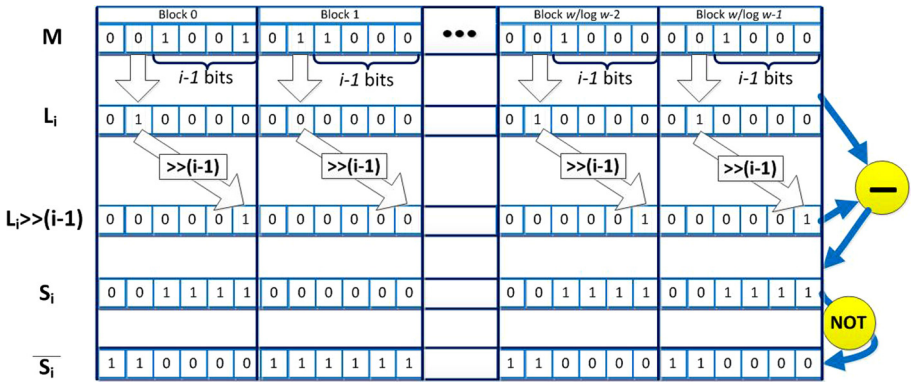

Fig. 4. The masks used during Phase 1.

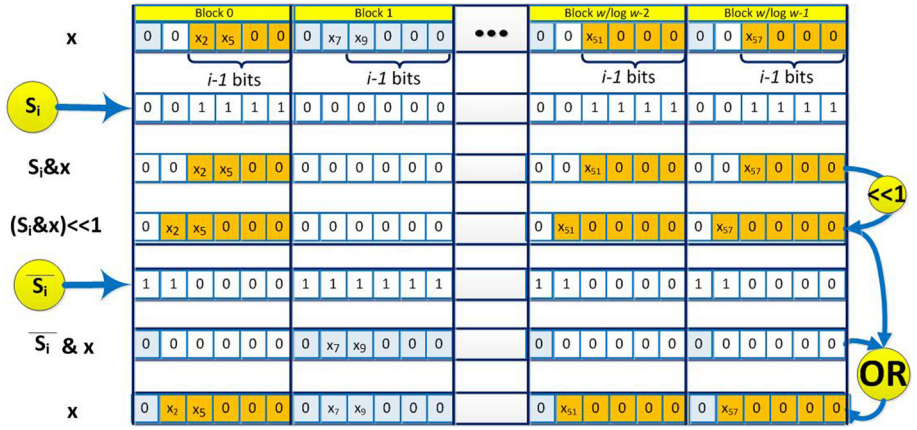

Fig. 5. A subphase of Phase 1.

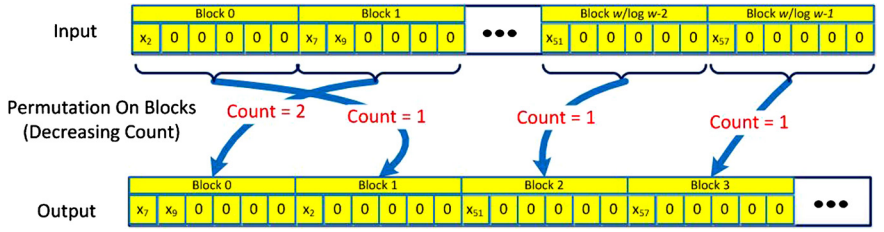

Fig. 6. An illustration of Phase 2.

Let Z_i be a word with 1 at the ith largest index of each block, and zeros elsewhere. We need Z_i during subphase i of Phase 1. Z_1 can be constructed on the fly in a register, in time $O(\log w)$, Z_i is simply a left shift of Z_1 by $i-1$. Let $L_i = \overline{M}$ AND Z_i, and let $S_i = L_i - (L_i \gg (i-1))$. See Fig. 4.

The ith subphase is as follows: We compute $y_1 = x$ AND S_i which gives the bits that have to be left shifted by one position, and we compute $y_2 = x$ AND $\bar{S}_i$ which gives a word containing the bits which are to remain in their positions. Finally, we set $x = (y_1 \ll 1)$ OR y_2. See Fig. 5.

5.2.3. *Phase 2: sorting blocks by size (Fig. 6)*

We permute $x[B_0], \ldots, x[B_{\frac{w}{\log w}-1}]$ such that they are in non-increasing order of $|B_j \cap I_1|$. Note that we know this permutation when preprocessing I_1. We implement this step using a simulation of a Benes network (described in Section 5.3). This simulation requires $O(\log w)$ operations, and uses $O(1)$ precomputed constants stored in $D(I)$, and $O(1)$ registers.

5.2.4. *Phase 3: dispersing bits (Fig. 7)*

Recall that r is the number of distinct values in I. Let $\sigma = B_0[0], B_1[0], \ldots, B_{r-1}[0]$, *i.e.*, σ is a sequence of the first (and smallest) index in every block. For any sequence π of indices, define $\tau(\pi)$ be a maximal subsequence of π without duplicate indices.

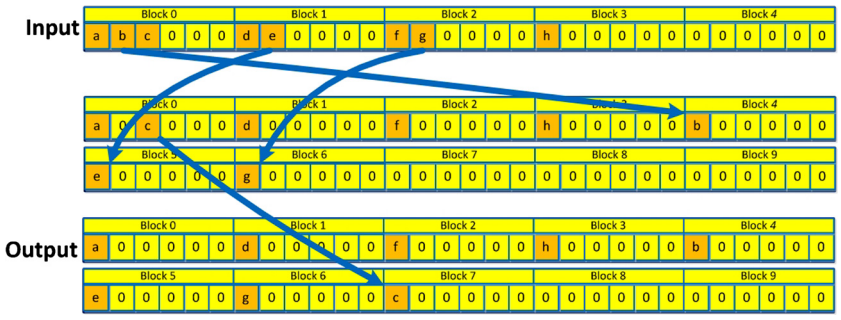

Fig. 7. An illustration of Phase 3.

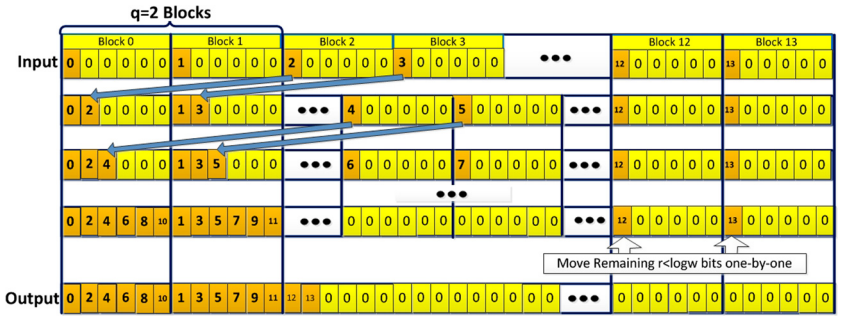

Fig. 8. An illustration of Phase 4.

In Phase 3 we disperse the bits of $x[I_2]$, so that

$$x[\sigma] = x[\tilde{\tau}],$$

for some sequence $\tilde{\tau}$ produced by some permutation on the order of $\tau(I)$. The description of $\tilde{\tau}$, is implicit in the description of Phase 3 below.

Following Phase 2, we have that $|B_0 \cap I_2| \geq |B_1 \cap I_2| \geq \ldots \geq |B_{w/\log w - 1} \cap I_2|$. Therefore, for $1 \leq i \leq \log w$, we can define

$$a_i = \left|\left\{ j \;\middle|\; |B_j \cap I_2| \geq i \right\}\right|.$$

Let $A_i = \sum_{\ell=1}^{i} a_\ell$ for $i = 1, \ldots, \log w$, and define $A_0 = 0$.

We can now define the sequence σ_i, $1 \leq i \leq \log w$,

$$\sigma_i = \sigma[A_{i-1}], \sigma\left[(A_{i-1}+1)\right], \ldots, \sigma\left[(A_i - 1)\right],$$

and the sequence

$$\xi_i = B_0[i-1], B_1[i-1], \ldots, B_{a_i - 1}[i-1], \quad 1 \leq i \leq \log w.$$

Phase 3 has $\log w$ subphases. Subphase i of Phase 3 performs the assignment $x[\sigma_i] \leftarrow x[\xi_i]$, this assignment can be implemented using $O(1)$ operations.

Isolate the bits to be moved (indices ξ_i), shift them to their new locations (indices σ_i, note that due to Phase 2, $\sigma_i[j] - \xi_i[j] = \sigma_i[j'] - \xi_i[j']$ for all $1 \leq j, j' \leq |\sigma_i|$), producing word y. Next, update x by setting $x[\xi_i]$ to zero and taking the OR with y.

Given the values a_i, the relevant masks used in this phase can be generated on the fly in constant time per mask, given the constant Z_1 from Phase 1.

5.2.5. Phase 4: packing bits (Fig. 8)

Let σ and $\tau(\pi)$ be as defined at the start of Phase 3. In Phase 4 we "push" the bits $x[\tau(I_3)]$ to the left, *i.e.*,

$$x[0, 1, \ldots, r-1] \leftarrow x[\tilde{\tau}],$$

where $\tilde{\tau}$ is a sequence produced by some permutation on the order of $\tau(I_3)$. As in Phase 3, the description of $\tilde{\tau}$, is implicit in the description of Phase 4 below. Note that $\tau(I_3)$ is a permutation of σ.

There are $\log w$ subphases in Phase 4.

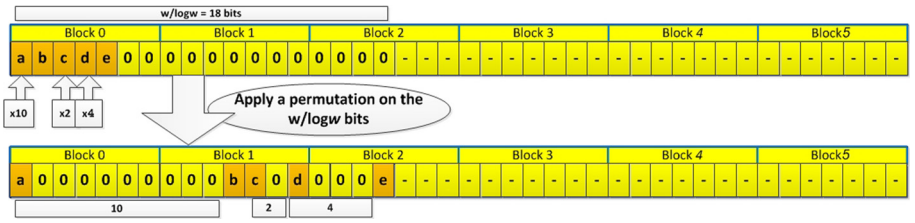

Fig. 9. An illustration of Phase 5.

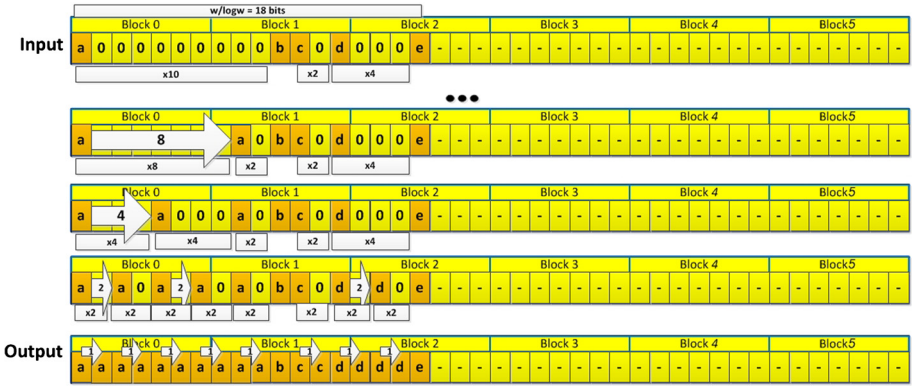

Fig. 10. An illustration of Phase 6.

Let $q = \lfloor r/\log w \rfloor$. In subphases $1, \ldots, \log w - 1$ we fill $x[B_0], x[B_1], \ldots, x[B_{q-1}]$, with some permutation of the first $q \log w$ bits of $x[\sigma]$. The last subphase is used to copy the $k < \log w$ leftover bits of $x[\sigma]$ into $x[B_q[0], B_q[1], \ldots, B_q[k-1]]$.

For $1 \le i < \log w$ define the sequences $\upsilon_i = i, \log w + i, \ldots, (q-1)\log w + i$ and $\zeta_i = iq \log w, (iq+1)\log w, \ldots, ((i+1)q - 1)\log w$.

In Subphase i of Phase 4 we perform the assignment

$$x[\upsilon_i] \leftarrow x[\zeta_i].$$

To do this using word operations, we first isolate the bits of $x[\zeta_i]$, shift them so as to be in their target locations, υ_i, and OR them into place.

The last subphase copies the remaining bits one by one, for a total of $O(\log w)$ operations.

5.2.6. *Phase 5: spacing the bits (Fig. 9)*

At the end of Phase 4 $x[0, 1, \ldots, r-1]$ is a permutation of the bits of $x[\tau[I_4]]$, and $x[j]$, $j \ge r$, are zero. Our goal is now to space the bits so as to make space for duplication of those bits whose indices appear multiple times in I_4.

In this phase we space the bits $x[0, 1, \ldots, r-1]$ by "inserting" $j-1$ zeros between $x[\ell]$ and $x[\ell+1]$ iff ℓ appears j times in I_4. We do this by permuting the bits of $x[0, 1, \ldots, k]$. There is at least one permutation that achieves this goal. This is done by simulating one such permutation by a Benes sorting network, in time $O(\log w)$, and using only $O(1)$ precomputed constants and $O(1)$ registers.

5.2.7. *Phase 6: duplicating bits (Fig. 10)*

For an index j let $m_j(I_5)$ be the number of occurrences of j in I_5. At the end of Phase 5, for every $\ell \in I_5$ such that $m_\ell(I_5) > 1$ we have that $x[\ell+1, \ell+2, \ldots, \ell+m_\ell(I_5)-1]$ contain zeros and none of the indices $\ell+1, \ell+2, \ldots, \ell+m_\ell(I_5)-1$ appear in I_5.

Phase 6 consists of $\log w$ subphases, $i = 1, \ldots, \log w$. Subphase i copies the bits of x specified by a $w/\log w$ bit mask M_i to the position offset from its current position by $\Delta_i = 2^{\log w - i}$. All these masks are precomputed at preprocessing and store in a single word with $D(I)$.

The masks M_i are defined as follows. Let $0 \le i_1 < i_2 < \ldots < i_\ell < w/\log w$ be the indices such that $m_{i_j}(I_5) > 1$. The mask M_i is 1 in positions $\{i_j + 2t\Delta_i \mid 0 \le t, (2t+1)\Delta_i < m_{i_j}(I_5)\}$ for every $1 \le j \le \ell$ such that $m_{i_j}(I_5) > \Delta_i$.

At query time in subphase i we set $x = (x \text{ OR } ((x \text{ AND } M_i) \gg \Delta_i))$ and implicitly update $M = (M \text{ OR } ((M \text{ AND } M_i) \gg \Delta_i))$. Intuitively, after subphase i the difference between the indices of consecutive 1s in M is at most Δ_i, and the corresponding bits in x were duplicated.

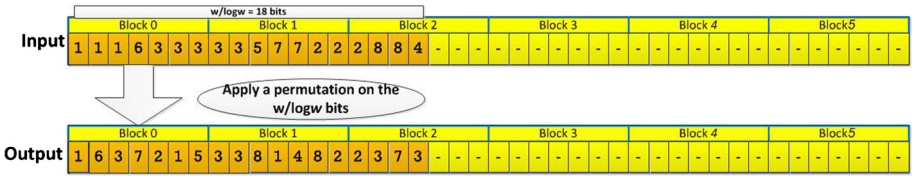

Fig. 11. An illustration of Phase 7.

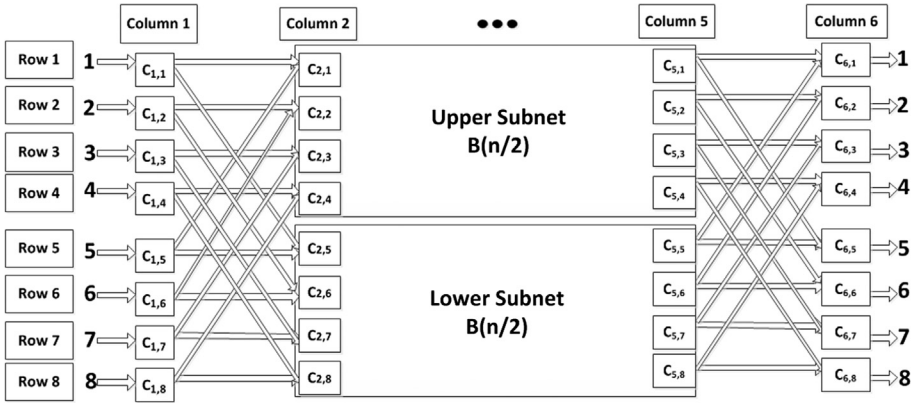

Fig. 12. One of many variants for the Benes network.

5.2.8. Phase 7: final positioning (Fig. 11)

At this stage we need permute the bits $x[1,\ldots,k]$, so as to get the final output. Note that I_6 is a permutation and it's inverse permutation, I_6^{-1} is the permutation we need to apply to x. This too requires simulation of a Benes network, see Section 5.3.

5.3. Permuting elements in a word by simulating a Benes network

We show how to prepare a set C of $O(1)$ words such that given C a Benes network implementing a given permutation σ can be applied to a word x in $O(\log w)$ operations (SHIFT, AND, OR).

We use such networks in two contexts:

- To permute the $b \leq w/\log w$ leftmost bits of x. We need this in Phases 5 and 7 of bit selection.
- To permute $w/\log w$ blocks of bits (each block of length $\log w$). We need this during Phase 2 of bit selection.

5.3.1. Overview of the Benes network

Assume that n is a power of 2. A Benes network, $B(n)$, of size n consists of two Benes networks of size $n/2$, $B_u(n/2)$, and $B_d(n/2)$. For $1 \leq i \leq n/2$, inputs i and $i+n/2$ of $B(n)$ can be routed to the ith input of $B_u(n/2)$ or to the ith input of $B_d(n/2)$. The outputs are connected similarly. For every $1 \leq i \leq n/2$ we define inputs i and $i+n/2$ as *mates*, analogously we define outputs i and $i+n/2$ as *mates*. Note that mates cannot both be routed to the same subnetwork. See Fig. 12.

The looping algorithm: A Benes network can realize any permutation σ of its inputs as follows [15]. We start with an arbitrary input, say 1, and route it to $B_u(n/2)$. This implies that the output $\sigma(1)$ is also routed to $B_u(n/2)$. The mate o of $\sigma(1)$ must then be routed to $B_d(n/2)$. This implies that $\sigma^{-1}(o)$ is routed to $B_d(n/2)$. If the mate of $\sigma^{-1}(o)$ is 1 we "completed a cycle" and we start again with an arbitrary input which we haven't routed yet. Otherwise, if the mate of $\sigma^{-1}(o)$ is not 1 then we route this mate to $B_u(n/2)$ and repeat the process.

Levels of the Benes network: If we lay out the Benes network then the 1st level of the recursion above gives us 2 "stages" consisting of n 2×2 switches, stage #1 connecting the inputs to $B_u(n/2)$ and $B_d(n/2)$, and stage # $2\log n-1$ connecting $B_u(n/2)$ and $B_d(n/2)$ to the outputs. Opening the recursion gives us $2\log n-1$ stages, each consisting of n 2×2 switches.

To implement any specific permutation, one needs to set each of these switches.

5.3.2. Permuting the $b = w/\log w$ leftmost bits of the word

We now describe an $O(1)$ word representation for any permutation σ on $b = w/\log w$ elements that allows us to apply σ to the b leftmost bits of a query word x while doing only $O(\log b)$ operations. We obtain this data structure by encoding the Benes network for σ in $O(1)$ words. To answer a query we use this encoding to apply each of the $2\log b - 1 = O(\log b)$

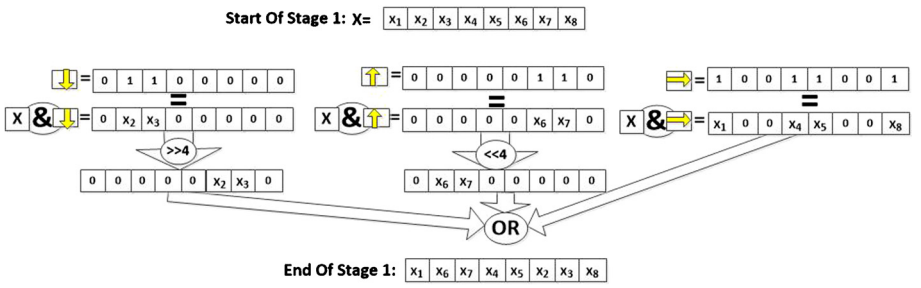

Fig. 13. Applying stage 1 of the Benes network to x.

stages of the Benes network for σ to the leftmost b bits of x. Every stage requires $O(1)$ operations giving a total of $O(\log b) = O(\log w)$ operations.

During preprocessing we prepare two $b \times (2\log b - 1)$ binary matrices Dir and C. Both Dir and C have $2b\log b - b \leq 2w$ bits, so they can fit into 2 w-bit words. The jth column of these matrices correspond to stage j of the Benes network, the ith row of these matrices corresponds to the ith input of the stage. Pictorially, we imagine that inputs are numbered top-down.

Recall that the mate of input i in stage j is some other input i' of stage j. If $(i' < i)$ we define $\mathrm{Dir}_{i,j} = 0$, otherwise, $(i' > i)$, and we define $\mathrm{Dir}_{i,j} = 1$, this is defined for $i = 1, \ldots, b$, $j = 1, \ldots, 2\log b - 1$. (Note that this matrix does not depend on the particular permutation that we encode with the Benes network, we could use a single Dir matrix for all Benes networks.)

The matrix C is computed as follows: $C_{i,j} = 0$ if input i of stage j routes to input i of stage $j+1$ (*i.e.*, goes "straight"), and $C_{i,j} = 1$ otherwise.

We pack the $b \times (2\log b - 1)$ binary matrix C into 2 w-bit words C_1, C_2 as follows:

$$C_1[1, 2, \ldots, w] \leftarrow C_{1,1}, \ldots, C_{b,1}, C_{1,2}, \ldots, C_{b,2}, \ldots, C_{\log b,1}, \ldots, C_{\log b,b};$$

$$C_2[1, 2, \ldots, w] \leftarrow C_{1,\log b+1}, \ldots, C_{b,\log b+1}, \ldots, C_{1,2\log b-1}, \ldots, C_{b,2\log b-1}.$$

We pack the matrix Dir into words Dir_1 and Dir_2 analogously.

During query processing we apply stage i (for $i = 1 \ldots \log b$) of the Benes network by computing

$$\begin{aligned} x = {} & \left(x \text{ AND } \overline{C_1[1, \ldots, b]}\right) \\ & \text{OR} \quad \left(\left(x \text{ AND } C_1[1, \ldots, b] \text{ AND } \mathrm{Dir}_1[1, \ldots, b]\right) \gg \left(b/2^i\right)\right) \\ & \text{OR} \quad \left(\left(x \text{ AND } C_1[1, \ldots, b] \text{ AND } \overline{\mathrm{Dir}_1[1, \ldots, b]}\right) \ll \left(b/2^i\right)\right). \end{aligned} \tag{1}$$

This should be parsed as follows:

- $(x \text{ AND } \overline{C_1[1, \ldots, b]})$ gives the bits of x that are not going to change position at stage i.
- $(x \text{ AND } C_1[1, \ldots, b] \text{ AND } \mathrm{Dir}_1[1, \ldots, b]) \gg (b/2^i)$ takes the bits of x that are to move "up" at stage i and shifts them accordingly.
- $(x \text{ AND } C_1[1, \ldots, b] \text{ AND } \overline{\mathrm{Dir}_1[1, \ldots, b]}) \ll (b/2^i)$ takes the bits of x that are to move "down" at stage i and shifts them accordingly.

In preparation for the next stage we also compute $C_1 = C_1 \ll b$, $\mathrm{Dir}_1 = \mathrm{Dir}_1 \ll b$ to prepare the control bits for the next stage of the Benes network.

Analogously, during stages $i = \log b + 1, \log b + 2, \ldots, 2\log b - 1$ we use the words C_2 and Dir_2 rather than C_1 and Dir_1.

An example of applying stage 1 of a Benes network of size 8 is shown in Fig. 13.

5.3.3. Permuting the $w/\log w$ leftmost blocks of the word

To operate the permutation on blocks of bits, we need masks that replicate the appropriate $C_{i,j}$ and $\mathrm{Dir}_{i,j}$ values $\log w$ times so that they operate upon all bits of the block simultaneously and not only on one single bit. To precompute and store such replications in advance requires $O(\log w)$ words of storage, and we allow, in total, only $O(1)$ words of storage for the entire bit selection. Thus, we need compute these "expansions" on the fly, and in $O(\log w)$ operations.

We split C into two matrices C^L and C^R where C^L consists of the leftmost $\log b$ columns of C padded by $\log w - \log b$ columns of zeros, and C^R consists of the rightmost $\log b - 1$ columns of C padded by $\log w - \log b + 1$ columns of zeros. Note that each of C^L and C^R has exactly $\log w$ columns. We split the matrix Dir into Dir^L and Dir^R analogously.

Previously, we packed the C and Dir matrices into words (C_1, C_2) and $(\mathrm{Dir}_1, \mathrm{Dir}_2)$, respectively, column by column. To perform block permutations we pack C^L (resp. C^R) and Dir^L (resp. Dir^R) into C_1 (resp. C_2) and Dir_1 (resp. Dir_2) row by row.

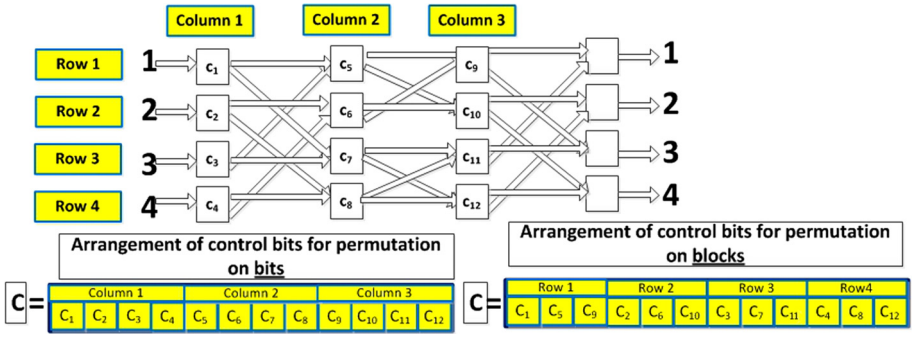

Fig. 14. Benes network control bits $C_{i,j}$ and $\mathrm{Dir}_{i,j}$.

This way the bits associated with stage j of the Benes network will be spaced out, $\log w$ bits apart. See Fig. 14. For $j \leq \log b$ these bits are in C_1 and Dir_1 and for $j > \log b$ they are in C_2 and Dir_2.

Given C_i or Dir_i, we seek to isolate and replicate the bits associated with stage $1 \leq j \leq 2\log b - 1$. We define a transformation $g : \{0,1\}^w \times \{1,\ldots,\log w\}$ such that for any w bit word z, and any $1 \leq j \leq \log w$, $g(z,j)$ is a mask such that for any block B, all bits of $g(z,j)[B]$ are equal to $z[B[j]]$.

We compute $g(z,j)$ in $O(1)$ time as follows: Let Z_j, $1 \leq j \leq \log w$, be a bit pattern with $w/\log w$ 1's at the jth index of every block. The operation $y_0 = Z_j$ AND z isolates the j'th bits of every block in z. Let $y_1 = y_0 \ll j-1$ and $y_2 = y_0 \gg (\log w - j)$, let $y_3 = y_1 - y_2$. Blocks B for which the bit $z[B[j]] = 1$, now have $y_3[B] = 011\ldots1$, blocks B for which the bit was zero now have $y_3[B]$ consisting only of zeros. Finally, set $g(z,j) = y_3$ OR y_1. Now, all bits of $g(z,j)[B]$ are equal to the bit $z[B[j]]$.

To simulate the Benes network and sort blocks rather than bits, for stages $j \leq \log b$ we use the masks $g(C_1, j)$ and $g(\mathrm{Dir}_1, j)$, analogously to our use of the masks $C_1[1,\ldots,b]$ and $\mathrm{Dir}_1[1,\ldots,b]$ in Eq. (1). For stages $j > \log b$ we use $g(C_2, j - \log b)$ and $g(\mathrm{Dir}_2, j - \log b)$ analogously to the use of $C_2[1,\ldots,b]$ and $\mathrm{Dir}_2[1,\ldots,b]$.

Given this transformation, we can simulate the Benes network in parallel, on entire blocks, and permute blocks at no greater cost than permuting *bits*.

5.4. Preprocessing bit selectors

As described above, given an input word x the actual bit selection of bits from x according to I takes $O(\log w)$ time and uses an auxiliary $O(1)$ word data structure $D(I)$ which consists of the following:

- The mask M, where bit i is one iff i appears in I. Given I, computing M takes time $O(w/\log w)$. (Used to zero irrelevant bits.)
- We need $\log w$ counters a_i, where a_i is the number of blocks (in x) that contain at least i indices from I. Computing the a_i's takes no more than $O(w/\log w)$ time, simply by processing I iteratively and transforming indices to blocks. (Used in Phase 3.)
- In Phase 6 we use $\log w$ masks M_i, each of length $w/\log w$. All these $\log w$ masks can be packed into a single w bit word. Mask M_i can be computed from M (describing the position of the bits after Phase 5). This is done by adding ones at appropriate offsets after the ones of M and takes time $O(w/\log w)$ per mask.
- In Phases 2, 5, and 7 we use the Benes network to permute blocks or bits. The $O(1)$ control words required to do so can be computed using the looping algorithm in $O(w)$ time. See Section 5.3.

To do the above precomputation we keep track of the current values of I and M. One can do so as follows: after building Phase i, we can apply it to M, (as we apply it to x during a query), this gives the new state of M in $O(\log w)$ time before the next phase.

There are no more than $\log w$ elementary operations applied to M (or x) during a single phase. For every such elementary operation, the sequence of indices I can be appropriately modified to reflect the new state of M (and x) in time that is linear in $|I| \leq w/\log w$. Thus, the overall time per phase to update I (and also over all phases) is $O(w)$.

6. Preprocessing time for α, β, and γ-nodes

In the context of a static data set, the preprocessing time is not really an issue, as long as it is polynomial.

In common to all our data structures one needs to construct a blind trie over $w/\log w$ keys of w bits. We do that in $O(w)$ time as follows. First we sort the keys ($O(w)$ time). Then we iteratively add the keys to the trie in ascending order. To add key i to the trie (currently including keys 1 to $i-1$) we find the length of longest common prefix of keys $(i-1)$ and i. (By XORing the keys and finding the most significant bit of the result using binary search in $O(\log w)$ time per key

and $O(w)$ total time without multiplication and in $O(1)$ time per key and $O(w/\log w)$ total with multiplication.) Then we add a new node to trie representing this common prefix.

We find the parent of this new node by traversing the rightmost path of the trie bottom up. We charge the work of this traversal to the nodes traversed. Each node (except the parent of the new node) is charged once as it leaves the rightmost path of the trie when we charge it. It follows that we add all these nodes in $O(w/\log w)$ time.

Constructing an α-node reduces to the construction of a special case of z-fast tries over the $w/\log w$ keys [4]. This takes $O(w)$ time.

Construction of a β-node can be done in $O(w)$ time. The basic issue is finding a prefix p such that between 1/3 to 2/3 of the keys start with p and then recurse in the appropriate subtries. We can do that by e.g. a depth first traversal of the trie in which we compute the size of the subtree of each node v (which equals to the number of keys starting with the prefix corresponding to v). We then cut out subtrie rooted by v add the appropriate key to this subtrie and to its complement subtrie and recurse. The running time is specified by the recurrence $T(n) = T(\alpha n) + T((1-\alpha)n) + O(n)$, $1/3 \leq \alpha \leq 2/3$. This recurrence gives $T(n) = O(n \log n)$. For $n = w/\log w$, $T(n) = O(w)$.

Construction of a γ-node can also be done in $O(w)$ time. The sequence of bit locations I is defined based on a trie decomposition which we compute in $O(w/\log w)$ time. Then we construct the bit selector $D(I)$ in $O(w)$ time, see Section 5.4.

7. Summary and open issues

The following theorem summarizes our results.

Theorem 7.1. *Given a set of $w/\log w$ w-bit keys, we've shown the following $O(1)$ word indices for computing predecessor queries:*

- *α-nodes and β-nodes: predecessor search in $O(1)$ probes and $O(\log w)$ time, both with high probability. Worst case predecessor search requires $O(\log w)$ probes and time. Preprocessing takes $O(w)$ time.*
- *γ-nodes: Worst case predecessor search requires $O(1)$ probes and $O(\log w)$ time. Does not require multiplication. Preprocessing takes $O(w)$ time.*

We also suggest the following open issues:

1. Our (k,k)-bit selector takes $O(\log w)$ operations, which are optimal when $k \geq w^{\epsilon}$ for any constant $\epsilon > 0$. What can be done for smaller values of k? (*E.g.*, for $k = O(1)$ one can definitely do better.)
2. It follows from Thorup [20] that, in the practical-RAM model, a search node with fan-out $w/\log w$ requires $\Omega(\log\log w)$ operations. Our γ-nodes have fan out $w/\log w$ and require $O(\log w)$ operations. Can this gap be bridged?
3. A natural open question is if the additive $O(\log w)$ in time complexity is required or not.

Acknowledgments

We wish to thank Nir Shavit for introducing us to the problems of contention in multicore environments, for posing the question of multicore efficient data structures, and for many useful discussions. We also wish to thank Mikkel Thorup for his kindness and useful comments.

References

[1] Miklós Ajtai, Michael L. Fredman, János Komlós, Hash functions for priority queues, Found. Comput. Sci. 24 (1983) 299–303.
[2] A. Andersson, P.B. Miltersen, M. Thorup, Fusion trees can be implemented with AC(0) instructions only, Theor. Comput. Sci. 215 (1–2) (1999) 337–344.
[3] P. Beame, F.E. Fich, Optimal bounds for the predecessor problem and related problems, J. Comput. Syst. Sci. 65 (1) (2002) 38–72.
[4] D. Belazzougui, P. Boldi, R. Pagh, S. Vigna, Monotone minimal perfect hashing: searching a sorted table with o(1) accesses, in: SODA, 2009, pp. 785–794.
[5] D. Belazzougui, P. Boldi, R. Pagh, S. Vigna, Fast prefix search in little space, with applications, in: ESA, 2010, pp. 427–438.
[6] D. Belazzougui, P. Boldi, R. Pagh, S. Vigna, Theory and practice of monotone minimal perfect hashing, ACM J. Exp. Algorithmics 16 (2011) 3.
[7] A. Brodnik, P.B. Miltersen, J.I. Munro, Trans-dichotomous algorithms without multiplication – some upper and lower bounds, in: WADS, 1997, pp. 426–439.
[8] U. Drepper, What every programmer should know about memory, http://lwn.net/Articles/250967/, 2007.
[9] P. Ferragina, R. Grossi, The string B-tree: a new data structure for string search in external memory and its applications, J. ACM 46 (1999) 236–280.
[10] G. Franceschini, J.I. Munro, Implicit dictionaries with o(1) modifications per update and fast search, in: SODA, ACM Press, 2006, pp. 404–413.
[11] M.L. Fredman, D.E. Willard, Trans-dichotomous algorithms for minimum spanning trees and shortest paths, Found. Comput. Sci. (1990) 719–725.
[12] M.L. Fredman, D.E. Willard, Surpassing the information theoretic bound with fusion trees, J. Comput. Syst. Sci. 47 (3) (1993) 424–436.
[13] R. Grossi, A. Orlandi, R. Raman, S.S. Rao, More haste, less waste: lowering the redundancy in fully indexable dictionaries, in: STACS, 2009, pp. 517–528.
[14] P.B. Miltersen, Lower bounds for static dictionaries on RAMs with bit operations but no multiplication, in: ICALP, 1996, pp. 442–453.
[15] J.I. Munro, R. Raman, V. Raman, S.S. Rao, Succinct representations of permutations, in: ICALP, 2003, pp. 345–356.
[16] M. Pătraşcu, M. Thorup, Time-space trade-offs for predecessor search, in: STOC, 2006, pp. 232–240.
[17] R. Raman, Priority queues: small, monotone and trans-dichotomous, in: ESA, 1996, pp. 121–137.
[18] R. Raman, V. Raman, S.R. Satti, Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets, ACM Trans. Algorithms 3 (4) (Nov. 2007).

[19] N. Shavit, Data structures in the multicore age, Commun. ACM 54 (3) (2011) 76–84.
[20] M. Thorup, On AC0 implementations of fusion trees and atomic heaps, in: SODA, 2003, pp. 699–707.
[21] P. van Emde Boas, Preserving order in a forest in less than logarithmic time and linear space, Inf. Process. Lett. 6 (3) (1977) 80–82.
[22] D.E. Willard, Log-logarithmic worst-case range queries are possible in space $\Theta(n)$, Inf. Process. Lett. 17 (2) (1983) 81–84.